System Analysis and Modeling Group
Hasso Plattner Institute for Digital Engineering
University of Potsdam
Potsdam, Germany

Model-Driven Engineering of Self-Adaptive Software

Dissertation
zur Erlangung des akademischen Grades
“doctor rerum naturalium”
- Dr. rer. nat. -

eingereicht an der
Digital-Engineering-Fakultät
des Hasso-Plattner-Instituts und
der Universität Potsdam

von
Thomas Vogel

March 2018

Thomas Vogel:
Model-Driven Engineering of Self-Adaptive Software
March 2018

Published online at the
Institutional Repository of the University of Potsdam:
URN urn:nbn:de:kobv:517-opus4-409755
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-409755

ABSTRACT

The development of self-adaptive software requires the engineering of an adaptation engine that controls the underlying adaptable software by a feedback loop. State-of-the-art approaches prescribe the feedback loop in terms of numbers, how the activities (*e. g.*, monitor, analyze, plan, and execute (MAPE)) and the knowledge are structured to a feedback loop, and the type of knowledge. Moreover, the feedback loop is usually hidden in the implementation or framework and therefore not visible in the architectural design. Additionally, an adaptation engine often employs runtime models that either represent the adaptable software or capture strategic knowledge such as reconfiguration strategies. State-of-the-art approaches do not systematically address the interplay of such runtime models, which would otherwise allow developers to freely design the entire feedback loop.

This thesis presents ExecUtable RuntimE MegAmodels (EUREMA), an integrated model-driven engineering (MDE) solution that rigorously uses models for engineering feedback loops. EUREMA provides a domain-specific modeling language to specify and an interpreter to execute feedback loops. The language allows developers to freely design a feedback loop concerning the activities and runtime models (knowledge) as well as the number of feedback loops. It further supports structuring the feedback loops in the adaptation engine that follows a layered architectural style. Thus, EUREMA makes the feedback loops explicit in the design and enables developers to reason about design decisions.

To address the interplay of runtime models, we propose the concept of a runtime megamodel, which is a runtime model that contains other runtime models as well as activities (*e. g.*, MAPE) working on the contained models. This concept is the underlying principle of EUREMA. The resulting EUREMA (mega)models are kept alive at runtime and they are directly executed by the EUREMA interpreter to run the feedback loops. Interpretation provides the flexibility to dynamically adapt a feedback loop. In this context, EUREMA supports engineering self-adaptive software in which feedback loops run independently or in a coordinated fashion within the same layer as well as on top of each other in different layers of the adaptation engine. Moreover, we consider preliminary means to evolve self-adaptive software by providing a maintenance interface to the adaptation engine.

This thesis discusses in detail EUREMA by applying it to different scenarios such as single, multiple, and stacked feedback loops for self-repairing and self-optimizing the mRUBiS application. Moreover, it investigates the design and expressiveness of EUREMA, reports on experiments with a running system (mRUBiS) and with alternative solutions, and assesses EUREMA with respect to quality attributes such as performance and scalability.

The conducted evaluation provides evidence that EUREMA as an integrated and open MDE approach for engineering self-adaptive software seamlessly integrates the development and runtime environments using the same formalism to specify and execute feedback loops, supports the dynamic adaptation of feedback loops in layered architectures, and achieves an efficient execution of feedback loops by leveraging incrementality.

ZUSAMMENFASSUNG

Die Entwicklung von selbst-adaptiven Softwaresystemen erfordert die Konstruktion einer geschlossenen *Feedback Loop*, die das System zur Laufzeit beobachtet und falls nötig anpasst. Aktuelle Konstruktionsverfahren schreiben eine bestimmte Feedback Loop im Hinblick auf Anzahl und Struktur vor. Die Struktur umfasst die vorhandenen Aktivitäten der Feedback Loop (z. B. Beobachtung, Analyse, Planung und Ausführung einer Adaption) und die Art des hierzu verwendeten Systemwissens. Dieses System- und zusätzlich das strategische Wissen (z. B. Adaptionsregeln) werden in der Regel in Laufzeitmodellen erfasst und in die Feedback Loop integriert. Aktuelle Verfahren berücksichtigen jedoch nicht systematisch die Laufzeitmodelle und deren Zusammenspiel, so dass Entwickler die Feedback Loop nicht frei entwerfen und gestalten können. Folglich wird die Feedback Loop während des Entwurfs der Softwarearchitektur häufig nicht explizit berücksichtigt.

Diese Dissertation stellt mit EUREMA ein neues Konstruktionsverfahren für Feedback Loops vor. Basierend auf Prinzipien der modellgetriebenen Entwicklung (MDE) setzt EUREMA auf die konsequente Nutzung von Modellen für die Konstruktion, Ausführung und Adaption von selbst-adaptiven Softwaresystemen. Hierzu wird eine domänenspezifische Modellierungssprache (DSL) vorgestellt, mit der Entwickler die Feedback Loop frei entwerfen und gestalten können, d. h. ohne Einschränkung bezüglich der Aktivitäten, Laufzeitmodelle und Anzahl der Feedback Loops. Zusätzlich bietet die DSL eine Architektursicht auf das System, die die Feedback Loops berücksichtigt. Daher stellt die DSL Konstrukte zur Verfügung, mit denen Entwickler während des Entwurfs der Architektur die Feedback Loops explizit definieren und berücksichtigen können.

Um das Zusammenspiel der Laufzeitmodelle zu erfassen, wird das Konzept eines sogenannten *Laufzeitmegamodells* vorgeschlagen, das alle Aktivitäten und Laufzeitmodelle einer Feedback Loop erfasst. Dieses Konzept dient als Grundlage der vorgestellten DSL. Die bei der Konstruktion und mit der DSL erzeugten (Mega-)Modelle werden zur Laufzeit bewahrt und von einem Interpreter ausgeführt, um das spezifizierte Adaptionsverhalten zu realisieren. Der Interpreteransatz bietet die notwendige Flexibilität, um das Adaptionsverhalten zur Laufzeit anzupassen. Dies ermöglicht über die Entwicklung von Systemen mit mehreren Feedback Loops auf einer Ebene hinaus das Schichten von Feedback Loops im Sinne einer adaptiven Regelung. Zusätzlich bietet EUREMA eine Schnittstelle für Wartungsprozesse an, um das Adaptionsverhalten im laufendem System anzupassen.

Die Dissertation diskutiert den EUREMA-Ansatz und wendet diesen auf verschiedene Problemstellungen an, u. a. auf einzelne, mehrere und koordinierte als auch geschichtete Feedback Loops. Als Anwendungsbeispiel dient die Selbstheilung und Selbstoptimierung des Online-Marktplatzes mRUBiS. Für die Evaluierung von EUREMA werden Experimente mit dem laufenden mRUBiS und mit alternativen Lösungen durchgeführt, das Design und die Ausdrucksmächtigkeit der DSL untersucht und Qualitätsmerkmale wie Performanz und Skalierbarkeit betrachtet. Die Ergebnisse der Evaluierung legen nahe, dass EUREMA als integrierter und offener Ansatz für die Entwicklung selbst-adaptiver Softwaresysteme folgende Beiträge zum Stand der Technik leistet: eine nahtlose Integration der Entwicklungs- und Laufzeitumgebung durch die konsequente Verwendung von Modellen, die dynamische Anpassung des Adaptionsverhaltens in einer Schichtenarchitektur und eine effiziente Ausführung von Feedback Loops durch inkrementelle Verarbeitungsschritte.

ACKNOWLEDGMENTS

I owe my deepest gratitude to my advisor Holger Giese, from whom I learned a lot on the way to my Ph.D., for giving me the opportunity to pursue my research and dissertation, for his patient support, encouragement, and guidance, as well as for the passionate discussions we had. Furthermore, I would like to thank him for giving me the freedom to collaborate with other researchers in side projects, through which I obtained a broader view of the research field while working on the focused topic of my dissertation. I would also like to thank Betty H.C. Cheng and Samuel Kounev for reviewing my dissertation.

Additionally, I would like to thank Kerstin Miers for her administrative support, which has made my daily work at the HPI easier.

My special thanks go to Leen Lambers and Basil Becker without whom the experience of pursuing a Ph.D. would not have been the same. I enjoyed a lot the technical and personal discussions we had during work and coffee breaks and I always appreciated their opinions. I am especially grateful to Sona Ghahremani and Joachim Hänsel for the enjoyable research collaboration, to Dominique Blouin for his technical hints on implementing model-driven engineering tools, to Joachim Hänsel for notably improving the quality of coffee in our research group, and to Ulf Hansen for all the fun during lunch breaks.

I would like to thank my parents, Ludmila and Robert, for continuously encouraging me to keep moving along my path. Finally, all my thanks go to Steffi who is always there if I need help and advice and who considerably motivated and supported me in pursuing and eventually finishing my Ph.D.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AS	Adaptation Strategy
ADL	Architecture Description Language
API	Application Programming Interface
ATL	ATLAS Transformation Language
CF	Critical Failure
CIM	Computation Independent Model
CompArch	Component Architecture Language
CPU	Central Processing Unit
CSL	Continuous Stochastic Logic
CTMC	Continuous Time Markov Chain
CWM	Common Warehouse Metamodel
DTMC	Discrete Time Markov Chain
DSL	Domain-Specific Language
ECA	Event-Condition-Action
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
EUREMA	Executable Runtime Megamodels
FIFO	First-In, First-Out
FLD	Feedback Loop Diagram
GPL	General Purpose Language
JavaCC	Java Compiler Compiler
Java EE	Java Enterprise Edition
JMX	Java Management Extensions
LD	Layer Diagram
LTL	Linear Temporal Logic
MAPE	Monitor, Analyze, Plan, and Execute
MAPE-K	Monitor, Analyze, Plan, Execute, and Knowledge
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta Object Facility
mRUBiS	Modular Rice University Bidding System
NAC	Negative Application Condition
OCL	Object Constraint Language
OMG	Object Management Group

PCTL	Probabilistic Computation Tree Logic
PI	Performance Issue
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
RUBiS	Rice University Bidding System
SD	Story Diagram
SDM	Story-Driven Modeling
SLOC	Source Lines of Code
SP	Story Pattern
SPEM	Software & Systems Process Engineering Meta-Model
TGG	Triple Graph Grammar
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

1 INTRODUCTION

1.1 MOTIVATION

Software has pervaded our everyday lives for over the past thirty years [250] and drastically shaped our society and economy [105]. It has become a driver of innovation and therefore an essential component of many products and services in different domains such as vehicles in transportation, devices in communication and health, machines in production, or applications in business administration [33]. Therefore, we highly depend on software everyday [387] such that "software systems must become more versatile, flexible, resilient, dependable, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changes that may occur in their operational contexts, environments and system requirements" [16, p. 1].

Adapting software is typically addressed by *software evolution* [297–299], that is, maintaining a software product after its delivery and deployment by modifying it to correct problems such as faults, improve properties such as performance, address changes in the operational environment, and meet changing requirements [392, 405]. Such modifications are required because software ages [339] and needs to be continually adapted to remain satisfactory for the user (*cf.* Lehman's Laws of Software Evolution [270–275]). The need and importance of maintaining software becomes apparent considering that maintenance costs are usually higher than development costs for custom systems [394].

This situation changes for the worse due to the growing complexity of computing systems that continue to automate tasks and processes of our daily lives in order to improve the productivity of individuals and businesses [225]. However, such an evolution by automation produces complexity to an extent that it "threatens to undermine the very benefits information technology aims to provide" [225, p. 4].

In addition to the complexity impeding evolution, software systems often have to adapt to individual contexts as perceived in ubiquitous and pervasive computing [372, 431] such as internet of things [345]. Thus, adaptation of software systems takes place at the level of individual deployments of the system rather than for all instances of the system.

Finally, software systems that are developed independently from each other are integrated to ultra-large-scale systems [321] or systems of systems [415], which requires dynamic adaptation to realize the integration in contexts that are not known a priori (*cf.* [442]).

Typical maintenance processes (*cf.* [249, 348, 353, 394]) that are performed mostly manually, decoupled from the runtime environment, and during scheduled down-times of systems do not meet such requirements to adapt systems. For instance, manual adaptations are difficult and expensive due to the size and complexity of systems that prevent a complete shut-down of a system for maintenance. Context-aware systems require timely adaptations coupled to the runtime environment, or mission-critical systems have to be permanently available.

Therefore, innovative ways of adapting and thus of developing, deploying, operating, and evolving such software systems are required. This has led to the research trends of *autonomic computing* [168, 225, 246] and *self-adaptive software* [112, 16]. The former considers "computing systems that can manage themselves given high-level objectives from admin-

istrators" [246, p. 41], hence taking an administration perspective of adapting systems. In contrast, the latter takes a broader software engineering perspective, as it is used in this thesis, and addresses "systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals" [16, p. 1]. Such self-adaptation is usually realized by software [112, 16] making the software engineering perspective relevant.

Thus, the aim of self-adaptive software is to automate adaptation and shift the responsibility for adaptation from developers or maintenance engineers to the software system itself. Eventually, the promise of such self-adaptation is to cope with the growing costs, complexity, and diversity of evolving software systems.

1.2 PROBLEM STATEMENT

The systematic development of self-adaptive software is an open research problem [112, 16] and despite existing research approaches, "there is still a lack of powerful languages, tools, and frameworks that could help realize adaptation processes and instrument sensors/effectors in a systematic manner" [370, p. 31]. In this context, most research approaches follow the *external approach* [370] that separates the self-adaptive software into the *adaptable software* and the *adaptation engine*. In between both, a *feedback loop* ensures that the adaptation engine dynamically adjusts the adaptable software if needed. This separation decouples the engine from the adaptable software but it makes the feedback loop a crucial element of the overall software architecture [93, 311, 386] and therefore essential for engineering self-adaptive software with the external approach. Particularly, Shaw [386] argues that typical software engineering paradigms such as object orientation are not the best option to architect software systems with feedback loops, which calls for other means to design and develop such systems. A popular blueprint for a feedback loop is the MAPE-K cycle consisting of four *adaptation activities* (monitor, analyze, plan, and execute) that operate on some shared *knowledge* [246].

Besides the trend to self-adaptation, the idea of *Models@run.time* [64, 79, 164] has emerged more recently. It equips feedback loops with causally connected runtime models of the adaptable software. Feedback loops then operate on such models (*cf. model-based adaptation* [370]) rather than directly on the adaptable software while exploiting benefits of Model-Driven Engineering (MDE) such as abstraction, automation, and analysis for self-adaptation.

In this thesis, we focus on the engineering of self-adaptive software using the external approach and models at runtime. To address the systematic development of such systems, we need specific means to design, implement, run, and maintain adaptation engines and feedback loops. This imposes requirements for engineering self-adaptive software, particularly for a modeling language to specify and design such software.

As argued by several researchers [93, 215, 251, 311, 386], feedback loops should be made explicit in the architectural design and analysis of self-adaptive software. This concerns several aspects such as the elements of a feedback loop (modularity), how these elements may vary in the design (variability), how these elements interact with each other (intra-loop coordination), when a feedback loop should be executed, if and how a feedback loop interacts with other feedback loops (inter-loop coordination), and how multiple feedback loops are distributed in the system. Additionally, the runtime models used by feedback loops should be made explicit as they correspond to the data processed by the loops.

Using runtime models requires that the feedback loop realizes the causal connection between the models and the adaptable software based on the sensors and effectors provided by the software (*cf.* [79, 285]). Sensors are used to update the runtime models if the software changes and effectors are used to change the software according to planned adaptations performed on the runtime models. Hence, the use of the sensors and effectors by the feedback loops has to be addressed when engineering these feedback loops.

To increase the flexibility of feedback loops, they may operate on top of each other resulting in layers of feedback loops as, for instance, proposed by Kramer and Magee [258] in their reference architecture for self-managed systems. In such layered architectures, a feedback loop at a higher layer can adapt the feedback loops at the layer directly below. Supporting such layered architectures requires providing some form of reflection of lower-layer, adaptable feedback loops that is used by higher-layer feedback loops for adaptation.

Though self-adaptation promises that the software adjusts itself by automating adaptation activities otherwise performed off-line for evolution, we cannot expect that software is able to cope with *all* needs for evolution and to fully automate *all* kinds of off-line activities with feedback loops. Thus, besides an adaptation engine realizing (on-line) self-adaptation, the engine's co-existence with off-line adaptation such as typical maintenance is required [1, 54, 135, 167, 208, 26, 433]. This calls for supporting the co-existence of on-line and off-line adaptation for the long-term evolution of self-adaptive software.

Based on the specification and design of an adaptation engine with its feedback loops, the concurrent execution of the feedback loops should be supported. The execution should be based on states as reflected by the runtime models describing the adaptable software and on events describing changes of the adaptable software. The execution should be runtime-efficient since self-adaptation works on-line and it should not significantly disturb the performance of the adaptable software.

1.3 APPROACH, CONTRIBUTIONS, AND THE STATE OF THE ART

This thesis presents *ExecUtable RuntimE MegAmodels* (EUREMA), a Model-Driven Engineering (MDE) approach that covers the specification, execution, and evolution of adaptation engines for self-adaptive software with multiple feedback loops. EUREMA is based on our idea of a *runtime megamodel* [31, 32] that captures the individual adaptation activities and runtime models of a feedback loop and thus the whole feedback loop. Moreover, such a megamodel is *executable* such that it can be directly executed to run the feedback loop.

Accordingly, EUREMA consists of a modeling language and a runtime interpreter. The language supports the development of adaptation engines by providing a domain-specific modeling solution for feedback loops that may use arbitrary kinds of adaptation activities and runtime models including problem-space oriented models. Thereby, the language addresses how feedback loops are connected to the adaptable software by sensors and effectors in an architectural view. Besides single feedback loops, the language covers multiple feedback loops in the same or different layers of the architecture. Based on the layered architecture, the same language as used for specifying feedback loops is employed to specify off-line adaptations, that is, changes due to maintenance, and their enactment in the running self-adaptive software system.

The models created with the EUREMA language are kept alive at runtime and are directly executed by the EUREMA interpreter to run adaptation engines and feedback loops. While the interpreter-based approach enables the flexibility to dynamically adapt and evolve feedback loops, it further executes the feedback loops efficiently by introducing

a low overhead and leveraging state- and event-based principles for an incremental execution. These principles are enabled by model-driven adaptation activities that additionally permit reuse of existing MDE tools and techniques for their implementation. Moreover, EUREMA keeps the models that specify such activities alive at runtime for execution.

Thus, EUREMA models explicitly capture and maintain the runtime models used within an adaptation engine, the interplay between these runtime models, and the adaptation activities forming a feedback loop and working on these runtime models. Therefore, the evolution of runtime models and feedback loops continues at runtime beyond the initial development of the software. The same EUREMA models are also used for executing the feedback loops enabling a seamless transition from the design and specification to the execution as well as the adaptation and evolution of feedback loops.

Therefore, EUREMA addresses the requirements for modeling, executing, and evolving feedback loops and it provides the following contributions:

C1 – Integrated MDE Approach

EUREMA is an *integrated* MDE approach as it rigorously and consistently uses models for engineering feedback loops. Models are used(1) to represent the adaptable software to perform self-adaptation as proposed by the idea of Models@run.time, as well as to engineer (2) individual adaptation activities of feedback loops, (3) feedback loops as a whole, and (4) the coordination among feedback loops. Finally, (5) these models are used throughout the self-adaptive system's life cycle, that is, for specifying, executing, and evolving the feedback loops with their adaptation activities.

As a consequence, EUREMA allows engineers to make the feedback loops with their adaptation activities, runtime models, and coordination to other feedback loops explicit in the design of self-adaptive software. Thereby, EUREMA does not impose any restrictions on the design of feedback loops. On the contrary, engineers are free to decide on the structure and number of feedback loops and how feedback loops are placed in an arbitrary number of layers. This allows engineers to create feedback loops that are entirely and individually designed for a specific problem at hand.

The state of the art[1] primarily focuses on the aspects (1) and (2). However, with respect to (1) the models are often related to the solution space of the adaptable software, and (2) the models are primarily used for the analysis and planning activities of the feedback loop (*e. g.*, Event-Condition-Action (ECA) rules). In contrast, we consider runtime models related to the problem space and the use of models for all activities of a feedback loop including the monitoring and execution steps. Moreover, the state of the art neglects the use of models for engineering (3) feedback loops and (4) their coordination, that is, feedback loops are not made explicit in the design and consequently, (5) corresponding models are not and cannot be used after development and deployment for executing, coordinating, and evolving feedback loops. Hence, the state of the art does not consequently use models throughout the system's life cycle.

Thus, the state of the art uses models for (1) and (2) to partially specify feedback loops. For execution, these models are used either directly by frameworks or by generators to produce incomplete code. In either case, the frameworks or the generators usually prescribe the structure and number of feedback loops (typically, one) as their goal is to ease development by reuse. Such approaches consequently prevent a free design of feedback loops since the corresponding design decisions are already made in the framework or generator (*cf.* [355]).

1 The state of the art in engineering self-adaptive software will be discussed in detail and backed with references in the context of related work in Chapter 10.

C2 – Open Approach

Though being an integrated MDE approach, EUREMA is an *open* approach as it does not restrict the types of models to be used for specifying and executing individual adaptation activities of a feedback loop and the kind of runtime models that represent the adaptable software. Thus, EUREMA is open for any modeling language to create the corresponding models. The EUREMA interpreter then maintains and uses these models to execute the overall feedback loops as specified with the EUREMA language.

Such an openness supports the model-driven development (*cf.* [164]) of individual adaptation activities, the direct execution of the corresponding models, and the reuse of existing MDE tools and techniques.[2] This finally eases the development of the adaptation activities similarly to frameworks for self-adaptive software that support reuse.

EUREMA promotes the use of *executable* languages for specifying the adaptation activities such that the resulting models can be maintained at runtime and directly executed (*cf.* C1 and C3) and are amenable for adaptation and evolution (*cf.* C4). However, it further allows the (re)use of non-executable languages with potentially existing code generators or any other method, even those that are not driven by models, to develop adaptation activities. EUREMA is even open for completely code-based adaptation activities.

The state of the art which are typically frameworks for self-adaptive software that use runtime models are closed as they prescribe the types of models and techniques to be used. Thus, engineers have no choice in selecting the languages to express the runtime model representing the adaptable software and the executable models that specify and run the adaptation activities.

C3 – Seamless Integration of Development and Runtime Environments

EUREMA models created for specifying feedback loops with their adaptation activities and runtime models are directly kept alive at runtime to execute the feedback loops. This avoids bridging the conceptual gap (*cf.* [164]) between domain-oriented specifications such as EUREMA models and implementation technologies such as code. In contrast, EUREMA seamlessly integrates the development and runtime environments (*cf.* [55]) as it allows the use of the same models in both environments and the exchange of these model between both environments. The same holds for (runtime) models used within the feedback loops, that is, models specifying adaptation activities or representing the adaptable software. Additionally, the executability makes EUREMA models amenable for simulation to analyze the self-adaptation in the development environment.

The state of the art does not keep alive their feedback loop specifications at runtime but often require semi-automated translation steps between the development and runtime environments. The focus of individual state-of-the-art approaches is typically either on the development or runtime environment but not on both. For instance, languages dedicated to modeling self-adaptive software often do not provide any execution support.

C4 – Adaptation and Evolution of Feedback Loops

Tackling an abstraction level similar to software architectures[3], explicitly maintaining models used within feedback loops at runtime, and keeping EUREMA models alive at runtime

2 For the EUREMA application examples, various types of models and their corresponding execution engines are (re)used without any modifications, for instance, Triple Graph Grammars (TGGs) [377], Story Diagrams (SDs)/Story Patterns (SPs) [155], and the Object Constraint Language (OCL) [327].

3 Several researchers argue that the software architecture is the appropriate abstraction level to deal with self-adaptation [86, 169, 227, 258, 286, 334, 335].

for an interpretation-based execution, EUREMA provides beneficial abstractions and the required flexibility to dynamically adapt (self-adapt) feedback loops in layered architectures and to evolve running feedback loops by means of off-line adaptation.

Therefore, EUREMA allows engineers to build adaptive control [50] architectures, in which higher-layer feedback loops dynamically change other lower-layer feedback loops without restricting the number of feedback loops and layers. Furthermore, it allows developers to evolve the adaptation engine by manually changing the feedback loops. Thereby, EUREMA coordinates the execution of changes originating from feedback loops and developers, an initial step toward the unification of self-adaptation and evolution (*cf.* [1]). Thus, EUREMA enables changes of feedback loops that are either initiated by self-adaptation or evolution.

The state of the art typically does not allow automatic changes of feedback loops as it does not support adaptive control architectures but focuses on a single feedback loop performing self-adaptation. Moreover, it rather neglects the need for evolving self-adaptive software by means of traditional maintenance (*i. e.*, off-line adaptation). If maintenance is supported, the related changes are constrained by the design decisions that a framework leaves open for the developer. Consequently, maintenance changes typically cannot modify the structure and number of feedback loops that are prescribed by the framework.

C5 – State- and Event-Based Feedback Loops

EUREMA supports feedback loops that are based on the state of the adaptable software *and* on events.

In state-based feedback loops, the individual adaptation activities operate on the runtime model that represents the state of the adaptable software to perform self-adaptation. Therefore, the activities have to process the whole state, for instance, to analyze the adaptable software. In contrast, adaptation activities of event-based feedback loops only use events to exchange and process knowledge about the adaptable software. Events contain information about changes of the state rather than the complete state of the adaptable software. Nevertheless, events must contain fractions of the state that correspond to the context of the change such that the activities can process the events in a meaningful way. Consequently, event-based activities process changes of the state rather than the (complete) state, which enables incremental processing. Ghezzi [180, p. 369] has identified incrementality as a key need for self-adaptation that "comes into play because often changes are local to restricted parts" of the adaptable software.

EUREMA combines both variants and thus leverages incremental processing of adaptation activities while using runtime models. On the one hand, the incrementality improves the runtime performance of executing feedback loops. On the other hand, using a runtime model avoids complicated events that carry the context of a state change as the model represents the state of the adaptable software and therefore also the context of the change. In this regard, we conceive a better changeability of self-adaptive software when using runtime models and events. For instance, if the context of a change type should be extended, it is only required to adjust the runtime model to cover the extended context to be used by all adaptation activities. In contrast, if we rely only on events, we have to adjust the individual events of all adaptation activities, thus resulting in a higher number of adjustments.

The state of the art often addresses either state- or event-based feedback loops and therefore does not combine the benefits of both variants as EUREMA does. In general, research on Models@run.time is focused more on state-based feedback loops while autonomic computing rather concentrates on event-based feedback loops.

In general, EUREMA heavily exploits MDE principles to provide appropriate abstractions and flexibility for specifying, adapting, and evolving feedback loops, automation by means of the EUREMA interpreter to execute EUREMA models and seamlessly connect the development and runtime environments, and analysis by means of simulation of feedback loops. Additionally, EUREMA promotes MDE for developing individual adaptation activities of feedback loops. Thus, developers using EUREMA can exploit MDE principles at this level, for instance, by reusing existing modeling and execution techniques for the monitoring, analysis, planning, and execution steps. The resulting models are then kept alive at runtime and maintained by the EUREMA interpreter.

While an MDE approach is obvious to address certain requirements such as the specification of feedback loops, keeping those specification models (including those models specifying individual adaptation activities of feedback loops) alive at runtime for execution, adaptation, and evolution contrasts other proposals. For instance, other approaches propose languages to specify self-adaptive software but they do not keep their models of feedback loops alive at runtime (*e. g.*, [34, 187, 262, 280]) or they suggest developing feedback loops as component-based systems that can be dynamically adapted similarly to the adaptable software [316, 347]. However, we believe that we can achieve a higher degree of flexibility when following a rigorous MDE approach that leverages runtime models to an extreme extent. Moreover, EUREMA contrasts frameworks (*e. g.*, [40, 170, 175, 202, 306]) by allowing developers to freely design feedback loops that are otherwise prescribed in the framework implementation. This combination of supporting a free design of feedback loops (including the adaptation activities and runtime models), making the feedback loops explicit in the design, as well as keeping the feedback loop specifications alive at runtime for execution, adaptation, and evolution makes EUREMA unique.

The research conducted in the context of this thesis generally follows the design science research methodology by Peffers et al. [344] that consists of six steps. We identified and motivated the research problem of engineering self-adaptive software (step 1). We transformed the problem into requirements for engineering self-adaptive software that are the objectives for solutions to the problem (step 2). Based on the requirements, we designed and developed EUREMA, that is, one solution to the problem (step 3). We demonstrated the use of EUREMA to solve instances of the problem (step 4) and evaluated EUREMA to provide evidence for its contributions (step 5). Finally, we communicated our research on EUREMA with this thesis and earlier in scientific papers, especially [21, 25, 31, 32] (step 6).

The first five steps of the methodology are addressed in different chapters of this thesis, such as step 1 in this chapter, step 2 in Chapter 3, step 3 in Chapters 4–8, step 4 in Chapters 5 and 9, and step 5 in Chapters 9 and 10. A detailed structure of the thesis is given in the following.

1.4 STRUCTURE OF THE THESIS

The rest of the thesis is structured into three parts. In the first part, we present the relevant foundations of EUREMA, which originate from the research fields of model-driven engineering and self-adaptive software (*cf.* Chapter 2). Moreover, we thoroughly discuss requirements for engineering self-adaptive software and particularly feedback loops that are driven by runtime models in Chapter 3.

We start the second part of the thesis by providing an overview of EUREMA in Chapter 4. This overview outlines the general idea as well as the roles of runtime models, runtime megamodels, and reflection in EUREMA. Moreover, it introduces the Modular Rice

University Bidding System (mRUBiS) as the running example that we use throughout this thesis. Afterwards, we discuss in detail the EUREMA language and its usage to model self-adaptive software (*cf.* Chapter 5). For this purpose, we show the use of the language to specify individual feedback loops, multiple coordinated feedback loops, stacked feedback loops in layered architectures, and finally off-line adaptation. These aspects are illustrated by EUREMA models defining the self-repair and self-optimization of mRUBiS.

While focusing on the modeling in Chapter 5, we discuss the execution of EUREMA models and therefore of EUREMA-based feedback loops in Chapter 6. For this purpose, we define the execution semantics of EUREMA, that is, how an EUREMA-based feedback loop is triggered and executed, how it interacts with the adaptable software and other feedback loops, and how it is safely adapted in layered architectures or by off-line adaptation.

With Chapter 7, the last chapter of the second part, we discuss the model-driven adaptation activities of feedback loops and the solution space for their implementation. In this context, we distinguish between code-based and model-driven activities and between state-based and event-based activities.

In the last part of the thesis, we validate EUREMA and provide the corresponding conclusions. For this purpose, we discuss the implementation of EUREMA in Chapter 8. The implementation covers the language, editor, and interpreter of EUREMA. We further present a framework that we developed as a basis to conduct experiments with EUREMA-based feedback loops, while EUREMA is completely independent of the framework and thus applicable in other technical contexts.

In Chapter 9, we comprehensively evaluate EUREMA considering different dimensions. First, we assess the design of the EUREMA language by discussing design guidelines for domain-specific modeling languages. Second, we investigate the expressiveness of the language by applying it to examples from literature. Third, we report on experiments to develop and execute EUREMA-based feedback loops for the self-repair, the self-optimization, the coordinated self-repair and self-optimization, and the three-layer architecture of mRUBiS. That is, we realized the running example that we used particularly in Chapter 5 to discuss the EUREMA language. Forth, we conducted a study in which we compared EUREMA-based feedback loops to alternative solutions developed by students with respect to development costs and runtime performance. Fifth, we discuss and evaluate how well EUREMA covers the quality attributes proposed by Asadollahi et al. [48] to assess development approaches for self-adaptive software. Finally, we discuss how well EUREMA fulfills the requirements introduced in Chapter 3.

We further use the same requirements to discuss the related work of EUREMA, which allows us to compare EUREMA to the state of the art in engineering self-adaptive software in Chapter 10. Finally, we summarize the EUREMA approach and provide conclusions and an outlook on future work in Chapter 11.

Part I

FOUNDATIONS AND REQUIREMENTS

In this part of the thesis, we discuss the foundations of the presented work, particularly, model-driven engineering and self-adaptive software. Then we introduce and discuss the requirements for engineering self-adaptive software that we have derived from literature. These requirements address the modeling and the execution of self-adaptive software systems.

2 FOUNDATIONS

The foundations of the presented work originate from the research fields of *model-driven engineering* and *self-adaptive software*. We will discuss both fields in this chapter as far as they are relevant for the presented work.

2.1 MODEL-DRIVEN ENGINEERING

According to France and Rumpe [164, p. 2], "[t]he term Model-Driven Engineering (MDE) is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations."[1] Thus, models are not only used for documenting the software to be developed but they become the primary artifacts of development [70, 164, 375] that are incrementally refined and transformed to the final product [294, 382], that are maintained along with code [243] to increase their longevity [51], and that are automatically processed by tools [71]. Such a use of models is motivated by the complexity of software systems that should, among others, run in distributed and embedded environments, target various devices and platforms, and operate dependably, and by the conceptual gap between the problem and solution spaces [164]. The problem space refers to the concepts in application domains such as telecommunication, insurance, or health, while the solution space refers to the domain of computing and implementation technologies such as components, classes, or methods [70, 375]. To develop software, requirements formulated with application domain concepts have to be realized with implementation technologies, which requires bridging the gap from the problem to the solution space. France and Rumpe [164] argue that bridging this gap by manual implementation efforts causes accidental complexity (*cf.* [88]) which complicates the development of software.

Consequently, MDE aims at mitigating this gap as well as the complexity of software by employing models that describe the software at different levels of abstraction and from different views, and by automated techniques to systematically transform and analyze models and to finally generate the running system [164]. Models at different levels of abstraction provide vertical views of the system [384]. Examples are models describing the system's requirements, architecture, implementation, or deployment [164]. For instance, architectural models abstract from the complexities of the implementation and deployment platform such that developers can focus on the domain concerns instead of struggling with the application programming interfaces of the platform and its usage [375]. This aspect is exemplified by the Model-Driven Architecture (MDA) discussed in Section 2.1.2. Models that focus on specific concerns such as performance, fault tolerance, or security (*cf.* [71, 164, 243, 383]) provide horizontal views of a system [384]. Such models are used to construct and analyze the individuals concerns addressed by the models.

1 Other terms such as "model-based software engineering" [373, 381] or "model-driven software development" [384] are often used as synonyms for model-driven engineering in literature. Throughout this thesis, we will use the latter term and its acronym MDE.

Thus, MDE aims at raising the level of abstraction in software engineering[2], automating the development by transformational and generative techniques, and analyzing models early in the development process for quality assurance in order to tame the complexity of engineering software [70, 71, 164, 294, 375, 382]. Therefore, MDE provides techniques, methods, processes, and tools [164] and it must specify for a given software project the modeling languages, models, transformations, and overall development process to be adopted [243]. Eventually, MDE "industrializes" software development [198] with the goal of improving the developers' productivity and software quality, which should reduce the costs and time to market of software development [51, 382, 384].

This goal should be in particular achieved by the power of models. Selic [382, p. 20] argues that models are "easier to specify, understand, and maintain" since they use concepts closer to the problem domain than to the implementation domain in contrast to most programming languages. In this context, France and Rumpe [164, p. 15] argue "that the complexity of software will overwhelm our ability to effectively maintain mental models of a system" and that models should be made explicit such that automated tools operating on models can be applied. They further argue that by explicit models and automated operations, "we are relieving significant cognitive burden and reducing the accidental complexities associated with maintaining mentally held models" [164, p. 15]. Consequently, we will elaborate in the following the notions of *models* and *model operations* in MDE.

2.1.1 *Models and Model Operations*

As previously outlined, *models* are the primary artifacts and *model operations* play an important role in MDE. We discuss each of them in the following.

Stachowiak [400] defines a model by three characteristics:(1) *mapping:* a model is a representation of a natural or artificial original that can be a model itself. (2) *reduction:* a model does not capture all properties of the original but focuses on those that are relevant to the creator or user of the model. On the other hand, a model may have properties that the original does not have. (3) *pragmatism:* a model serves a certain purpose for a user, that is, it is used as a replacement of the original at a particular point in time and for a period of time to perform certain actual or mental operations. This notion of a model is similar or even equal to definitions proposed by several other researchers who are discussing the role of models in software engineering (*e. g.*, [164, 282, 294, 366]). That is, a model serves as a replacement of the original [71] if it is "simpler, safer, or cheaper" [366, p. 75] to use the model instead of the original or even if the original cannot be handled [282] for a specific purpose. Thereby, the original that is represented by a model may already exist or not, such that the model is said to be *descriptive* or *prescriptive*, respectively [71, 282, 379]. Therefore, models reduce risks by supporting engineers in better understanding a problem and its solutions before actually spending any efforts in implementing one of the solutions [382] or they generally support the development and evolution of a solution [375].

In this context, Selic [382] considers five key characteristics of an engineering model: (1) *abstraction:* a model abstracts from the system to be developed. (2) *understandability:* the notation used to describe the system should match the intuition or comprehension of the domain, for which the system is developed. (3) *accuracy:* the model must be a valid representation of the system. (4) *predictiveness:* the model should allow engineers to predict system properties by analyzing the model. (5) *inexpensiveness:* it should be cheaper to construct and analyze the model than directly the system. These characteristics influence the

2 Abstraction is considered as a critical and important factor in software engineering [257].

purpose of a model [294, 383] such as mitigating complexity (*cf.* abstraction), supporting the understanding of the system and communication among people (*cf.* understandability), predicting and analyzing system properties (*cf.* predictiveness), and guiding or generating the implementation. All these purposes rely on accurate and inexpensive models.

A particular aspect that supports the understandability is the execution of models. *Executable* models typically describe the behavior rather than the structure of the system and they can be animated or executed to simulate the system under development, which provides early and direct experience with the system [164, 381, 382]. In this context, simulation is also a means to validate a model and to improve the quality of the system under development [142, 320, 438].

As discussed previously, MDE employs various models at different levels of abstraction to obtain concern-specific views of the system. This typically calls for various formalisms, that is, *modeling languages* to express the models (*cf.* [70, 71, 243, 379]). According to Atkinson and Kühne [51], a modeling language comprises four aspects, an *abstract syntax*, a *concrete syntax*, *well-formedness*, and *semantics*.

In MDE, the abstract syntax of a language is specified by a *metamodel* that defines the concepts of the language and how these concepts can be combined to create a model [71, 243, 337, 383]. A model created with a modeling language is said to be an *instance of* the corresponding metamodel [51].

The concrete syntax of a language is the human readable notation to create and represent a model [383], or more specifically, the rendering of the concepts defined by the metamodel [51]. In general, the concrete syntax can be textual or graphical [426]. A textual syntax is typically specified by a grammar while there are no standard approaches despite [323] but only specific tools available to define a graphical syntax [337].

Well-formedness refers to rules on how to apply the concepts of a language [51], for instance, to assert or constrain the use of a language [70, 243, 383]. To specify such rules, constraint languages such as OCL can be used [51, 337]. Typically, a model is said to be *valid* if it is an instance of the metamodel defining the abstract syntax of the language used to create the model and if it is well-formed with respect to the rules.

Finally, the semantics describes the meaning of the language concepts and thus, supports the interpretation of models expressed in this language [51, 207, 379, 383]. The interpretation maps a model to the original being modeled, which gives the model a meaning [379]. In this context, Harel and Rumpe [207] consider a mapping from the concepts of the language to the semantic domain. They advocate an explicit specification of the language, semantic domain, and the mapping to clearly define the semantics. This can be done formally (*e. g.*, using mathematics) or informally (*e. g.*, using natural language) (*cf.* formal and informal designs of languages in [302]). For executable models, the semantics of the corresponding language particularly covers how such models are executed by a computer [383]. While the abstract syntax of a language is typically specified by a metamodel, there does not exist a corresponding formalism that is commonly used to specify the behavioral/execution semantics such that the formalization of the semantics is challenging [94, 383]. In this context, Bryant et al. [94, p. 248] state that "[t]ypically, the behavioral semantics [...] is described within individual hard-coded model interpreters" that either transform a model to an executable form such as code or that directly execute a model. Selic [383, p. 316] makes a similar observation and states that the execution semantics are "often [...] specified in the form of computer programs written in some programming language". Consequently, the semantics encoded in the implementation of an interpreter is implicit and can only be accessed by testing created models with the interpreter [42].

In MDE the semantics can be explicitly specified as operational, translational, and denotational semantics [42, 118]. Operational semantics defines the computations for language concepts that are performed when the individual concepts are executed. Translational semantics translates concepts of the language to concepts of another language that already has an accurate semantics. Denotational semantics uses a mathematical formalism to define the language concepts by providing mathematical objects as denotations for the concepts. Finally, Clark et al. [118] additionally consider extensional semantics for a language, which extends the semantics of an existing language.

A modeling language itself, particularly, its metamodel is defined in a modeling language that is called *meta-metamodel* [51, 71, 379]. This results in the typical modeling infrastructure of MDE with its four levels (*cf.* [51, 70, 71, 379]): The lowest level *M0* refers to the runtime instances of the software such as data objects. The models specifying or describing the software and created by the developer are located at level *M1*. The level *M2* contains the modeling languages by means of the metamodels used to create the M1 models. The highest level *M3* contains the meta-metamodels that are used to define the metamodels located at M2. Concepts of a certain level are instances of concepts of the next higher level while the M3 level is reflexive or in other terms self-defining, that is, the meta-metamodel is defined in the same language as it describes (*cf.* [70, 71, 379, 383]).

This infrastructure is exemplified by the Object Management Group (OMG) employing the Meta Object Facility (MOF) [328] as the meta-metamodel at M3 and the Unified Modeling Language (UML) [329] together with the OCL [327] at M2.[3] Then, models created by the user with UML such as class diagrams are located at M1 while M0 contains the runtime instances of the application.

The use of one common meta-metamodel in an MDE infrastructure supports generically navigating models and establishing relations between arbitrary models written in different languages that share a meta-metamodel. Thus, given a common meta-metamodel, all models as well as their metamodels can be treated and generically processed in a *unified* manner, which is called the "unification power of models" by Bézivin [71]. This is exploited for specifying *model operations*, that is, automated activities working on or using models such as transforming, synchronizing, or differencing models (*cf.* [70, 71]). Such operations may just work on and use models (so-called *state-based operations* such as transforming one model to another one), or they may use model changes that drive their processing (so-called *change-based operations* such as synchronizing changes from one model to another one).[4]

Model transformations are "automated processes that take one or more source models as input and produce one or more target models as output, while following a set of transformation rules" [384, p. 42f.]. Thus, model transformations address, among other things, the problem of keeping multiple models describing the software from different views and at different levels of abstraction consistent to each other [384]. In this context, a horizontal transformation automatically derives a view from another one at the same abstraction level while a vertical transformation either refines a model from higher to lower levels of abstraction or abstracts a model from lower to higher levels of abstraction [300, 384]. Thereby, transformations capture expert knowledge such as best practices that might get lost otherwise if the transformation is not explicitly defined but done manually [294, 303]. Hence, explicit transformations promise to improve the software quality in MDE [89]. If the source and target models are expressed in the same language, the trans-

3 Besides UML, the OMG defines and employs several other languages at M2 such as the Common Warehouse Metamodel (CWM) [322] or Software & Systems Process Engineering Meta-Model (SPEM) [330].

4 This is similar to state-based and change-based versioning for software configuration management [121].

formation is said to be endogenous, otherwise exogenous [300]. In MDE, transformation languages are used to express the rules defining a model transformation [384]. Examples of such languages are Triple Graph Grammars (TGGs) [220, 377], Query/View/Transformation (QVT) [326], and the ATLAS Transformation Language (ATL) [238]. Transformation rules define at the level of metamodels how target models are produced from source models using the common meta-metamodel to relate concepts of the source metamodel to the target metamodel. A transformation is *bidirectional* if it additionally supports producing source models from target models and thus, transformations in both directions (*cf.* [403, 404]). While a model transformation produces the target model completely anew with each run from the source model, *model synchronization* maintains both models and incrementally propagates the changes from the source model to the target model improving the transformation performance [188]. Similar to transformations, model synchronizations can be endogenous or exogenous and unidirectional or bidirectional. Finally, when explicitly specifying transformations with rules expressed in a certain language, such rules can be considered themselves as models and they can be themselves subject to transformations (*cf.* higher-order transformations) [71].

2.1.2 *Model-Driven Architecture (MDA)*

An example of an MDE approach is the Model-Driven Architecture (MDA) [324, 325] that is proposed by the Object Management Group (OMG) and considered as a subset of MDE [70, 243]. It aims at (semi-)automating the transformations from abstract models to executable systems while tackling the complexity of software technologies and platforms (*cf.* [89, 324, 325]). This should be achieved by modeling the software under development using three different views (*cf.* [89, 164, 243, 303, 324, 325]): (1) a business or domain model, formerly called Computation Independent Model (CIM), describes the domain and requirements of the software. (2) a Platform Independent Model (PIM) specifies the software's features and realization that are independent from the technologies and platforms used for implementation. (3) a Platform Specific Model (PSM) introduces the technical details relevant for implementing the PIM concepts using a specific technology and platform. Though each of these views may reflect different aspects of the software [89], MDA focuses on separating the business concerns and software designs from technology-dependent and platform-specific implementation concerns [243, 324, 325]. This should separate business/domain-oriented decisions from platform decisions [89], enable reuse of domain and expert knowledge [294], and ease the migration of a software from one to another platform [164].

To ease the development and migration of software, MDA relies on (semi-)automated model transformations and code generation (*cf.* [89, 243]). Model transformations among the different views are explicitly defined and automated if appropriate patterns are available, for instance, on how to refine a PIM to a PSM by automatically adding technical details. Eventually, code is generated from the PSM and developers complete the generated code manually. To achieve (partial) automation, the MDA is based on standards for modeling and transformations languages (*i. e.*, metamodels) [89] such as the Unified Modeling Language (UML) [329] and the Query/View/Transformation (QVT) language [326] that share a common meta-metamodel, called Meta Object Facility (MOF) [328].

Finally, the idea of separating domain concerns and implementation concerns is also relevant for reverse engineering when existing implementations are mined to obtain abstractions at the domain level, that is, obtaining a PIM from a PSM [243].

2.1.3 *Domain-Specific Modeling and Languages*

As argued by Schmidt [375, p. 25] "[m]odel-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively." The MDA as discussed in Section 2.1.2 addresses the problem of platform complexity. However, MDA is based on UML, which is a general-purpose modeling language, and therefore requires from engineers to translate domain concepts to UML concepts in order to express them. In contrast, to directly and effectively express domain concepts, problem-level abstractions should be leveraged in modeling languages [164] such that the syntax and semantics of the language match the domain [375]. Hence, software solutions should be specified with concepts from the problem domain and following the MDE idea, the final products should be generated automatically from such domain-level specifications [242].

This idea is followed in the *domain engineering* field [360], that is, amongst others "the activity of systematically modeling domains" [136, p. 28]. Kelly and Tolvanen [242, p. 3] define "a domain as an area of interest to a particular development effort". In general, they distinguish horizontal and vertical domains. The former refers to technical aspects such as persistence, communication, or transactions while the latter considers different businesses such as banking, robot control, or telecommunications. However, the scope of a domain in practice can be even more restricted focusing on a particular product or platform [242].

In the context of MDE, *domain-specific modeling* [242] refers to engineering approaches that use Domain-Specific Languages (DSLs) to develop software. While for France and Rumpe [164, p. 10] "[a] domain specific language consists of constructs that capture phenomena in the domain it describes", a more detailed view is provided by Deursen et al. [136, p. 26]: "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." This view is shared by Fowler [162] who in addition states that DSLs are used by humans and therefore, they should be fluent and easy to understand by humans but also executable by a computer. Though Deursen et al. [136] and Fowler [162] emphasize the execution by computers, DSLs in general do not necessarily have to be executable [302]. Furthermore, Fowler [162] characterizes the limited expressiveness of a DSL, because of the clear focus on a domain, as the main difference to a General Purpose Language (GPL).

Based on such a focus of a DSL by means of the language's notations and constructs targeting a specific domain, Mernik et al. [302, p. 317] argue that a DSL provides "substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs." In the same direction, Kelly and Tolvanen [242] argue that a full code generation that does not require manual adjustments of the generated code is rather feasible for domain-specific models than for general-purpose models. Hence, the use of DSLs promises improved productivity and quality due to fully automated generation steps that together with the language realize best practices for creating, analyzing, and transforming models to high-quality code and applications. A recent empirical study that investigated the use of MDE in the industry has backed this view stating that "companies who successfully applied MDE largely did so by creating or using languages specifically developed for their domain, rather than using general-purpose languages such as UML" [439, p. 80].

Besides improved development productivity as already discussed and also envisioned in [136, 162, 425], the use of DSLs promises further benefits for developing software. The

improved quality is leveraged by automated transformation or generation steps that realize best practices [242] or take over repetitive work usually performed by developers [425]. Moreover, using and focusing on domain concepts and abstracting from implementation details, a DSL removes unnecessary degrees of freedom for developers who can therefore exclusively concentrate on the domain problem [425]. In general, this should separate the essential from the accidental complexity (*cf.* [425]) such that developers can concentrate on the essence, that is, the domain problem. Specifying a software with a DSL enables the analysis (*e. g.*, the validation, verification, testing, or optimization) of the models and software at the level of domain concepts [136, 302, 425], which does not require any translation of the analyses or their results to or from the implementation domain. Additionally, DSL models capture domain knowledge at a higher level of abstraction which enables the reuse [136] and longevity of this knowledge [425]. For instance, the platform independence of DSL models supports their reuse to realize a solution for a different platform or to migrate an existing software to a new version of the platform. This enhances the portability and maintainability of the software [136, 162, 425]. Moreover, the DSL can be (re)used in different projects targeting the same domain since the language definition typically does not depend of any specific application [383]. Finally, by leveraging domain concepts in the syntax and semantics of a DSL, domain experts can be involved to understand, validate, or create DSL models, which eventually improves the communication between domain experts and software engineers [136, 162, 425].

However, to realize such benefits by using a DSL, several problems have to be addressed or at least considered. Developing and maintaining a DSL with its associated tools (*e. g.*, editors and code generators) causes costs [136, 162, 302, 425], particularly, if many different languages are used in a project [162, 425], that have to be confronted with the productivity and quality gains of using the DSL. For instance, developers using the DSL have to be trained in getting familiar with the language [136, 302], especially, if the language is only used within one organization [162], and developers building the DSL have to obtain language engineering skills [425]. In this context, a challenge is to identify the proper and right scope and abstraction of a DSL when developing and co-evolving the language with the domain [136, 162]. Finally, there might be performance issues with the generated code that is less efficient than hand-crafted software [136]. However, modern generators can remove inefficient domain abstractions during the generation process to improve efficiency [425].

Through code generation, a DSL is enacted by means of execution. Another option for executing DSL models, if they are executable, is interpretation [302, 425]. According to Voelter [425], generated code is easier to inspect and debug and often provides better performance than an interpreter while an interpreter-based approach enables dynamic changes of the model during execution and has shorter turnaround times since no generation, compilation, and packaging steps are required.

Besides the alternatives of code generation and interpretation, a DSL can be developed as an *external* or *internal* language (*cf.* [162]). An external DSL is a distinct language separated from the main language the application under development is implemented in. Hence, it might exhibit a different syntax and semantics than the main language. However, it requires the development of a corresponding code generator or interpreter that is typically specific for a DSL [136]. An internal DSL is based on an existing GPL and uses constructs of this GPL in a certain style providing "the feel of a custom language, rather than its host language" [162, p. 28]. Thereby, the DSL can partially use, specialize, restrict, or extend the GPL [302]. While an internal DSL avoids the development of a dedicated code generator or interpreter by reusing the host language's infrastructure, its syntax and semantics as well

as its analysis capabilities are a compromise between the domain-specific concepts and the GPL concepts [136, 302, 383]. Therefore, an internal DSL is feasible if its semantic concepts are similar to the ones of the host language [383]. An example for such a case is the UML profile mechanism that allows DSLs that "fall within the syntactic and semantic envelope defined by standard UML" [383, p. 308].

2.1.4 *Megamodels*

As discussed in the previous sections, MDE approaches employ a variety of models at different levels of abstraction and expressed in different (general-purpose and domain-specific) languages. Throughout the development process, models are created, changed, analyzed, or transformed manually by engineers or automatically by tools [164]. Such activities establish relationships between models, for instance, dependencies, transformations, refinements, or realizations. To manage the development process, all these models and their relationships have to be tracked [164], for instance, to capture how models as different views of a system are related to each other [368]. France and Rumpe [164, p. 14] argue that "[m]anually tracking [...] adds significant accidental complexity to the MDE development process". Hence, concepts and tools are needed for managing the models and their relationships, which is typically called *model management* (*cf.* [53, 164]).

One approach to tackle model management is megamodeling [72, 73]. In the literature, various perspectives on megamodels exists. For instance, Bézivin et al. [72, p. 30] consider a megamodel as "a model with other models as elements" including the relationships between the models [53]. Salay et al. [368, p. 142] use the term *macromodel* to describe a model that "consists of elements denoting models and links denoting intended relationships between these models with their internal details abstracted away". Finally, for Perovich et al. [346, p. 1] a megamodel is "a model composed of related models". All these various perspectives share a common understanding of a megamodel such that Hebig et al. [211, p. 7] propose a unified definition of a *megamodel* as "a model that contains models and relations between them". Hence, megamodels are about describing and manipulating models and the relationships between the models. Thereby, a relationship can be associated with a model operation such as a model transformation or change propagation (*i. e.*, model synchronization) to automatically derive a model from another one or to synchronize two models. Thus, a goal of a megamodel could be keeping the models consistent to each other (*cf.* [211]), which is a challenge in MDE [384].

Besides using megamodels as a core for model management approaches [72, 73], they have been proposed for addressing more specific problems such as model repositories [72], traceability between models [53], modeling and reasoning about MDE [149], transformation chains [166], or to represent software architectures [346]. All these approaches have in common that they employ megamodel concepts to describe multiple models and their relationships to obtain an overview of the employed models and model operations in an MDE approach as well as to manipulate both.

2.1.5 *Runtime Models*

Besides models for developing software as we have discussed so far, researchers have started to investigate the application of such models in runtime environments. These efforts are coined by the term *Models@run.time* [49, 62, 79] and they aim for extending the use of MDE principles to runtime environments and thus leveraging the benefits of MDE

for dynamically adapting software at runtime. The research field of models at runtime is still novel and so far not well established in MDE [312] although it is considered as a part of it. For instance, in France and Rumpe's "broad vision of MDE, models are not only the primary artifacts of development, they are also the primary means by which developers and other systems understand, interact with, configure and modify the runtime behavior of software" [164, p. 3]. Hence, they consider two types of models in MDE, development models and runtime models, although the line between these two types can be blurred [79]. This is particularly the case for a "controlled ongoing design" [79, p. 23], that is, for using runtime models to modify the initial design of the software and to enact these modifications in the running system.

The motivation of using models at runtime is to exploit MDE principles such as abstraction, automation, and analysis to handle the complexity of adapting or generally managing running software systems (*cf.* [79, 164]). Models should provide "abstractions of runtime phenomena" [164, p. 3] that enable analyzing and reasoning about the running system by developers or automatically by tools. Similar to MDE for developing software, models@run.time aim for abstractions that hide the complexities of platforms and that provide problem-oriented views of the system, in this case, the running system.

In this context, Blair et al. [79, p.23] define a runtime model as "a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective". This definition exemplifies different views ("structure, behavior, or goals") that runtime models may provide and that should be related to the problem space rather than to the solution space. Such a runtime models should be a "causally connected self-representation", a term that originates from the field of computational reflection [285, 390].

According to Maes [285], every computational system is concerned with a domain and the system operates on internal data representing this domain. A system is *causally connected* to its domain if changes in the domain are reflected in the system's internal data and if changes of the data effect the domain. Likewise, a *reflective* computational system is concerned with *itself* and thus it includes data representing itself. This motivates the use of the term *self-representation* for this data. The self-representation is causally connected to the system and thus, the system may use it for meta-computations[5] such as observing (inspection/introspection) and changing (adaptation/intercession) its own behavior, for example, for self-optimization or self-modification purposes [285]. Thus, a reflective system is concerned with the domain and with itself such that these two concerns are often split into two conceptual levels: an object or base level covering the domain concerns and the reflective or meta level covering the meta-computation concerns (*cf.* [78, 252, 285]).

In this context, Maes [285] discusses two architectures for reflection that have different impact on the causal connection. In *procedural reflection*, the implementation or the program of the system directly serves as the self-representation. Thus, there is no distinction between the system implementation and the self-representation, and *one* representation is used for computations about the domain and computations about the system. This architecture fulfills the causal connection by construction since it avoids an explicit self-representation of the system and thus any consistency issues between them. However, this implies that computations about the system are performed at the abstraction level of the system implementation. Moreover, procedural reflections allows an infinite tower of reflection [75,

5 In the field of computational reflection, the use of the term "meta" is motivated by the fact that the system performs computations about or on itself, and it should not be confused with the use in the context of models, metamodels, and meta-metamodels as discussed in Section 2.1.1.

362, 390, 391], that is, an infinite stacking of meta-circular interpreters such that base-level elements are interpreted by meta-level elements, these meta-level elements are interpreted by meta-meta-level elements, and so on. In contrast, *declarative reflection* considers an explicit self-representation of the system, which is different from the implementation of the system. This makes the realization of the causal connection challenging because the self-representation need not to be at the same abstraction level as the system implementation.

Thus, declarative reflection maintains an explicit representation of the system, which can be considered as a runtime model. Threads of research, particularly, the software architecture, model-driven engineering, and requirements engineering fields, that aim at raising the level of abstraction of runtime representations, consider explicit runtime models.

The idea of reflection, primarily procedural reflection, has been realized in programming languages [248, 285, 331, 362, 390] and middleware [74–78, 252]. The runtime models used as causally connected self-representations in these fields are inherently coupled to the computation model of the programming language or middleware. For instance, Nierstrasz et al. [319] use the abstract syntax tree of the program as the runtime model. Therefore, these models are at a low level of abstraction and refer to the system's solution space (*cf.* [79]). This is often the consequence of implementing the system and the self-representation in the same language [75]. In contrast and as defined by Blair et al. [79], models@run.time aim for causally connected self-representations at higher levels of abstractions that are rather related to the problem space than to the solution space. This aim shares similarities with declarative reflection that considers abstract statements about the system such as what the system behavior is instead of how the behavior is realized (*cf.* [75, 285]).

In this context, the software architecture has been considered as an appropriate abstraction for runtime models (*cf.* [79, 161, 172, 335]) that are expressed in Architecture Description Languages (ADLs) [293] and therefore, in different languages as used for the implementation of the system. Even more abstract runtime models than at the software architecture level are considered in the requirements engineering field that investigates the use of requirement models such as goal models at runtime [65].

For such abstract runtime models, a crucial aspect is to continuously guarantee the causal connection, that is, the synchronization between the running system and the runtime model if one or the other changes. This is a key challenge for runtime models, particularly, when considering the abstraction gap between a running system and a high-level runtime model (*cf.* [79]). A recent survey discussing the current status of runtime models in the research literature particularly highlights the focus of models@run.time on providing appropriate runtime abstractions for analyzing and adapting running software system [408].

2.2 SELF-ADAPTIVE SOFTWARE

This section discusses the permanent need to change software beyond typical software evolution approaches, which has led to the vision of self-adaptive software. Finally, this section outlines basic engineering principles for such software systems.

2.2.1 *Software Change*

In software engineering, awareness of the continuous need to adapt software has emerged over the past decades. Parnas [339] coined the term of *software aging* and identified two reasons why software ages. First, needs and expectations of users change and the software must be modified according to these changes. If such modifications are not done (*cf. lack*

Table 1: Lehman's Laws of Software Evolution [274].

No.	Name	Description
I	Continuing change	An E-type system must be continually adapted else it becomes progressively less satisfactory in use
II	Increasing complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it
III	Self-regulation	Global E-type system evolution processes are self-regulating
IV	Conservation of organizational stability	Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime
V	Conservation of familiarity	In general, the average incremental growth (growth rate trend) of E-type systems tends to decline
VI	Continuing growth	The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime
VII	Declining quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type system will appear to decline as it is evolved
VIII	Feedback system	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

of movement), the users are not satisfied with the software and change to an alternative product. Second, modifications made to the software, particularly due to urgent change requests, are often not aligned with the original design rationale of the software, which degrades the structure of the software and invalidates the rationale (*cf. ignorant surgery*). Hence, software either ages because it is not changed at all or not properly changed with respect to the (initial) design. Consequences of software aging are a reduced performance and reliability of the product and the inability to keep up with changing needs since modifying software typically by means of adding code makes the software bigger[6], which in turn makes further modifications more difficult [339]. Finally, Parnas [339] notes that we might design for change during development, however, only to some extent as we cannot predict *all* changes that will happen in the future. Therefore, software aging is inevitable.

Similar observations are made by Lehman et al. [270–275] who have investigated existing software systems to state and to some extent empirically demonstrate the *laws of software evolution* listed in Table 1.[7] These laws hold for so called *E-type* programs, that is, software that addresses real world problems and applications and therefore, must be continually evolved as the reality changes [274]. Such an evolution is inevitable since the reality or more specifically the problem domain addressed by the software are unbounded concerning their properties including user needs while the software is bounded by the subset

6 We may additionally observe that "[s]oftware tends to grow over time, whether or not a rational need for it exists" [224, p. 10], that is, even without the need to adjust software to meet changing user requirements.

7 According to Cook et al. [123, p. 6f.] the term "laws" as used by Lehman should be interpreted as in social science, that is, as "general principles that are believed to apply to some class of social situation" but that might have to be modified for a particular situation. Therefore, some laws could not have been validated more recently [217] as for instance for the open source development of the Linux kernel [191], or only partially as for software product lines [332, 333]. In contrast, Lehman mainly studied propriety and monolithic systems developed by one organization (*cf.* [190]).

of implemented properties. Hence, every software is an abstraction of the domain that is based on assumptions to leave out certain properties in the software. However, changes of the domain and user needs likely invalidate such assumptions over time causing the need to evolve the software. In this context, Lehman emphasizes the need to capture and manage assumptions [272, 274] and states the laws shown in Table 1.

The first law postulates the need to continuously adapt the software to remain useful. This law is similar to the observation of lack of movement by Parnas [339]. Likewise, the second law stating that the complexity of the software increases with every evolution step is similar to the ignorant surgery phenomenon [339] since changes typically add new features to the software and likely degrade the original design structure. The third law refers to all activities of the evolution process that is self-regulated concerning the distributions of created product and process artifacts in the evolution steps. The self-regulation is based on feedback, for instance, by the organization, the product, or the process itself. The fourth law indicates the observation that evolution is somehow stable and happens at a constant rate per release, and the fifth law states the observation that the incremental growth of the system rather declines with each evolution step mainly due to the increasing complexity of the system (*cf.* Law II). The sixth law points out that the software continually grows by means of functionality, that is, by covering domain properties such as features that have been omitted previously but are now needed to satisfy user needs. The seventh law is a consequence of the first, second, and sixth laws. Quality of the software, for instance, performance or reliability, declines when the software grows and when it is not (*cf.* lack of movement) or not properly (*cf.* ignorant surgery) adapted. Finally, the eighth law considers evolution processes as feedback systems involving activities in the domain, environment, organization, business etc. in which the processes are carried out. For instance, the use of a software affects the domain and environment that in turn may affect the software.

From a technical or engineering point of view, the Laws I and VI are specifically relevant as they emphasize the need to continuously adapt permanently growing software. Moreover, these two laws have been confirmed by recent studies in the field of free and open source software development (*cf.* [217]). The software engineering community has addressed the need for change throughout the software life cycle by *software evolution* [297–299] and incremental and iterative development [269]. The latter does not consider evolution as en explicit stage in the life cycle but each iteration and increment constitute an evolution step (*cf.* [190]). Such evolutionary approaches shift the "bulk of software development" from the initial development to the evolution phase [353, p. 133]. Thereby, the transition from initial development to the evolution happens when a first running version of the software is available [354] though there need not to be a strict boundary between development and evolution [81].

As noticed by Bennett and Rajlich [66], there is no standard definition of the term *software evolution*. In contrast, the term is typically used as a synonym for *maintenance* [66, 190]. Software maintenance is generically defined as "the totality of activities required to provide cost-effective support to a software system" [392, p. 4], which includes post-delivery activities such as modifying software and pre-delivery activities such as planning for modifications. Moreover, maintenance is categorized into corrective (*i. e.*, repairing the software to correct problems such as bugs), adaptive (*i. e.*, adapting the software to its changing environment), perfective (*i. e.*, improving the software with respect to functional and non-functional requirements), and preventive (*i. e.*, modifying the software to ease future modifications and prevent problems before they occur) [190, 392]. The first three categories originate from Swanson [405] and are quite accepted while the last category is controversially

discussed [108, 190]. The relevance of adapting software becomes apparent considering the high costs of maintenance [66] that are typically higher than the development costs for the case of custom systems [394], as well as the time and risks of evolving software [60].

In the context of maintenance, processes to adapt software include activities such as requesting a change, analyzing the impact of the change, implementing and verifying the change, and finally releasing the change (*cf.* [353, 394] while such releases or updates are enacted during scheduled downtimes of the software system (*cf.* "stop-and-go maintenance" [348, p. 2]). Such change activities are mostly performed manually by different groups of people such as support staff and developers, and they typically follow some higher-level management process determining, for instance, the release schedules [249].

However, such adaptation processes that are performed mostly manually, decoupled from the runtime environment, and during scheduled downtimes of systems do not meet the requirements of many modern software systems [1]. Systems we depend on every day [387], particularly, mission-critical systems have to be permanently available which is compromised by shutting down the systems to deploy changes. For instance, for companies this bears the risk of losing customers or missing business opportunities. Therefore, adaptation must happen in the runtime environment while the system is running and available. Self- and context-aware systems as observed in ubiquitous and pervasive computing such as internet of things [345, 372, 431] require timely adaptations of the self according to changes of the individual contexts. However, manual maintenance processes introduce delays until adaptations are enacted in the running systems and they do not scale with respect to adapting each instance of the system to its individual context. Thus, a maintenance process must be carried out individually for each instance of the system with its context rather than for all instances. Despite the trend to configurable systems easing the adaptation of a system's self to its context, the complexity of such systems makes their manual configuration by administrators difficult and expensive [225]. For instance, manually changing ultra-large-scale systems [321] is difficult and costly or even impossible due to their complexity and size avoiding a complete shut-down of such systems for maintenance. The infeasibility of traditional maintenance processes to dynamically adapt mission-critical, self- and context-aware, or ultra-large-scale software has led to the vision of *autonomic computing* and *self-adaptive software*. This vision builds upon the self- and context-awareness of a system to enable the system to automatically configure, optimize, heal, and protect itself according to its operational context [370].

2.2.2 *Vision*

To address the requirements of timely adapting mission-critical, context-aware, or ultra-large-scale software while it is running, innovative ways of adapting and therefore of developing, deploying, operating, and evolving such software are needed. This has led to the research trends of *autonomic computing* [168, 225, 246] and *self-adaptive software* [112, 266, 16]. The former considers "computing systems that can manage themselves given high-level objectives from administrators" [246, p. 41], hence taking an administration perspective of adapting systems. In contrast, the latter takes a software engineering perspective and addresses "systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals" [16, p. 1]. Considering the hardware, operating system, network, middleware, and application layers of software systems, autonomic computing rather addresses the lower layers including the middleware, and self-adaptive software focuses on the two upper layers [370].

Both research trends share the same vision of automating adaptation and shifting the responsibility for adaptation from developers or administrators to the software system itself [1, 348]. Instead of performing a traditional maintenance process if the system's goals (*e. g.*, user needs), its context (*e. g.*, the location of the system in the environment), or the system's self (*e. g.*, the functional and non-functional behavior) change, the system itself should cope with such changes by automatically adapting itself. Such a self-adaptation shifts software engineering activities from development time to runtime and thus blurs the strict, traditional boundary between development time and runtime [55]. This blurring requires novel approaches to combine maintenance/evolution and self-adaptation.

The software evolution community has started to investigate runtime adaptation, that is, changing a running system without pausing or shutting it down [95, 301], and the self-adaptive software community has started to investigate the maintenance of self-adaptive software [1, 167]. Hence, instead of replacing maintenance/evolution with self-adaptation, the current efforts rather lead to a co-existence of them. Thus, while the self-adaptive system is running, change activities can occur *on-line* by means of self-adaptation (*i. e.*, internally to the system) and *off-line* by means of maintenance (*i. e.*, externally to the system) [1]. Nevertheless, the promise of self-adaptation is to mitigate the growing costs, complexity, and diversity of maintaining, operating, and generally evolving software systems.

Besides the term self-adaptation, other designations such as self-managing, self-managed, self-governing, self-maintenance, autonomic, self-control (*cf.* [276, 370]) are used in literature to describe the general capability of systems to automatically adjust to changing conditions. Salehie and Tahvildari [370] decompose this general capability into major and primitive capabilities originating from [221, 225, 246]. The four major capabilities are:(1) *self-configuring*, that is, automatically installing, configuring and integrating parts of a system while the rest of the system adapts to the new parts. (2) *self-optimizing*, that is, automatically discovering and exploiting opportunities to improve performance and resource efficiency of the system. (3) *self-healing*, that is, automatically identifying, diagnosing, and repairing errors, faults, and failures in the system. (4) *self-protecting*, that is, automatically identifying and handling security breaches as well as defending the system against attacks. These four major capabilities are based on two primitive ones:(1) *self-awareness*, that is, the system is aware of itself (*e. g.*, its own current behavior and state). (2) *context-awareness*, that is, the system is aware of its operational environment and conditions. The major and general capabilities should be achieved by equipping a system with a feedback loop resulting in a closed-loop system that is able to adapt itself to changes at runtime [222, 370].

2.2.3 *Engineering Principles*

The central principle of engineering self-adaptive software is the *feedback loop*[8] as a mechanism for realizing and controlling self-adaptation [93]. Such feedback loops are often inspired or even based on control theory [5, 215, 251]. In contrast to open-loop systems that cannot dynamically change themselves after deployment, a feedback loop turns a system into a closed-loop system that is able to automatically adjust itself to changes of the self, context, or requirements (*cf.* [222, 370]).

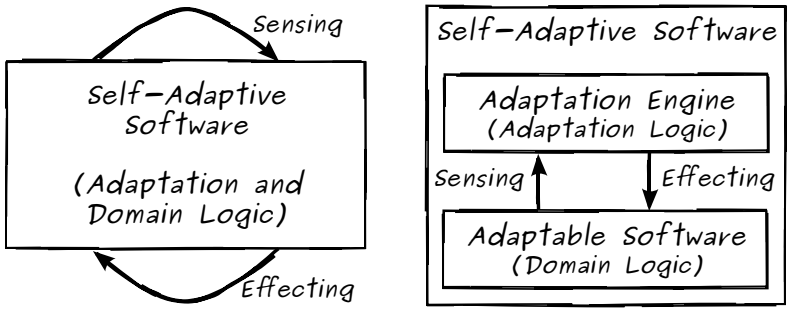
Based on a feedback loop for autonomic communications [139], Brun et al. [93] describe a generic feedback loop for self-adaptive software consisting of four activities. The first activity uses sensors to *collect* data from the context and system to obtain the current

8 Other terms such as "control loop" [139, 168, 221] or "adaptation loop" [370, 416] are also used as synonyms for a feedback loop in the literature while we will use the term feedback loop throughout this thesis.

contextual and system state. The next activity *analyzes* the collected data to identify needs for adaptation, for instance, when the system does not achieve its goals in its current context. Afterwards, the *decide* activity makes a decision on how to adapt the system and devises a plan for the adaptation. Finally, the *act* activity executes the devised plan. After one run of the feedback loop, the next run starts again with the collect activity.

This generic feedback loop can been seen as a refinement of the three-steps sense-plan-act [174] loop used for autonomous robots as well as the observe-decide-act [222] and learn-reason-act [10, 12] loops used in self-aware computing. Despite this refinement, the distinction between the analyze and decide activities can be blurred and therefore both activities are sometimes combined [27].

In general, it is software that realizes the self-adaptation by means of a feedback loop [93]. According to Salehie and Tahvildari [370], there are two general approaches for engineering self-adaptive software, that is, equipping a software application with a feedback loop to have a closed-loop system. As illustrated in Figure 1a showing the self-adaptive software sensing and effecting itself, the *internal* approach entangles the adaptation logic (*i. e.*, the feedback loop) and the domain logic of the application at the level of programming languages. Examples are the use of conditional expressions or exception handling that rather address local adaptations of the domain logic. Therefore, this approach often does not scale and the software becomes costly to test and maintain when intertwining adaptation and domain concerns [370].



(a) Internal Approach. (b) External Approach.

Figure 1: Internal and External Approaches to Self-Adaptive Software (*cf.* [370]).

In contrast, the *external* approach separates the adaptation and domain logic in the design by splitting the self-adaptive software into two parts: an *adaptation engine* implementing the adaptation logic and an *adaptable software* implementing the domain logic while the former controls (*i. e.*, senses and effects) the latter (*cf.* Figure 1b). Separating domain and adaptation concerns in a self-adaptive software system follows the idea of distinguishing computations about the domain and computations about the system itself in reflection [41], which has been discussed in Section 2.1.5. This separation is similar to the architectural reflection pattern described in [96] that distinguishes a base (*i. e.*, the adaptable software) and a meta level (*i. e.*, the adaptation engine). Finally, the separation allows engineers to use adaptation mechanisms that are technically independent from the application and its domain logic [370], such as policies (*cf.* [230]). This supports reusability of the adaptation engine or parts of it across different applications [370]. Salehie and Tahvildari [370] have discovered in their survey of research projects on self-adaptive software that all of them use the external approach. Therefore, we will focus in the following on the external approach.

A popular reference model for self-adaptive software that follows the external approach is *MAPE-K* [246]. Figure 2 depicts the model that considers an adaptation engine controlling the adaptable software.[9] Both parts are integrated by sensors and effectors to observe and respectively adjust the adaptable software. Sensors collect data about the software to be processed by the adaptation engine, and effectors are used by the engine to actually change the software. Both, the sensors and effectors are typically specific to the application realized by the adaptable software [230] and to the technical software platform.

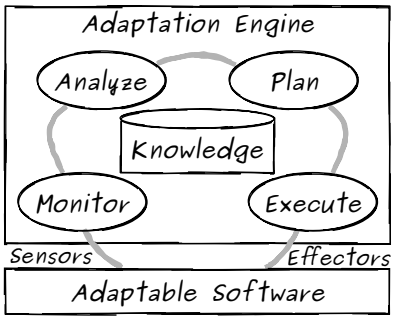


Figure 2: MAPE-K Reference Model for Self-Adaptive Software (*cf.* [246]).

In MAPE-K, the adaptation engine is a software component [230] that realizes a feedback loop consisting of the *Monitor*, *Analyze*, *Plan*, and *Execute* activities that share some *Knowledge*. These five elements account for the acronym MAPE-K. This feedback loop is similar to the generic feedback loop described by Brun et al. [93]. The monitor activity collects data about the adaptable software and the context, which is analyzed by the next activity to assess whether the software satisfies its goals. If adaptations are required to achieve the goals, they are finally planned and executed to the adaptable software. All activities share and operate on some *knowledge* such as an architectural runtime model (*cf.* Section 2.1.5) of the adaptable software [230], symptoms indicating problematic situations, change requests describing the need for adaptation, change plans specifying an adaptation, and policies [124]. For instance, based on its observations the monitor activity creates symptoms that are analyzed. If adaptations are required, the analyze activity sends a change request to the plan activity that devises a change plan to be finally executed [124, 370].

Policies are used to express and operationalize the goals of the system [230] and therefore the adaptation mechanism. Three types of policies are discussed by Kephart and Walsh [247]. First, action policies explicitly specify the adaptation behavior, that is, how the adaptable software should be adjusted by means of actions if it meets a given condition. A variant of such policies are Event-Condition-Action (ECA) rules (*cf.* [291]) that additionally define events whose occurrences trigger the application of the rules (*cf.* [230]). Second, goal policies specify the desired state, in which the adaptable software should be. If the software is in any other state, it is adapted to drive the software to the desired state. The policies do not define how the adaptation happens, that is, the path of actions from the current to the desired state. The adaptation engine has to plan this path based on the goal policies. Third, utility function policies define an objective function that quantifies for each currently feasible state of the adaptable software the desirability of the state. By adaptation, the software should be driven to the most desirable of all feasible states for optimization purposes.

9 The original terminology of MAPE-K has been adjusted from the autonomic computing [246] to the self-adaptive software field [370]. Originally, the self-adaptive software is named *autonomic element*, the adaptation engine is named *autonomic manager*, and the adaptable software is named *managed element*.

Policies, especially ECA rules, may operate in a stateless manner, that is, a feedback loop does not maintain any state information about the adaptable software. In contrast, a feedback loop may keep such state information by means of an architectural model of the adaptable software [230]. Such a model as part of the feedback loop's knowledge is a *runtime model* (*cf.* Section 2.1.5) of the software to which the symptoms, change requests, change plans, and policies are related rather than directly to the running software. This indirection through the model supports analyzing an adaptation at the model level before enacting it to the software, for instance, to avoid violations of application invariants by an adaptation [335]. To create such models, Architecture Description Languages (ADLs) such as *Darwin* [178], *xADL* [127], or *Acme* [170, 171] are often used. The use of architectural models is motivated by considering the software architecture as an appropriate abstraction level for adaptation (*cf.* [86, 169, 227, 258, 286, 334, 335]). Moreover, the architecture supports weak and strong adaptation [370] while the former refers to changing parameters of architectural elements and the latter to structurally changing the architecture [292].

The two-layer reference model distinguishing the adaptable software and the adaptation engine as proposed by the external approach or MAPE-K is further refined by Kramer and Magee [258]. They present a three-layer reference architecture for self-adaptive systems, which is depicted in Figure 3. This reference architecture is inspired by a general architecture for robots considering bottom up the three layers of reactive feedback control, reactive plan execution, and deliberative computations such as planning [174]. Kramer and Magee [258] aim for exploiting the benefits of such robot architectures by means of flexibility and responsiveness and for leveraging them in self-adaptive software systems.

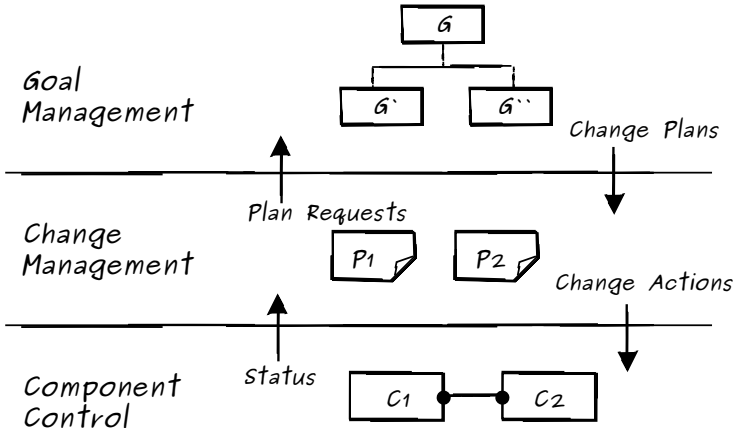


Figure 3: Three-Layer Reference Architecture for Self-Adaptive Software (*cf.* [258]).

In the reference architecture by Kramer and Magee [258], the *Component Control* layer is similar to the adaptable software in the external approach discussed previously. It refers to a component-based software system that implements the application's domain logic. Additionally, it reports its status to the higher level and supports modifying its structure (*i. e.*, creating, deleting, and (un)wiring components) based on change actions received from the higher level. The adaptation engine in the external approach is refined to the *Change Management* and *Goal Management* layers. The change management layer reacts to status reports from the underlying layer by selecting one of the precomputed plans to follow. Enacting a plan might involve adapting the component control layer, for instance, by adding components if additional behavior is needed to execute the selected plan. If the set of precomputed plans cannot cope with the state of the underlying layer, the change management layer re-

quests new plans from the higher layer. The goal management layer maintains the goals of the system and it computes plans to achieve these goals considering the current state of the system. Computing plans typically involves costly planning and is either triggered by requests from the underlying layer or by introducing new goals. In both cases, a new plan is provided to the change management layer, which enriches the set of precomputed plans.

According to Kramer and Magee [258], the different time scales of the various computations are the major criteria to distinguish the layers and to place the computations in different layers. The lowest layer addresses behavior with immediate responses while the highest layer considers time consuming behavior such as planning. Thus, the response time increases bottom-up in the architecture. Additional criteria are the degree of statefulness and the degree of strategic planning that both decrease top-down in the architecture, and generally separation of concerns [256].

The reference architecture by Kramer and Magee [258] is related to *adaptive control* that employs two feedback loops: a normal feedback loop with a controller and a plant and a feedback loop that adjusts the controller [50]. Thereby, the adaptation mechanism by means of the controller can be dynamically adjusted to flexibly react to runtime phenomena or changing goals. The same applies for the reference architecture by Kramer and Magee [258] considering a feedback loop with the change management and component control layers and another one with the goal management and change management layers. In each of these feedback loops, the higher layer adapts the lower layer. Another commonality to adaptive control is the time scales of the different loops, that is, the feedback loop adapting the controller or the change management layer is usually slower than the feedback loop controlling the plan or the component control layer (*cf.* [50, 258]).

A refinement of the three-layer reference architecture is *MORPH* [83, 84]. While MORPH adopts the key principles of the reference architecture, it explicitly distinguishes at the individual layers of the adaptation engine between mechanisms for adapting the configuration or the behavior of the adaptable software. In MORPH, these mechanisms are structured at the architectural level while considering their independent or coordinated operation.

The external approach and related models such as the three-layer reference architecture, MORPH, as well as robot or control architectures popularize *layered* architectures for designing self-adaptive software systems (*cf.* [138, 141, 177, 209, 410]). Besides having *one* feedback loop per layer, there are in general cases in which multiple feedback loops even within one layer are useful [87, 246, 435]. Examples of such cases are specialized feedback loops for individual self-adaptation capabilities such as self-configuration, self-optimization, self-healing, and self-protection (see Section 2.2.2) (*cf.* [22, 28, 30]), specialized feedback loops for individual concerns such as non-functional requirements addressed by one self-adaptation capability (*cf.* [165, 245]), and finally, multiple feedback loops to achieve decentralized self-adaptations, typically in distributed systems (*cf.* [428, 436]), for instance, to distinguish between local and global adaptation [117, 129, 200]. However, splitting up the adaptation into multiple feedback loops often requires coordination among these loops [428], for instance, to resolve conflicts between local and global adaptation.

Summing up, the fundamental principle for engineering self-adaptive software is the feedback loop, either single or multiple ones. A feedback loop typically consists of four activities, namely, collect/monitor, analyze, decide/plan, and act/execute that all share some knowledge such as symptoms, change requests, change plans, policies, or a runtime model of the adaptable software. In the design, such a feedback loop is often separated from the application to be adapted (*cf.* external approach) and sometimes split up across several layers to increase flexibility (*cf.* adaptive control).

3 REQUIREMENTS

In this chapter, we thoroughly discuss requirements for engineering self-adaptive software, particularly, feedback loops that are driven by runtime models. We derived these requirements from the literature. As the *external approach* is typically adopted in self-adaptive software [370], we only consider this approach as discussed in Section 2.2 and depicted in Figure 1b on Page 25. The external approach assumes a basic architecture that splits the *self-adaptive software* into the *adaptation engine* and the *adaptable software* while the former one controls (*sensing* and *effecting*) the latter one. The adaptable software realizes the *domain logic* and the engine implements the *adaptation logic* as a feedback loop sensing and effecting the adaptable software, which constitutes self-adaptation.

According to Salehie and Tahvildari [370], the external approach has the advantage of the reusability of the adaptation engine while the internal approach is often restricted to local adaptations and therefore does not scale and is difficult to test and maintain. Such arguments and tackling self-adaptation at the architectural level [86, 169, 227, 258, 286, 334, 335] motivate preferring the external over the internal approach.

However, when following the external approach, the engineering of an adaptation engine and feedback loops is essential. This requires a modeling language and techniques to design, specify, implement, execute, and evolve such an adaptation engine with its feedback loops. In the following, we discuss corresponding requirements for engineering self-adaptive software and particularly for modeling (*i. e.*, designing and specifying) and executing such software. Finally, we summarize and provide an overview of all requirements.

3.1 MODELING

The requirements (**R**) for modeling self-adaptive software cover the feedback loops, the runtime models used within feedback loops, sensors and effectors to connect feedback loops to the adaptable software, layered feedback loops to dynamically adapt feedback loops, and finally off-line adaptation to evolve feedback loops. These requirements are illustrated in Figure 4 on the next page and discussed in detail in the following.

R1 ***Feedback Loops***: Separating the adaptation engine from the adaptable software makes the *feedback loop* between them a crucial element of the architecture. Therefore, the feedback loop has to be made *explicit* in the design and analysis of self-adaptive software [93, 215, 251, 311, 386] (**R1.1** ***Explicit Feedback Loops***). Particularly, Shaw [386] argues that typical software engineering paradigms such as object orientation are not the best option to architect software systems with feedback loops, which calls for other means to design such systems. Such means should be inspired by process control to emphasize the feedback loops in the architectural design, which requires appropriate support by modeling languages.

A more detailed view of the feedback loop is provided by the *MAPE-K* reference model (Monitor/Analyze/Plan/Execute-Knowledge) [246] discussed in Section 2.2.3 and depicted in Figure 2 on Page 26. The feedback loop is refined to four adaptation activities that share some knowledge. The adaptable software is *monitored* and *analyzed*, and if changes are required, adaptation is *planned* and *executed* to this software. Such a feedback

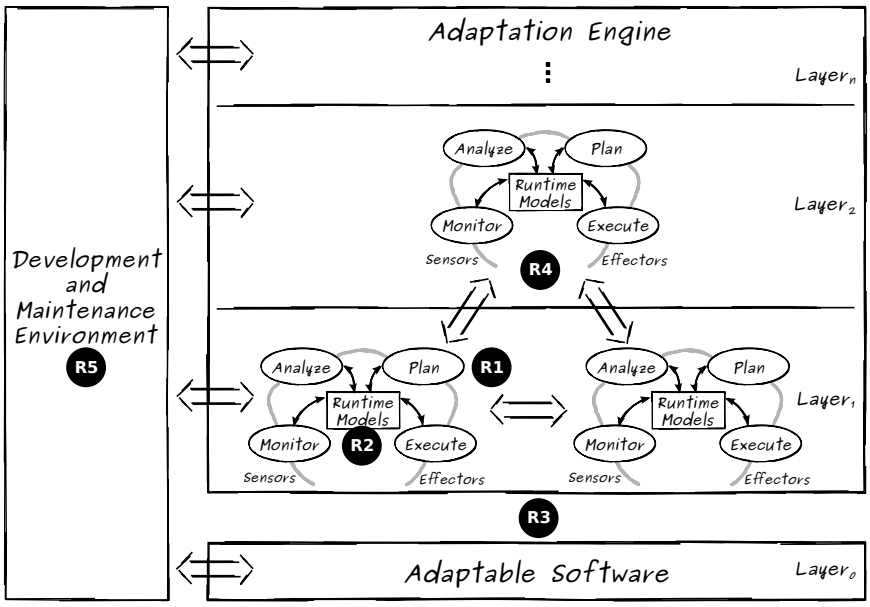


Figure 4: Overview of the Requirements for Modeling.

loop is shown in Figure 5 while the knowledge has been refined to a set of runtime models (*cf.* R2). Similar views of a feedback loop are discussed in literature (*e. g.*, [93, 139, 222]) that refine the loop into several activities such as collect, analyze, decide, and act [93]. Thus, when specifying a feedback loop, the individual adaptation activities and the knowledge (runtime models), that is, the elements of the feedback loop, have to be specified too. A feedback loop is modularly built up of such elements, which requires addressing the modular structure of a feedback loop (**R1.2** ***Modularity of Feedback Loops***).

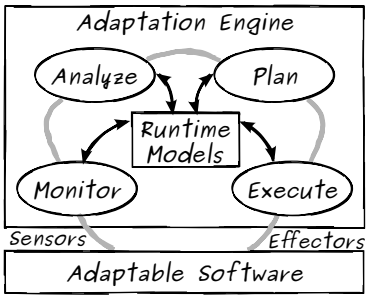


Figure 5: Feedback Loop and its Elements.

Addressing individual elements of a feedback loop, there might be design alternatives for composing these elements to a specific feedback loop. For instance, two alternative monitor activities are conceivable, one performing fine-grained and other more coarse-grained monitoring. Such alternatives should be made explicit as they reveal variability in the design of feedback loops and thus support establishing a design space for self-adaptive software [92] (**R1.3** ***Variability of Feedback Loops***).

Specifying individual elements of a feedback loop requires defining how these elements interact with each other, that is, how they are coordinated within a feedback loop [428]. On the one hand, coordination concerns how the adaptation activities form a feedback loop by means of their control flow. Besides the individual activities, the control flow determines the behavior of the feedback loop and makes the dependencies between individual activ-

ities explicit. On the other hand, the coordination should include how the activities of a feedback loop use the knowledge and thus, which activity processes which data. Thus, a modeling language for feedback loops should support the coordination of feedback loop elements, to which Vromant et al. [428] refer as intra-loop coordination, to enable a well-defined coordination and execution of a feedback loop (**R1.4** ***Intra-Loop Coordination***).

In addition to the control flow between adaptation activities, it has to be specified *when* the whole feedback loop should be executed. Similar to event- and time-triggered systems [253], a feedback loop might have to be executed if certain events occur (*e. g.*, notifying about changes of the adaptable software or environment), or periodically based on time. Moreover, a combination of both, events and time, is conceivable. Such triggers should be supported by a modeling language to specify conditions whose evaluation determine if and when a feedback loop should be executed (**R1.5** ***Triggers of Feedback Loops***).

Besides employing a single feedback loop in a self-adaptive software system, multiple feedback loops should be considered as well [87, 246, 435]. Decomposing self-adaptation to multiple feedback loops is motivated by separately handling different concerns [165, 245, 22, 28, 30], to distinguish between local and global adaptation [117, 129, 200], or to decentralize control in general [428, 436]. For instance, the concerns of failures and performance should be individually addressed by a self-healing feedback loop repairing the failures and a self-optimization feedback loop tuning the performance. An example for decentralizing control is to distinguish local adaptation of individual components and global adaptation of the architecture with separate feedback loops to take the locality into account.

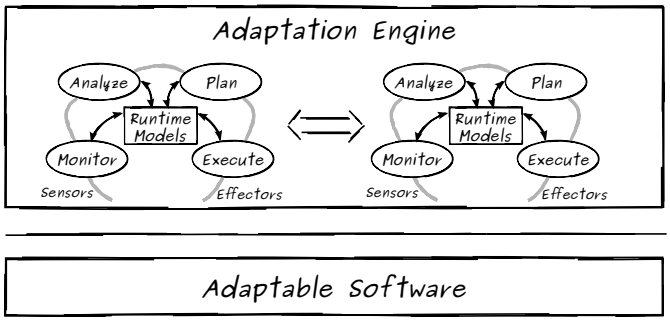


Figure 6: Multiple Feedback Loops and Inter-Loop Coordination.

However, employing multiple feedback loops requires coordination between them (*cf.* inter-loop coordination [428]), for instance, to avoid conflicting adaptations by different feedback loops, to handle interferences between feedback loops, or to enable collaboration. The language should therefore support the modeling of multiple, interacting feedback loops and their coordination as sketched in Figure 6 (**R1.6** ***Inter-Loop Coordination***).

Finally, the distribution of feedback loops should be supported, which is typically a prerequisite for decentralizing control. Hence, the modeling language should address how feedback loops are spread in the system (**R1.7** ***Distribution of Feedback Loops***).

R2 ***Runtime Models***: Considering a MAPE-K feedback loop, the adaptation activities are the computations performing self-adaptation and the knowledge base provides the information required for these computations. Weyns et al. [435, p.8:56] have observed that though "there is a shared understanding on the different types of computations in a MAPE-K [... feedback loop], the role of *knowledge* is less clear". This motivates refining the abstract notion of knowledge as well as explicitly modeling the refined knowledge.

As discussed in Section 2.2.3 in the context of foundations for self-adaptive software, the knowledge of a feedback loop can be a model, oftentimes an architectural runtime model [79], causally connected to the adaptable software as well as models used by or defining the adaptation activities such as symptoms, change requests, change plans, and policies (*cf.* [124, 230]). Depending on the used models, they may describe the whole state or rather changes of the state of the adaptable software. The former requires processing of the whole state, for instance, to identify adaptation needs while the latter allows the incremental processing of changes by feedback loops. This distinguishes the state-based from the event-based execution of feedback loops, which is discussed in detail for R6.2.

Based on the research field of runtime models and self-adaptive software, we have presented a categorization of runtime models as used by feedback loops [31, 32]. It shows that different kinds of models are simultaneously used at runtime, either to represent the adaptable software, context, and conducted analyses, or to specify and run the individual adaptation activities. The categorization is discussed in detail in Chapter 4. After our categorization, Weyns et al. [435] similarly proposed different kinds of models used by feedback loops, which are *subsystem models* representing the adaptable software, *concern models* representing the goals of the system by means of ECA rules, *environment models* representing the operational context of the system, and *MAPE working models* representing data shared by the adaptation activities.

These various categorizations show that multiple models that refine the abstract notion of the knowledge are simultaneously used by feedback loops at runtime. Therefore, the employed runtime models and their interplay and usage should be explicitly captured in the feedback loop design (**R2.1** ***Capturing Runtime Models***).

Weyns et al. [435] argue that models used by feedback loops such as those representing shared data of adaptation activities are oftentimes domain-specific. This calls for an *open* approach to support various domains by allowing engineers to employ any model expressed in any arbitrary language when engineering self-adaptive software systems. Hence, a modeling language for specifying feedback loops should not restrict the types, that is, the languages to express the runtime models that are used within the feedback loops (**R2.2** ***Openness for Languages of Runtime Models***).

Despite the openness for languages of runtime models, the modeling approach should support *abstract* runtime models that are causally connected to the adaptable software. The level of abstraction should particularly refer to the problem space rather than to the solution space of the adaptable software and adaptation concerns. Targeting the problem space is a major goal of runtime models [79] (**R2.3** ***Abstract Runtime Models***).

R3 ***Sensors and Effectors***: As shown by the MAPE-K reference model (*cf.* Figure 2 on Page 26), the adaptation engine and the adaptable software are connected by sensors and effectors. The engine uses sensors to observe the software and effectors to enact adaptations to the software. According to the observed sensor data, the monitor activity updates the runtime models representing the adaptable software (*cf.* R2) which are then used by the analyze and plan activities to prescribe an adaptation in the model. According to the planned adaptation, the execute activity changes the adaptable software through effectors. Thus, the monitor and execute activities use the sensors and effectors to maintain the causal connection between the runtime models and the adaptable software. Therefore, the modeling language should cover when the monitor and execute activities use the sensors and effectors to realize the causal connection and thus, how the feedback loops and the adaptable software are connected (**R3.1** ***Connecting Feedback Loops and the Adaptable Software***).

Addressing self-adaptation at the level of software architecture, that is considered as an appropriate abstraction level [161, 169, 286, 334, 335], sensors and effectors should support *parameter and structural adaptation* [292], which in turn should be supported by the feedback loops. A feedback loop may observe parameters or the structure of the adaptable software through sensors. Likewise, the adaptation through effectors may change parameters or the structure of the adaptable software. Parameter adaptation adjusts existent variables of the software while structural adaptation replaces (architectural) elements of the software and therefore can introduce new elements to the software that were not anticipated during development [292]. To benefit from this flexibility and handle architectural changes, structural adaptation should be particularly supported (**R3.2** ***Parameter and Structural Adaptation***). Likewise, Salehie and Tahvildari [370] distinguish between weak and strong adaptation while the former refers to parameter adaptation and the latter to structural adaptation.

However, since sensors and effectors usually depend on the specific adaptable software [230], we may assume that they are provided by the adaptable software and that most of their details can be hidden in the implementation of the monitor and execute activities. This assumption is also made by other researchers [113, 170, 307, 395] and it is motivated by programming language and middleware platforms that have recognized the need for runtime management [38, 234]. Such platforms either support the development of sensors and effectors or they already provide them through Application Programming Interfaces (APIs). Examples are the *Java Management Extensions (JMX)*[1] for *Java* or the *GlassFish*[2] and *OSGi*[3] platforms. Additionally, work as been done to enrich such standard management APIs for adaptation [3, 91]. Thus, we assume that software realized for such platforms provide sensors and effectors, which make the software observable and adaptable, and that the details of using them can be hidden in the implementation of the monitor and execute activities.

Despite this assumption, sensor details should be made visible for triggering conditions of feedback loops (*cf.* R1.5), that is, the language should support referencing sensor data, for instance, to characterize sensor events that should initiate the execution of a feedback loop. Thus, sensor events can be the basis for event-triggered feedback loops.

Aiming for flexible solutions for adaptation engines (*cf.* R4 and R5 discussed below), the monitor and execute activities of feedback loops and therefore, the sensors and effectors as well might have to be adapted. This has been investigated especially for adaptive monitoring [143, 356, 422, 424]. This requires that the modeling language should support cases where the adaptable software is dynamically instrumented by adding, removing, or changing sensors and effectors (**R3.3** ***Dynamic Sensors and Effectors***).

R4 ***Layered Architectures***: To achieve flexible solutions for self-adaptation, feedback loops should be as well open for change. Thus, feedback loops should support open and dynamic adaptation activities [370], that is, it should be possible to extend feedback loops at runtime with new adaptation actions such as policies. Such an open adaptation is typically realized by layered architectures, in which feedback loops operate on top of each other as illustrated in Figure 7 on the next page. The bottommost layer contains the adaptable software while the adaptation engine with its feedback loops is split it up in an arbitrary number of layers on top. Thus, feedback loops at a certain layer can dynamically adapt the feedback loops at the layer directly below. Such a layered architecture is needed for realizing adaptive [50, 251, 343] and hierarchical [154, 218, 246] control schemes or the reference

1 Java Management Extensions (JMX) Specification, v1.4, http://www.jcp.org/en/jsr/detail?id=3.
2 Application Server for the Java Enterprise Edition (Java EE), http://glassfish.java.net/.
3 OSGi Alliance: The Dynamic Module System for Java: https://www.osgi.org/.

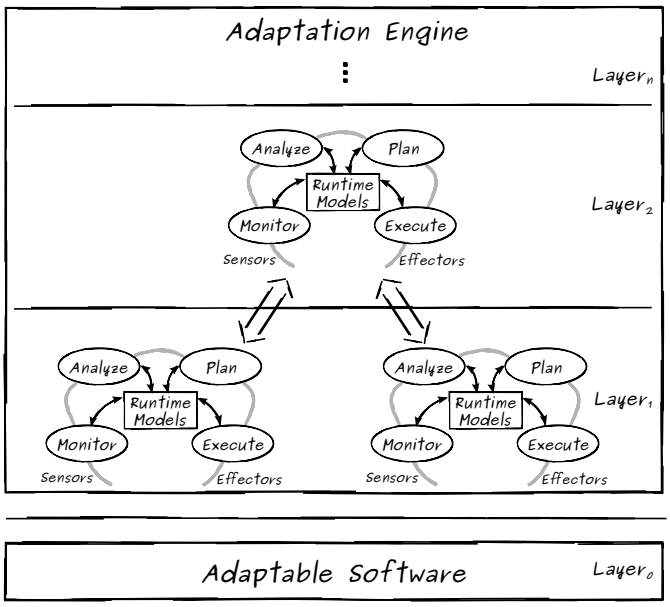


Figure 7: Layered Architecture for Self-Adaptive Software.

architecture for self-managed software systems proposed by Kramer and Magee [258] and discussed in Section 2.2.3. Though such architectures often consists of three layers [251, 258, 343], it is conceivable to have more than three and an arbitrary number of layers [154].

In layered architectures, feedback loops are themselves subject to adaptation. Therefore, they have to provide sensors and effectors that enable their adaptation by other feedback loops. This requires modeling support for adaptable feedback loops for $Layers_{1..n}$ and adapting feedback loops for $Layers_{2..n}$ (**R4.1** ***Adaptable and Adapting Feedback Loops***).

If parameters of the lower-layer feedback loop should be adapted, it would be sufficient that the higher-layer loop can observe and change some variables of the lower-layer loop. However, for structural adaptation, the higher-layer feedback loop has to operate on a structural representation of the lower-layer loop. Thus, some form of reflection of the lower-layer feedback loop has to be provided at runtime, which enables the adaptation of this feedback loop by higher-layer loops [41]. In this context, *declarative* and *procedural reflection* [285], as discussed in Section 2.1.5, can be supported. Hence, a modeling language for specifying feedback loops should support creating and maintaining reflective views of feedback loops that enable an adaptation of these loops (**R4.2** ***Procedural and Declarative Reflection on Feedback Loops***). Such reflective views are similar to runtime models representing the adaptable software and used by $Layer_1$ feedback loops to perform self-adaptation.

R5 ***Off-line Adaptation***: The promise of self-adaptive software is that the software is able to adjust itself and thus, that it automates and takes over some of the adaptation activities that are otherwise performed off-line in the context of maintenance and evolution. However, we cannot expect that self-adaptive software is able to cope with *all* needs for evolution itself and thus, to fully automate and take over *all* kinds of off-line adaptation activities. This calls for supporting the traditional maintenance of self-adaptive software and its feedback loops to address the long-term evolution of the software. Thus, besides an adaptation engine realizing (on-line) self-adaptation, the engine's co-existence with off-line adaptation such as typical maintenance is required [1, 54, 135, 167, 208, 26, 433] (**R5.1** ***Maintenance of Self-Adaptive Software***).

Similar to Andersson et al. [1], we consider an adaptation activity to be *off-line* if it is performed externally to the running self-adaptive software as it is typically done today in development and maintenance environments. In contrast, if an adaptation activity is performed internally to the self-adaptive software as part of a feedback loop, we refer to *on-line* adaptation. This is depicted in Figure 8 illustrating interactions between off-line activities in a development and maintenance environment and on-line activities for self-adaptation. Moreover, this figure illustrates that off-line adaptations may target the adaptable software as well as the feedback loops operating in the various layers on top of the software.

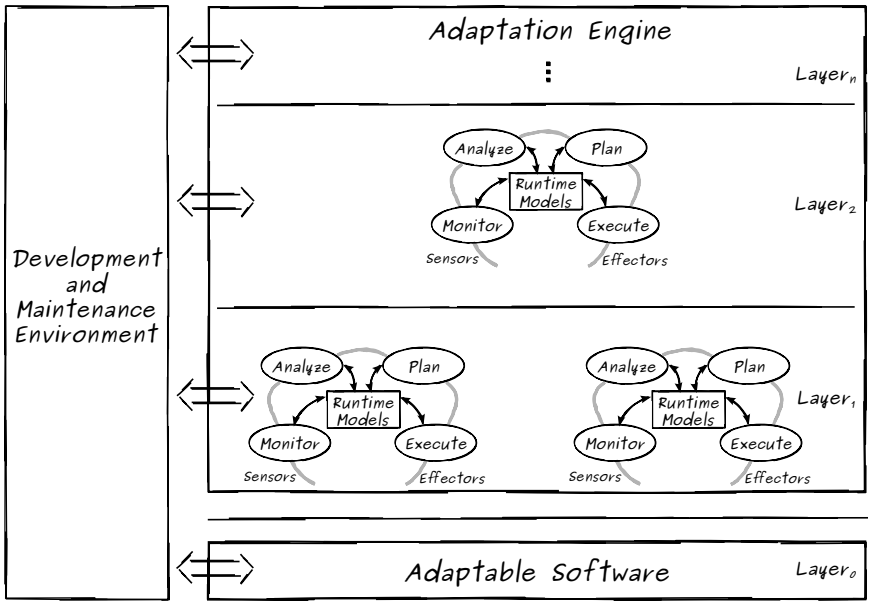


Figure 8: Off-line Adaptation of Self-Adaptive Software.

Of particular interest is the execution of an adaptation that has been analyzed and planned off-line to the running self-adaptive software. Thus, the execute activity is performed on-line while at the same time feedback loops might be operating for self-adaptation. Therefore, the modeling language and its runtime environment should support the co-existence of self-adaptation and maintenance by coordinating the respective on-line and off-line activities for self-adaptation and maintenance (**R5.2 *Coordination of Self-Adaptation and Maintenance***). For instance, the adaptation engine should allow engineers in development and maintenance environments to monitor the feedback loops and to integrate adaptations that have been analyzed and planned off-line by engineers to the running self-adaptive software for on-line execution while the feedback loops are running.

3.2 EXECUTION

Having modeled a self-adaptive software system using a language that fulfills the requirements discussed in the previous section (*cf.* R1-R5), the corresponding specification (models) should be brought to execution.

R6 *Execution*: Thus, the language for specifying self-adaptive software should provide runtime support to execute the feedback loops. The execution can be directly based on the models created with the language or translation steps from these models to an executable

format can be introduced. Such support should ease the transition from specifications to the execution of feedback loops. The execution should support all requirements R1-R5 that are addressed by the modeling language, that is, each of these requirements has an execution dimension such that all the features of the languages can be brought to execution. In following, we discuss further requirements that are only focusing on execution aspects.

Employing multiple feedback loops in a self-adaptive system, concurrency should be supported to simultaneously execute multiple feedback loops if needed or rather accordingly specified by engineers. On the one hand, concurrency decouples separate feedback loops with respect to execution, that is, the execution of a certain feedback loop is not blocked by another loop that is currently running. On the other hand, the runtime performance of executing feedback loops can be improved. Additionally, concurrency can be introduced at the level of adaptation activities, that is, individual activities of one feedback loops may run concurrently (**R6.1** ***Concurrent Execution of Feedback Loops***).

Concerning how a feedback loop operates, that is, how its adaptation activities are triggered and share knowledge, we may distinguish two variants. First, in *state-based* feedback loops the individual adaptation activities operate on a runtime model that represents the state of the adaptable software to perform self-adaptation. Therefore, the activities have to process the whole state, for instance, to analyze the adaptable software. Second, adaptation activities of *event-based* feedback loops only use events to exchange and process knowledge about the adaptable software. Events contain information about changes of the state rather than the complete state of the adaptable software. Nevertheless, events must contain fractions of the state by means of the context of the change such that the activities can process the events in a meaningful way. Consequently, event-based activities process changes of the state rather than the (complete) state. This is a prerequisite for the incremental processing of feedback loops while incrementality is considered as a key need for self-adaptation [180]. Thus, the execution concept of feedback loops should ideally address both variants (**R6.2** ***State- and Event-Based Execution***).

Finally, Salehie and Tahvildari [370, p. 14:3] point out that "an adaptation mechanism is expected to trace software changes and take appropriate actions at a reasonable cost and in a timely manner". Thus, the runtime efficiency of executing feedback loops is critical (**R6.3** ***Runtime-Efficient Execution***). On the one hand, this concerns timely adaptations in response to occurring changes as the feedback loops operate on-line. On the other hand, executing feedback loops should have a low overhead and thus, should not significantly disturb the performance of the adaptable software.

3.3 SUMMARY

In this section, we have discussed requirements for modeling and executing feedback loops in self-adaptive software, which we have derived from the research literature in this field as well as related fields such as control. All these requirements are structured and listed in Table 2 on the following page. We will use them in this thesis to discuss our approach to engineering self-adaptive software and to contrast our approach with related work.

Table 2: Requirements for Engineering Self-Adaptive Software.

No.	Name	Sub-requirements
R1	Feedback Loops	
		R1.1 Explicit Feedback Loops
		R1.2 Modularity of Feedback Loops (modular feedback loops elements)
		R1.3 Variability of Feedback Loops (variations of feedback loops elements)
		R1.4 Intra-Loop Coordination (interaction between elements within a feedback loop)
		R1.5 Triggers of Feedback Loops (when to execute a feedback loop)
		R1.6 Inter-Loop Coordination (interaction between feedback loops)
		R1.7 Distribution of Feedback Loops (distributing feedback loops)
R2	Runtime Models	
		R2.1 Capturing Runtime Models
		R2.2 Openness for Languages of Runtime Models
		R2.3 Abstract Runtime Models
R3	Sensors and Effectors	
		R3.1 Connecting Feedback Loops and the Adaptable Software
		R3.2 Parameter and Structural Adaptation
		R3.3 Dynamic Sensors and Effectors
R4	Layered Architectures	
		R4.1 Adaptable and Adapting Feedback Loops
		R4.2 Procedural and Declarative Reflection on Feedback Loops
R5	Off-line Adaptation	
		R5.1 Maintenance of Self-Adaptive Software
		R5.2 Coordination of Self-Adaptation and Maintenance
R6	Execution	
		R6.1 Concurrent Execution of Feedback Loops
		R6.2 State- and Event-Based Execution
		R6.3 Runtime-Efficient Execution (incl. low disturbance of the adaptable software's performance)

Part II

APPROACH

In this part of the thesis, we discuss our approach to engineering self-adaptive software, particularly, feedback loops with runtime models. Our approach is a domain-specific modeling solution, called Executable Runtime Megamodels (EUREMA). After giving an overview of EUREMA, we discuss how the EUREMA language is used to model self-adaptive software and how the resulting models are used to execute the self-adaptive software. Finally, we discuss the adaptation activities that constitute a feedback loop and their solution space in the context of EUREMA.

4 EXECUTABLE RUNTIME MEGAMODELS (EUREMA)

In this chapter, we provide an overview of our approach to engineering self-adaptive software, called *ExecUtable RuntimE MegAmodels* (EUREMA[1]). EUREMA rigorously follows MDE principles by consistently using models for specifying *and* executing self-adaptive software. In particularly, it is focused on specifying and executing adaptation engines with multiple feedback loops that employ runtime models. Therefore, EUREMA provides a domain-specific modeling language to support the specification and a runtime interpreter to support the execution of feedback loops.

The fundamental assumption of EUREMA is that the feedback loops use runtime models as introduced in Section 2.1.5. While the state of the art primarily just considers causally connected representations of the adaptable software as runtime models, we further treat executable specifications of the feedback loop's adaptation activities explicitly as runtime models. Such a holistic perspective on runtime models motivates the need for a runtime megamodel as a means to handle *all* runtime models in a unified manner. The idea of a runtime megamodel is the underlying principle of EUREMA. Therefore, we first discuss the role of runtime models and runtime megamodels in self-adaptive software. Then we give an overview of EUREMA and introduce the running example used throughout the remainder of this thesis.

4.1 RUNTIME MODELS

As mentioned previously, in EUREMA we consider feedback loops that use runtime models similar to the state of the art in engineering self-adaptive software and even beyond it by considering a more holistic view on runtime models.

On the one hand, the state of the art uses runtime models that represent and are causally connected to the adaptable software. However, such models are oftentimes related to the software's solution space and therefore at a rather low level of abstraction. Despite the abstraction level, such runtime models correspond to the definition of a runtime model by Blair et al. [79] (*cf.* Section 2.1.5). On the other hand, techniques such as Event-Condition-Action (ECA) rules or in general policies are used to operationalize the goals of the system and to define the adaptation activities of a feedback loop on top of such runtime models [230] (*cf.* Section 2.2.3). More specifically, existing approaches realize the adaptation activities by employing reusable infrastructures extended with application-specific plug-ins [170] or gluing together a series of tools [98] that provide such techniques. Thus, techniques realizing the adaptation activities operate on runtime models that represent the adaptable software. Besides such runtime models, other kinds of runtime models are not explicitly considered in practice as well as theoretically by Blair et al. [79].

In contrast, with EUREMA we take a more holistic perspective on *runtime models* and consider *all* models that are used at runtime by any adaptation activity of a feedback loop. For instance, we also treat ECA rules and policies as runtime models that are unified in a common MDE infrastructure with runtime models representing the adaptable software.

1 "Eurema is a widespread genus of grass yellow butterflies in the family Pieridae." See https://en.wikipedia.org/wiki/Eurema.

In this context, unification refers to a common meta-metamodel (*cf.* Section 2.1.1) for all runtime models used by feedback loops. This allows us to treat every (kind of) runtime model uniformly in EUREMA. Consequently, in EUREMA adaptation activities of feedback loops such as monitor, analyze, plan, and execute are specified by models that are kept alive and explicit at runtime to run these activities. Moreover, these activities work on runtime models that represent the adaptable software such that we can consider them as model operations (*cf.* Section 2.1.1).

In this context, we conducted an informal literature review [31, 32] and updated it for this thesis to investigate the role of runtime models, which resulted in a categorization of runtime models for feedback loops. This categorization is based on the purpose of runtime models in a feedback loop as depicted in Figure 9.

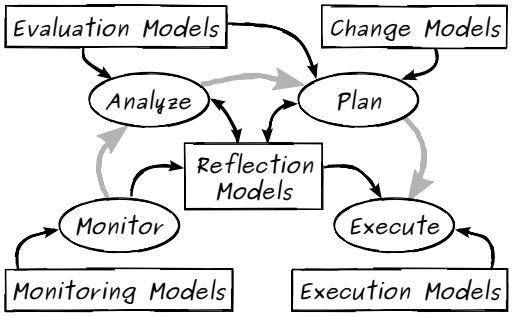


Figure 9: Runtime Models for Feedback Loops [25].

Reflection Models reflect the adaptable software and its environment. The monitor activity observes the adaptable software and environment and updates accordingly the reflection models. Thereby, *Monitoring Models* map system-level observations to the abstraction level of the reflection models. The reflection models are then analyzed to identify adaptation needs by applying *Evaluation Models* that, for instance, define constraints on reflection models. Thus, evaluation models specify the reasoning about the current state of the adaptable software. If adaptation needs have been identified, the analyze activity may annotate the problematic elements of the reflection models. Then, using these annotations, the planning activity devises a plan prescribing the adaptation on the reflection models. Planning is specified by *Change Models* describing the adaptable software's variability space. Evaluation models such as utility preferences may guide the exploration of this space to find an appropriate adaptation, for instance, by evaluating alternative options for adaptation based on their utilities. Finally, the execute activity enacts the planned adaptation on the adaptable software based on *Execution Models* that refine model-level adaptation to system-level adaptation. Throughout the feedback loop, all adaptation activities may exchange information using the reflection models, that is, by updating, annotating, or changing them.

These kinds of runtime models are categorized in the following by either combining or refining them as depicted in Figure 10.[2]

2 After our categorization, Weyns et al. [435] have proposed a similar categorization considering *subsystem models* representing the adaptable software, *environment models* representing the operational context of the adaptable software, *concern models* representing the goals of the system by means of ECA rules, and *MAPE working models* representing data shared by the adaptation activities. The first three models respectively map to system models, environment models, and adaptation models of our categorization while the role of MAPE working models is fulfilled by reflection models that can be enriched with information shared by the MAPE activities such as analysis results created by the analyze activity. In contrast to our categorization, monitoring and execution models (causal connection models) are not considered by Weyns et al.

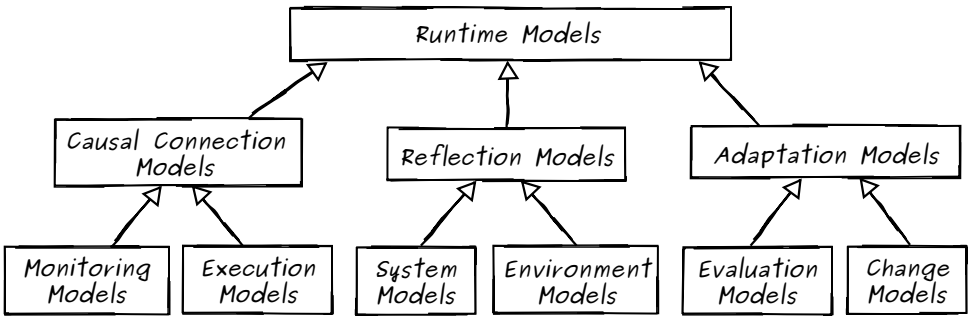


Figure 10: A Categorization of Runtime Models (*cf.* [31, 32]).

Monitoring and execution models are concerned with the synchronization of the adaptable software and the reflection models. This is known as the causal connection in reflective systems (*cf.* Section 2.1.5) playing a major role for self-adaptive software [41, 413], such that we consider them as *Causal Connection Models*. For instance, in our work [22, 28] we used Triple Graph Grammar (TGG) rules, a formalism for incrementally synchronizing changes between models (*cf.* Section 2.1.1), to propagate changes from low-level to high-level reflection models for monitoring and vice versa for execution such that the analyze and planning activities may operate on abstract, problem-oriented reflection models. Other approaches fairly neglect causal connection models. They either do not abstract from the adaptable software and provide reflection models at the same level of abstraction as the software (*cf.* [395]), which makes the required monitoring and execution models rather trivial. Or, if abstractions are provided, they consider dedicated architectural components such as the translation infrastructure and gauges in *Rainbow* [170, 173] that maintain mappings between elements of the adaptable software and elements of the reflection models. However, such mappings are not made explicit as causal connection and runtime models.

Reflection models are *System Models* representing the adaptable software or *Environment Models* representing the operational context of the self-adaptive software. They are descriptive models if they represent the current and actual state, or prescriptive if they represent a planned or predicted state of the software or environment. An example for a prescriptive system model is a reference model describing the target configuration of the adaptable software and the adaptation engine aims for achieving this configuration.

A system model can be specific to the implementation and computation model of the adaptable software as it is used in computational reflection [285, 390]. Such models directly refer to the concepts of the programming language or platform used for implementing the software [125, 239, 265]. Therefore, they are at a low level of abstraction and related to the solution space of the software. Examples of such implementation models are class and object diagrams for structural aspects and sequence diagrams and state machines for behavioral aspects of the adaptable software [52, 189, 203, 223, 239, 288, 361, 378]. System models at higher levels of abstraction but still causally connected to the adaptable software describe the configuration and architecture of the software. Several researchers argue that the software architecture is the appropriate abstraction level to deal with self-adaptation [86, 169, 227, 258, 286, 334, 335] such that many approaches employ architectural runtime models [101, 161, 170, 172, 196, 214, 306, 318, 22, 28, 444]. Behavioral aspects of the adaptable software can be represented at a higher level of abstraction by state machines or in general automata [182, 192] or by process or workflow models that provide business-oriented views [236, 371]. Runtime models at higher levels of abstraction are rather related to problem spaces as they abstract from the implementation and underlying technologies of the

adaptable software. This corresponds to the view of Blair et al. [79] on runtime models. With respect to self-adaptation, system models enable the system's self-awareness (*cf.* Section 2.2.2) at an appropriate level of abstraction and they are the basis for feedback loops (*i. e.*, for monitoring and analyzing the system and for planning and executing adaptations to the system).

An environment model describes the operational context of the self-adaptive software. To represent a context, a variety of models can be used: semi-structured tags and attributes, object-oriented or logic-based models [376], some form of variables such as key value pairs [306, 376], or even feature models [36]. With respect to self-adaptation, a context model enables context-awareness of the system (*cf.* Section 2.2.2) by monitoring the context and reflecting changes of the context in the model. The context model is then used together with the system model to analyze the current situation and to plan an adaptation if needed.

System and environment models may directly be analyzable and thus be used by the analysis and planning activities of feedback loops for reasoning. Examples are architectural runtime models that are annotated or enhanced with elements or attributes to address non-functional properties such as performance or failures to achieve self-optimization or self-healing [127, 170, 28]. On the other hand, dedicated analytical models combining relevant aspects of the system and environment can be used for reasoning. Examples of such models are Discrete Time Markov Chains (DTMCs) to analyze the reliability [98, 146, 150], or Continuous Time Markov Chains (CTMCs) [98], queuing networks [101], and stochastic colored Petri nets [318] to analyze the performance of a system. Such models are usually created manually and dynamically updated with runtime system information (*cf.* [146]) while they can also be synthesized from system and environment models obtained by runtime measurements [99]. Hence, a reflection model can be directly used for analysis or it can be transformed to analytical models to support specific properties and reasoning techniques. We consider such analytical models as analyzable reflection models.

How reflection models are analyzed and used for planning an adaptation is determined by *Adaptation Models* [27]. Such models express and operationalize the goals of the system and define the adaptation on top of the reflection models. Hence, they specify the analyze and plan activities of a feedback loop, that is, when and how the adaptable software should be adjusted. As discussed in Section 2.2.3, three different ways of specifying an adaptation exists [247] and each of them uses different types of models that we consider as runtime models, particularly as adaptation models.

Action-based approaches employ adaptation rules, often Event-Condition-Action (ECA) rules, that describe when (periodically or at the occurrences of context or system events) and under which conditions the software is adapted by performing reconfiguration actions. Such rules are models that specify explicitly the adaptation behavior and examples of such adaptation models can be found in [36, 114, 140, 170, 176, 8, 307, 308]. Goal-based approaches define goals that the self-adaptive system should achieve and the self-adaptation should drive the system into a state in which the goals are fulfilled. Therefore, the goals must be represented in the system, which is typically done by goal models [58, 65, 106, 342, 352, 427]. Utility-based approaches define an objective function that quantifies for each feasible state of the adaptable software the desirability of the state. By adaptation, the software should then be driven to the most desirable of all feasible states. Such objective functions are usually called utility functions and they are used in [157, 161, 367, 429].

Such rules, goal models, and utility functions are related to the reflection models, for instance, to evaluate whether the current state of the adaptable software fulfills the conditions of the ECA rules or the goals, or to compute the current utility of the current state.

Likewise, they are used for planning by relating them to predictive reflection models that represent planned target states of the adaptable software. Since such rules, goal models, and utility functions determine the analyze and plan activities, that is, the adaptation behavior, we consider them as adaptation models.

Besides such adaptation models, to express conditions of the rules, to operationalize goals, or to guide an adaptation, constraints and properties can be formulated and checked at runtime against descriptive and predictive reflection models. For instance, if an architectural constraint is violated on a descriptive structural model, an adaptation can be triggered (*cf.* [170]), or constraints may exclude certain kind of adaptations when predicting the future behavior—based on a predictive behavioral model—of the adaptable software. Constraints and properties can be expressed, among others, in the Object Constraint Language (OCL) (*cf.* [203, 28]) or in formal languages such as Linear Temporal Logic (LTL) (*cf.* [192]), Probabilistic Computation Tree Logic (PCTL) (*cf.* [98, 150]), or Continuous Stochastic Logic (CSL) (*cf.* [98]).

While constraints and properties are mainly used for analyzing the current or planned states of the adaptable software, hence belonging to the category of evaluation models (see Figures 9 and 10), the planning of adaptations is determined by change models that define the variability of the software. For instance, ECA rules specify the variability implicitly because they have to be applied on reflection models to obtain a new variant of the software. In contrast, the variability can also be defined explicitly by describing the components types of the software with their different realizations [160, 161, 22] or by feature models [107, 147, 179, 306]. Such models span the configuration space of the adaptable software and they are used to find a configuration that fulfills the goals or achieves an acceptable utility.

Finally, there does not have to be a clear distinction between evaluation and change models, that is, whether the models are exclusively used for analysis or planning. For instance, ECA rules realize the analyze (evaluating the condition) and the plan (applying the actions) activities in one step such that the purpose of evaluation and change models are achieved by one adaptation model [27].

The presented categorization shows that many research approaches simultaneously use different kinds of (runtime) models in self-adaptive software systems. Thus, it demonstrates the popularity of using models at runtime. Moreover, it provides an insight into which kind of models are used and how they are used. For instance, the work of Morin et al. [306] uses an architectural runtime model (system model) of the adaptable software, a model of the context (environment model), a feature model for the variability (change model), and ECA rules (evaluation and change model). These rules describe which feature should be (de-)activated on the architectural model depending on the context model.

However, the state of the art considers only reflection models as runtime models by following the definition of a runtime model given by Blair et al. [79], which covers only models of the adaptable software and environment (*cf.* Section 2.1.5). For instance, at the Dagstuhl Seminar on Models@run.time [49] almost all participants, who have presented their individual perspectives on runtime models (*cf.* [340]), refer either directly to this definition or to characteristics for runtime models that are mentioned in the definition. Though Blair et al. [79, p. 25] admit "that in practice, it is likely that multiple [reflection] models will coexist and that different styles of models may be required to capture different system concerns", other kinds of runtime models such monitoring, evaluation, change, and execution models are not considered. Concrete approaches use evaluation and change models (adaptation models) without explicitly treating them as runtime models and fairly neglect monitoring and execution models (causal connection models).

However, we argue that such a narrow perspective on runtime models limits their potential for self-adaptive software. State-of-the-art approaches to self-adaptive software systems already employ different kinds of models at runtime and we argue to treat all of them explicitly as runtime models to support engineering. For instance, the categorization refines and clarifies the notion of knowledge in MAPE-K feedback loops, which is often abstract and less clear [435], and therefore supports the engineering of feedback loops.

Which categories of runtime models are used, and which kind of and how many models for each of the used categories are employed is specific to each self-adaptive software system. This depends, among others, on the domain of the system, the purpose such as which functional and non-functional concerns are of interest for the self-adaptation, and which adaptation activities (monitor, analyze, plan, or execute) are supported.

Finally, it is important to note here that using multiple runtime models in a feedback loop requires means to handle the interplay between these models. Such interplay actually forms the feedback loop and we address it with runtime megamodels.

4.2 RUNTIME MEGAMODELS

Megamodels have been proposed to manage the models used in software development (*cf.* Section 2.1.4). In this context, a variety of interrelated models at different abstraction levels and expressed in different languages are used to describe the software under development. Such models are continuously created, changed, analyzed, or transformed manually by engineers or automatically by tools, which affects related models. To manage development, all models and the relationships between them have to be tracked. This tracking can be achieved by a megamodel that captures the models and their relationships while supporting their manipulation. For instance, a model operation such as model synchronization can be associated with a relationship between two models to automatically propagate changes between these models if one of them is manipulated. Thus, a megamodel enables an overview of the used models and model operations, and it supports their manipulation.

Similar issues exist for *runtime models* and relationships between such models in self-adaptive software. As discussed in Section 4.1, multiple runtime models are simultaneously used and they are related with each other. For instance, the monitor activity of a feedback loop maintains multiple reflection models to provide different views of the adaptable software, or the analyze and plan activities apply adaptation models on reflection models to decide about an adaptation. This requires capturing the interplay of various runtime models, which actually forms the feedback loop. While there exists approaches to manage multiple models during development (*cf.* Section 2.1.4), the problem of managing multiple *runtime* models and their relationships/interplay is neglected. To the best of our knowledge there is no approach that explicitly considers this problem beyond ad-hoc, code-based solutions.

Therefore, we have proposed *runtime megamodels* [31, 32] to leverage similar benefits of development-time megamodels at runtime. Likewise to a development-time megamodel as "a model that contains models and relations between them" [211, p. 7] (*cf.* discussion in Section 2.1.4), a runtime megamodel captures and maintains two aspects,(1) runtime models and (2) the relationships between these runtime models, while it is kept alive at runtime. Concerning (1), the kinds of runtime models used in self-adaptive software have been discussed in Section 4.1 and a runtime megamodel captures such models. With respect to (2), we interpret the adaptation activities of a feedback loop such as monitor, analyze, plan, and execute as *runtime model operations* that relate the various runtime models. That is, adaptation activities have runtime models as input or output and they operate on these

models. An output model of one activity can then be the input model for another activity. For instance, considering Figure 9 on Page 42 the analyze activity has evaluation and reflection models as inputs and it outputs the reflection models annotated with analysis results obtained by applying the evaluation models on the reflection models. These annotated reflection models are then used by the plan activity to project an adaptation. Thus, the adaptation activities determine the interplay between the runtime models to form a feedback loop and they are therefore captured in a runtime megamodel.

Such a runtime megamodel results in two major benefits for self-adaptive software. First, it makes the employed runtime models, the adaptation activities, and the interplay between these models and activities *explicit* by capturing and organizing them. Consequently, the runtime models become amenable for analysis and adaptation that can either occur online or off-line (*cf.* Section 2.2.2). Particularly, we consider runtime models that represent the adaptable software and environment as well as capture information shared among the adaptation activities (*cf.* reflection models in Section 4.1) and that specify the behavior of the adaptation activities (*cf.* causal connection and adaptation models in Section 4.1). In other words, runtime models either externalize the working data or the specification of adaptation activities such that the activities are often stateless and just execute the specification while operating on the working data. Therefore, analysis and adaptation of feedback loops especially targets the runtime models, for which model-driven techniques can be exploited (*cf.* Section 2.1). In contrast, implicit models are hard-coded or generated into the adaptation logic and therefore, not directly and easily adaptable. In this context, the flexible nature of (explicit) runtime models supports changeability. Since changing a model is usually easier than modifying (compiled) code, adaptation is eased. Moreover, models are usually at a higher level of abstraction than code such that adaptation may happen at an appropriate level of detail, which further supports changeability.

Second, a runtime megamodel does not only structurally capture the runtime models, adaptation activities, and the interplay between them but it further supports execution. That is, it treats adaptation activities (*i. e.*, runtime model operations) as executable units that take runtime models as input and produce runtime models as output. Consequently, a runtime megamodel can be considered as an executable process for self-adaptation.[3] If the runtime megamodel as the specification of the process is detailed enough, mechanisms such as an interpreter can be employed to execute the process and thus, the self-adaptation.

Therefore, we define a *runtime megamodel* as

> *a runtime model that specifies the feedback loops in a self-adaptive software system by describing the elements of these feedback loops, which are runtime models and adaptation activities, as well as the interplay between these elements, and that enables direct execution of the model to change an adaptable software at runtime, that is, to perform self-adaptation.*

Thus, a runtime megamodel is itself a runtime model since it is maintained at runtime and it is executable to run the feedback loops of the system. While it captures the runtime models, adaptation activities, and their interplay within the feedback loops, the definition does not restrict the number and kinds of(i) runtime models (*i. e.*, the categories of runtime models and the languages to express runtime models), (ii) adaptation activities (*i. e.*, the activities can follow arbitrary schemes discussed in Section 2.2.2 such as collect-analyze-decide-act, monitor-analyze-plan-execute, sense-plan-act, or observe-decide-act), and (iii) interplay (*i. e.*, which feedback loop elements are interrelated and how they are

3 Self-adaptation is also considered as a process in [389, 416] but without an underlying runtime megamodel.

interrelated). For instance, the interplay may describe dependencies between runtime models if they overlap, the input/output relationships between activities and models, or the control flow between activities. In this context, Figure 9 on Page 42 can be seen as a sketch of a runtime megamodel that captures the monitor, analyze, plan, and execute activities, the control flow between these activities, various runtime models, and finally which activities have which runtime models as input or output.

From a technical point of view, a runtime megamodel can uniformly capture every kind of runtime model discussed in Section 4.1 because they are unified in a common MDE infrastructure, that is, they share a common meta-metamodel (*cf.* Section 2.1.1). Consequently, a runtime megamodel may apply generic MDE techniques to process any runtime model, for instance, to load, navigate, or adapt the runtime models. Likewise, a runtime megamodel itself is just a model and therefore, the same MDE techniques can be applied on it, for instance, to reflect upon and adapt a feedback loop. This aspect of reflection and runtime megamodels is outlined in the following.

4.3 REFLECTION

As outlined in the previous section, runtime megamodels promote the changeability of a feedback loop as they make its runtime models and adaptation activities as well as the interplay between both explicit. Therefore, the elements of a feedback loop become amenable for adaptation. Primarily, the runtime models are either the executable specifications of the activities or they capture the state and working data of the activities. As a consequence, the activities are often stateless and adaptation mainly targets the runtime models. The flexible nature of explicit runtime models eases their adaptation in contrast to implicit feedback loops with hard-coded and compiled mechanisms.

Adapting a feedback loop requires a reflection of it (*cf.* Sections 2.1.5 and 2.2.3). Based on the reflection, an agent analyzes and changes the feedback loop. For instance, an agent may analyze how well the feedback loop performs and—if needed—it may adapt or exchange a runtime model within the feedback loop.

If the feedback loop is specified by a runtime megamodel, this runtime megamodel can be directly used as a reflection (model) of the feedback loop. That is, the agent may directly analyze and change the runtime megamodel to adapt the feedback loop. Using the same representation—in terms of the runtime megamodel—for specifying/executing and adapting a feedback loop corresponds to the idea of procedural reflection (*cf.* Section 2.1.5). The advantage of procedural reflection is that the causal connection is ensured by construction because there is only one representation of the feedback loop, which is the runtime megamodel. However, the reflection model is at the same abstraction level as the specification of the feedback loop. Moreover, it does not allow the agent to preview the adaptation since the change is immediately enacted to the running feedback loop.

In contrast, declarative reflection (*cf.* Section 2.1.5) separates the reflection model from the specification of the feedback loop. This separation supports reflection models at higher levels of abstraction than the specification we well as previewing an adaptation in the reflection model before enacting it to the feedback loop. However, this separation requires maintaining the causal connection, that is, the synchronization of the reflection model and the specification. Reflecting in this manner on a feedback loop specified by a runtime megamodel requires maintaining a causal connection between the reflection model and the runtime megamodel. Technically, a runtime megamodel is just a model such that model synchronization techniques (*cf.* Section 2.1) can be used to realize the causal connection.

Consequently, runtime megamodels eases the reflection on feedback loops since they can be either used directly as reflection models or MDE techniques such as model synchronization can realize the causal connection.

An agent that reflects on a feedback loop can be another feedback loop that is typically located at a higher layer in the architecture. In general, layered architectures of self-adaptive software promote the idea that feedback loops at a certain layer reflect on and adapt feedback loops at the layer directly below (*cf.* Section 2.2.3). Considering feedback loops that are specified by runtime megamodels, such a layered architecture results in stacking runtime megamodels. For procedural reflection, the runtime megamodel specifying a lower-layer feedback loop is then used as a normal runtime model within the runtime megamodel specifying the higher-layer feedback loop. For declarative reflection, the runtime megamodels are still stacked. However, an explicit runtime model, particularly, a reflection model of the lower-layered feedback loop is used within the higher-layer feedback loop. That is, the higher-layer megamodel uses the reflection model and realizes the causal connection between this reflection model and the lower-layer megamodel.

Besides stacking feedback loops, agents reflecting on feedback loops can be engineers who maintain the self-adaptive software. To do this, they can manually and directly analyze and change a runtime megamodel to adapt the corresponding feedback loop. Thus, runtime megamodels as reflections of feedback loops enable automated adaptation (by stacked feedback loops) and manual maintenance (by engineers) of the adaptation logic. This paves the way for the co-existence of self-adaptation and maintenance (*cf.* Section 2.2.2).

Summing up, runtime megamodels leverage reflection by being themselves models. This requires a modeling language (*cf.* Section 2.1) to express runtime megamodels. Consequently, we propose Executable Runtime Megamodels (EUREMA), which is a modeling language to express feedback loops and an interpreter to run these feedback loops by executing the resulting models. Runtime megamodels as discussed in this section is the underlying principle of EUREMA. In the following section, we give an overview of EUREMA.

4.4 OVERVIEW

To express and execute *runtime megamodels* as discussed in Section 4.2, we propose and present in this thesis EUREMA (*ExecUtable RuntimE MegAmodels*) [25]. It follows the external approach to self-adaptive software, is inspired by the MAPE-K reference model for feedback loops and by Models@run.time principles, and adopts a layered architecture based on the idea of adaptive control (*cf.* Sections 2.1.5 and 2.2.3). Specifically, EUREMA supports the engineering of an adaptation engine that employs multiple feedback loops in a layered fashion. Thereby, it assumes that the adaptable software has been instrumented with sensors and effectors for monitoring and adapting the software and that these sensors and effectors are exposed to the adaptation engine.[4] To achieve this support for engineering adaptation engines, EUREMA provides a domain-specific modeling language and an interpreter to specify and execute feedback loops. The same language and interpreter are furthermore used to evolve feedback loops, either by on-line adaptation (*i. e.*, layered feedback loops) or off-line adaptation (*i. e.*, maintenance).

In general, we consider EUREMA as a domain-specific approach since it is focused on specifying and executing feedback loops based on the idea of runtime megamodels. Consequently, the EUREMA language is able to capture the feedback loop elements (*i. e.*, runtime

4 This assumption is also made by other researchers [113, 170, 307, 395] and it has been discussed in the context of requirement R3 (Sensors and Effectors) in Chapter 3.

models and adaptation activities) and the interplay between these elements. The interplay is substantiated by EUREMA to the usage of runtime models by adaptation activities and to the control flow among these activities to form a feedback loop. Such elements and the interplay map to concepts in the domain of self-adaptive software following the external approach, the MAPE-K reference model, Models@run.time principles, and layered architectures. Hence, engineers use such known domain-specific concepts that are leveraged by the EUREMA language to specify feedback loops for the self-adaptation problem at hand while execution support is directly provided by the EUREMA interpreter to run the feedback loops. Thereby, using a different language for modeling the adaptation engine with its feedback loops than for the adaptable software promotes separating adaptation concerns from business logic concerns.

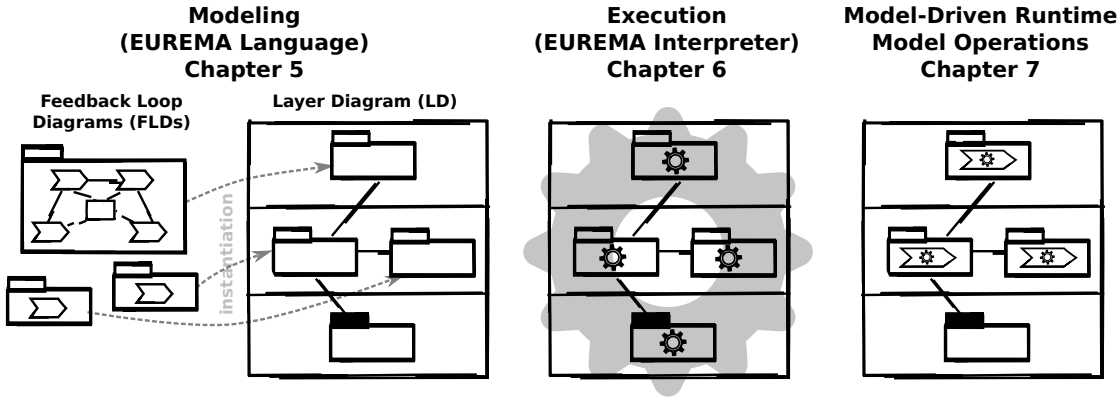


Figure 11: Overview of EUREMA.

An overview of EUREMA is shown in Figure 11, which is also used as a guide for the next chapters of this thesis to discuss the approach. EUREMA as a visual modeling language provides two types of diagrams that are used by engineers to model feedback loops. A behavioral *Feedback Loop Diagram (FLD)* is used to model a feedback loop or individual adaptation activities and runtime models of a feedback loop. Thus, an FLD encapsulates the specification of a partial or complete feedback loop. Such specifications (FLDs) are instantiated to *megamodel modules* to be used in a *Layer Diagram (LD)*. In EUREMA, the specification and the runtime instance of a feedback loop eventually collapse because instances can be individually adapted at runtime (*cf.* discussion in the next paragraph), which changes their individual specifications. An LD is a structural diagram that describes how the megamodel modules and the adaptable software are related to each other in an instance situation of the self-adaptive software. Though the adaptable software is not specified by EUREMA, it is depicted as a *software module* in the LD (*cf.* packages with a black tab in Figure 11). Thus, engineers model the self-adaptive software by creating FLDs to describe the behavior of feedback loops and an LD to describe the layered architecture. Thereby, FLDs provide white-box views of feedback loops as they capture the individual elements of the loops while an LD considers feedback loops encapsulated in megamodel modules as black boxes. Hence, EUREMA models specify feedback loops and their structuring in layered architectures and they make the feedback loops explicit in the architectural design of self-adaptive software. In this context, EUREMA does not impose any restrictions on the number and structure of feedback loops or on the number of layers. The modeling of self-adaptive software with the EUREMA language is discussed in detail in Chapter 5.

The EUREMA models describing the self-adaptive software are kept alive at runtime and they are directly executed by an interpreter. This avoids bridging the conceptual gap between domain-oriented specifications such as EUREMA models and implementation technologies such as code. In contrast, EUREMA seamlessly integrates the development and runtime environments as it allows the use of the same models in both environments and the exchange of these model between the environments. Additionally, the executability makes EUREMA models amenable for simulation to analyze the self-adaptation in the development environment. To execute EUREMA models in order to run the feedback loops, EUREMA provides an interpreter. This interpreter further supports exchanging models and therefore knowledge between the development and runtime environments, and it enables the evolution of feedback loops by means of on-line and off-line adaptation. On-line adaptation refers to stacking feedback loops in layered architectures while off-line adaptation corresponds to the maintenance of feedback loops. On-line and off-line adaptation is enabled by leveraging principles of runtime megamodels in the EUREMA language and interpreter. On the one hand, EUREMA models as runtime megamodels keep alive the runtime models and adaptation activities of feedback loops (*cf.* Section 4.2). This makes the runtime models and activities amenable for adaptation after the initial development of the feedback loops. On the other hand, EUREMA supports the reflection on feedback loops by exploiting the flexible nature of runtime megamodels to stack feedback loops (*cf.* Section 4.3). Reflection is a prerequisite for adapting feedback loops either automatically by higher-layer feedback loops or manually by engineers. Thus, realizing the principles of runtime megamodels, EUREMA supports the seamless design, execution, and evolution of feedback loops. We discuss the evolution of feedback loops together with the modeling of feedback loops in Chapter 5 while we focus on the execution of EUREMA models and the EUREMA interpreter in Chapter 6.

Afterwards in Chapter 7, we focus on the realization and execution of individual adaptation activities of feedback loops. In EUREMA, we consider such activities as runtime model operations that operate on runtime models. For the realization, we particularly consider model-driven operations. Such operations are specified themselves by models that are kept alive at runtime for their execution (*cf.* Sections 4.1 and 4.2). These models can be directly changed to adapt the behavior of the operations. Thus, this approach supports changeability of adaptation activities. In this context, we additionally emphasize the openness of EUREMA since engineers may use or even reuse arbitrary types of models and execution engines to specify and run adaptation activities. Even completely code-based activities can be integrated. The reuse of existing languages and execution engines eases the development of the adaptation activities.

For the execution of individual adaptation activities, we discuss state- and event-based operations. A state-based operation performs its task by processing runtime models that comprehensively represent the state and thus knowledge about the adaptable software and context. In contrast, an event-based operation performs its task by processing events that notify about individual changes of the adaptable software or context. Hence, instead of processing the whole state, individual changes are processed incrementally. In EUREMA, we especially consider the combination of state- and event-based operations to benefit from runtime models providing comprehensive knowledge and to enable an incremental execution that improves the runtime performance of feedback loops.

Consequently, EUREMA is an integrated MDE approach that consistently uses models for designing, executing, and evolving feedback loops with their constituting runtime models and runtime model operations. EUREMA models explicitly capture and maintain the

runtime models used within an adaptation engine, the interplay between these models, and the feedback loops working on these models. Therefore, the evolution of runtime models and feedback loops continues at runtime beyond the initial development of the software. The same EUREMA models used for the design and specification of self-adaptive software are also used for executing the feedback loops enabling a seamless transition from the design and specification to the execution as well as the adaptation and evolution of feedback loops. While the interpreter-based approach enables the flexibility to maintain and dynamically adapt feedback loops in layered architectures, the EUREMA interpreter efficiently executes feedback loops by introducing a low overhead and by leveraging state- and event-based principles for an incremental execution. Hence, EUREMA aims for addressing the "lack of powerful languages, tools, and frameworks that could help realize adaptation processes [...] in a systematic manner" [370, p. 31] while furthermore considering the execution and evolution of the adaptation processes.

In the following section, we introduce the running application example that will be used in Chapters 5 and 6 to discuss the modeling and execution of feedback loops in EUREMA and how EUREMA fulfills the requirements for modeling and execution stated in Chapter 3. Thus, from the point of view of a modeling language, we first focus on the use of the language (*i. e.*, modeling) and then on the operational semantics of the language (*i. e.*, execution). This discussion will be completed by Chapter 7 discussing the internals of feedback loops in EUREMA by means of the model-driven runtime model operations.

4.5 RUNNING EXAMPLE: MRUBIS

Throughout this thesis, we use a running example to illustrate EUREMA. The example is the Modular Rice University Bidding System (mRUBiS) [18], an internet marketplace on which users sell or auction products. It is based on RUBiS[5], which is a popular case study for self-adaptive software systems that manage performance [343]. We extended the RUBiS with new functionalities, migrated it to version 3 of Enterprise Java Beans (EJB) [132], and modularized it into 18 components that are individually deployed. The modularization enables architectural adaptations that are not possible with the monolithic RUBiS.

The company running mRUBiS aims for high sales volumes by achieving customer satisfaction and encouraging customers to additional purchases. Therefore, the system should be highly available and the response times should be low or at least acceptable. To reach these properties, self-adaptation should be employed to automatically repair failures (*i. e.*, self-healing or self-repair by detecting, diagnosing, and recovering from disruptions [184, 350]) and improve performance (*i. e.*, self-optimization by reconfiguring the system [443]). For instance, to mitigate faulty components, they can be restarted, redeployed, or replaced with alternatives. Performance can be improved by reconfiguring the pipe of filter components that improves the quality of search results on the marketplace, for instance, by taking user preferences into account. Changing the number of filter components and reordering them result in different performance characteristics that are exploited for self-optimization.

In this thesis, we will use the self-repair and self-optimization cases of mRUBiS to discuss EUREMA. In this context, the focus is on modeling, executing, and evolving feedback loops with EUREMA and not on specific mechanisms used within feedback loops to analyze and plan adaptations (*e. g.*, reasoning or model-checking techniques). Consequently, we use an action-/rule-based approach for such mechanisms since it is a basic approach and the foundation for more sophisticated goal- and utility-based approaches [247].

5 Rice University Bidding System (RUBiS): http://rubis.ow2.org/.

5 MODELING

In this chapter we present the EUREMA language and its usage to model self-adaptive software. Particularly, we apply the language to model variants of self-adaptive software covering single feedback loops, multiple feedback loops within one layer or at different layers of the architecture, and off-line adaptation (see Sections 5.1 to 5.4). All these variants are based on the mRUBiS application example introduced in Section 4.5 and cover self-repair as well as self-optimization capabilities. Since our focus is on introducing the EUREMA language and how it is applied to model feedback loops, we use a basic action/rule-based approach for the adaptation logic. This avoids unnecessary complexity in the examples. Moreover, such a basic approach is the foundation of more sophisticated approaches that use goals and utilities to realize the adaptation logic [247].

5.1 FEEDBACK LOOP

We start discussing the EUREMA language and its features by modeling a single feedback loop. For this purpose, we use two types of diagrams provided by EUREMA: the Feedback Loop Diagram (FLD) and the Layer Diagram (LD). We provide an overview of the language by modeling the MAPE-K blueprint in the next section and we use the running mRUBiS example (*cf.* Section 4.5) in the further sections to discuss all aspects of the language.

5.1.1 *Language Overview with MAPE-K*

To shed light on the EUREMA language, we first model the conceptual MAPE-K feedback loop before modeling the specific feedback loops for the mRUBiS example. The MAPE-K blueprint for feedback loops has been discussed in Section 2.2.3 and it considers a sequence of monitor, analyze, plan, and execute activities that all share some knowledge.

To handle the individual adaptation activities and the knowledge materialized by runtime models within a feedback loop as well as the interplay between them, EUREMA adopts the principles of *runtime megamodels* (*cf.* Section 4.2). Thus, we consider adaptation activities as *model operations* that operate on *runtime models*. The interplay is reflected in EUREMA by the *control flow* among the operations and the *model usage* defining how operations use the runtime models. To model a feedback loop with its model operations, runtime models, control flow, and model usage, the EUREMA language provides the Feedback Loop Diagram (FLD). Such a diagram specifies the behavior of a feedback loop.

The FLD for MAPE-K is shown in Figure 12. It is framed and labeled with its name MAPE-K. It specifies a feedback loop with an initial and a final state (Start and Executed, respectively) as entry and exit points for the execution of the loop. The four adaptation activities of MAPE-K are specified as *model operations* that are represented by hexagon block arrows and labeled with their names (Monitor, Analyze, Plan, and Execute). Each operation has an exit compartment that specifies the operation's return status such as monitored of the Monitor operation. The *control flow* between operations is specified by solid arrows, each connecting an exit compartment of an operation to a subsequent operation. Hence, this FLD specifies a strict sequence of the Monitor, Analyze, Plan, and Execute operations.

Operations work on *runtime models* that are represented by rectangles. The FLD shows one runtime model, called Knowledge, that captures the knowledge part of MAPE-K. The *usage of models* as inputs or outputs of an operation is defined by dotted arrows pointing to the operation or to the model, respectively. The FLD specifies that each of the four model operations works on the Knowledge model by having this model as an input and output.

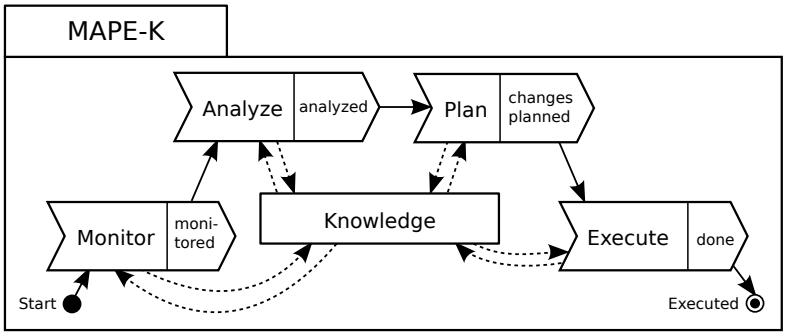


Figure 12: Feedback Loop Diagram (FLD) for MAPE-K.

This view of the MAPE-K feedback loop keeps the knowledge abstract while a refined view is proposed in [124]. Particularly, the generic notion of knowledge is refined to topology knowledge about the adaptable software, symptoms indicating issues to be analyzed, change requests showing the need for an adaptation, policies defining the adaptation logic, and change plans describing the adaptation to be executed. The corresponding feedback loop using these refined runtime models is specified by the FLD shown in Figure 13.

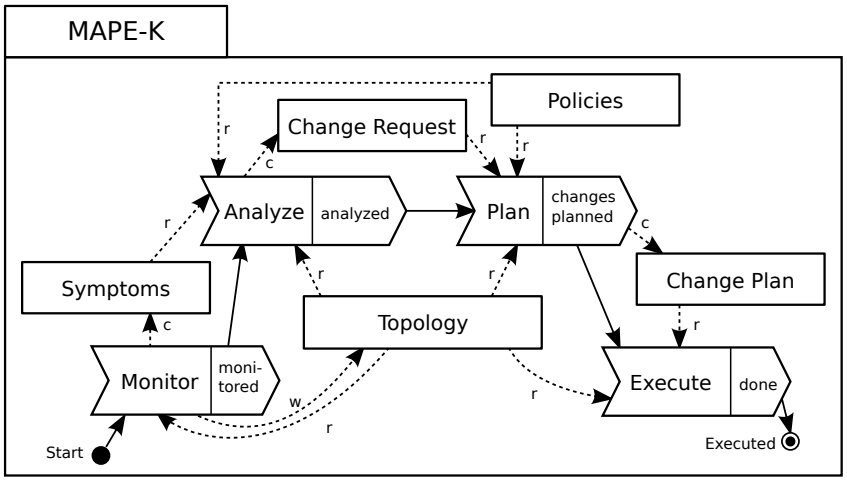


Figure 13: Refined Feedback Loop Diagram (FLD) for MAPE-K according to [124].

In addition to refining the models, we substantiate the operations' use of the models. Particularly, we distinguish between **c**reating, **d**estroying, **w**riting, **r**eading, and **a**nnotating runtime models. In Figure 13, the Monitor operation observes the adaptable software. According to the observations, it reads and writes (*i. e.*, updates) the Topology model representing the topology of the adaptable software and creates Symptoms that should be analyzed. The Analyze operation reads these Symptoms to locate issues in the up-to-date Topology and checks them against the Policies. If a need for adaptation has been identified, the Analyze operation creates a Change Request. The Plan operation uses the Change Request, the Topology,

and the Policies to create a Change Plan prescribing an adaptation for execution. This plan is finally enacted by the Execute operation taking the current Topology into account.

To keep this FLD simple, we omitted the modeling if and when certain models are destroyed by operations. On the one hand, models can be continuously used by operations across multiple runs of the feedback loop such that they are permanently maintained at runtime and not destroyed by operations. Examples are the Policies and the Topology models. More specifically, the Topology model is updated and not created from scratch by the Monitor operation in each run of the feedback loop such that it is preserved at runtime. On the other hand, models can be used only temporarily, for instance, to exchange information among operations, such that they can be destroyed by operations after they have fulfilled their purpose. For instance, the Monitor operation creates the Symptoms model to provide hints for the analysis to the Analyze operation that could destroy the model after having used these hints. Likewise, the Plan and Execute operations could destroy the Change Request respectively the Change Plan model when having finished the planning respectively the execution. Such a destruction of a model is reflected in the FLD by a model usage link (*i. e.*, dotted arrow) pointing to the model and labeled with **d**. However, if a runtime model is not destroyed by an operation, then a repeated creation of the model denotes the overwriting of this model, which implicitly destroys the model.

This example illustrates how adaptation activities can be considered as abstract model operations working on runtime models. Hence, an FLD as shown in Figures 12 and 13 captures the runtime models and model operations (*i. e.*, adaptation activities) of a feedback loop as well as the usage of the models by the operations. Moreover, an FLD describes the control flow among model operations to support executing the flow of operations. These characteristics of an FLD match the principles of a runtime megamodel (*cf.* Section 4.2).

Moreover, this basic example further shows that FLDs can be used iteratively to refine the design of a feedback loop. For instance, the initial design of the MAPE-K cycle described by the FLD in Figure 12 keeps the runtime models and their usage by the operations abstract and focuses on the flow of operations. A next design iteration may refine the runtime models and their usage, which results in the FLD shown in Figure 13.

While FLDs support the behavioral specification of feedback loops, EUREMA further provides the Layer Diagram (LD) that structurally reflects all feedback loop instances (*i. e.*, instances of FLDs) of a self-adaptive software as well as their relationships to each other and to the adaptable software instance. Since we consider self-adaptive software that follows the external approach and adopts a layered architecture, an LD provides an architectural view of an instance situation of the self-adaptive software that explicitly captures the layers as well as the feedback loops and adaptable software at various layers. Such an abstract and complete view of the self-adaptive software makes the external approach, the layered architecture, and all feedback loops visible in the architectural design.

The example LD for MAPE-K is depicted in Figure 14. This diagram reflects an instance view of the self-adaptive software and it specifies that an instance of the MAPE-K feedback loop as specified by the FLD in Figure 13 is located at Layer-1 and that it directly senses and effects the Adaptable Software instance at Layer-0. Sensing (effecting) relationships are reflected by dotted arrows, whose directions indicate a data flow, and labeled with r for reading (w for writing) the adaptable software instance. An instance of a feedback loop that is specified by an FLD is considered as a *megamodel module* and depicted as a package with a white tab in the LD. In contrast, packages with a black tab represent *software modules* such as an instance of the adaptable software that are not specified by EUREMA. Finally, partitions in LDs represent the layers of the self-adaptive software.

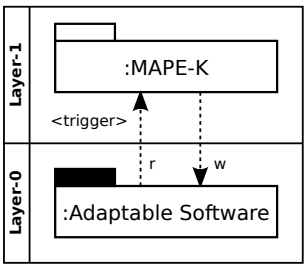


Figure 14: Layer Diagram (LD) for MAPE-K.

In addition to the modules, either instances of feedback loops or of the adaptable software, at various layers, the LD further defines the triggering of feedback loop instances, that is, *when* an instance should be executed. Since the triggering often depends on other modules, mainly the adaptable software instance that emits events notifying about changes to be handled by self-adaptation, it is addressed by LDs that capture the various modules and the relationships between these modules. That is, a megamodel module (*i. e.*, a feedback loop instance) may sense another module (*i. e.*, either another feedback loop instance or an instance of the adaptable software) and therefore the events emitted by this other module. Such events may trigger the execution of a feedback loop instance.

Thus, a trigger is defined in the LD by labeling the corresponding sensing relationship through which events are emitted and observed. Considering the LD in Figure 14, the trigger is annotated to the sensing relationship between the :MAPE-K feedback loop and the :Adaptable Software. Since the MAPE-K feedback loop is just a blueprint, we omit details of triggers here and discuss them for the following examples.

Having applied the EUREMA language to the MAPE-K reference model, we may already provide a rough overview of the language with its two types of diagrams. An FLD provides a white-box view and therefore a detailed behavioral specification of a feedback loop (*e. g.*, Figures 12 and 13) An LD provides a complete structural view of an instance of the self-adaptive software with black-box views of the feedback loops (*e. g.*, Figure 14). That is, feedback loops (FLDs) are instantiated to megamodel modules and structured in a layered architecture (LD) together with the adaptable software.

In the following, we will discuss EUREMA in detail for concrete and advanced cases of adaptation engines and feedback loops. Thereby, we will refine the modeling language for both types of diagrams. Consequently, we will discuss the entire concrete syntax of FLDs and LDs while we just gave a rough overview in this section to grasp the basic idea.

5.1.2 *Single Feedback Loops (Self-Repair)*

In this section, we start using the running mRUBiS example introduced in Section 4.5 to model concrete feedback loops with EUREMA in contrast to the conceptual MAPE-K blueprint. In the mRUBiS example, we consider different types of failures that should be automatically handled by self-repair capabilities. These types are component crashes (*i. e.*, an invalid life cycle state of a component) and destructions (*i. e.*, a component is removed from the system) as well as the occurrence of exceptions in components. Moreover, components can be continuously affected by these types of failures requiring special repair mechanisms. The mechanisms we consider are restarting, redeploying, reconfiguring, and replacing the affected components. Thus, the self-repair capabilities have to identify failures of various types and handle them by different repair mechanisms.

A feedback loop supporting these self-repair capabilities is specified by the FLD depicted in Figure 15. This FLD is framed and labeled with its name Self-repair. In contrast to the FLD for MAPE-K (see Figure 12 on Page 54), this FLD employs refined elements of the language. These elements enable the exclusive branching of the control flow. Therefore, a model operation may have more than one return status and thus more than one exit compartment used for continuing the control flow. At runtime, the implementation of the model operation determines the return status and therefore which exit compartment is activated. Moreover, the control flow can be exclusively branched using the decision node (diamond element) and conditions. The language for these conditions is used to create boolean expressions that refer to counter and timing information about the execution of EUREMA elements such as how often or when an operation has been executed lastly.

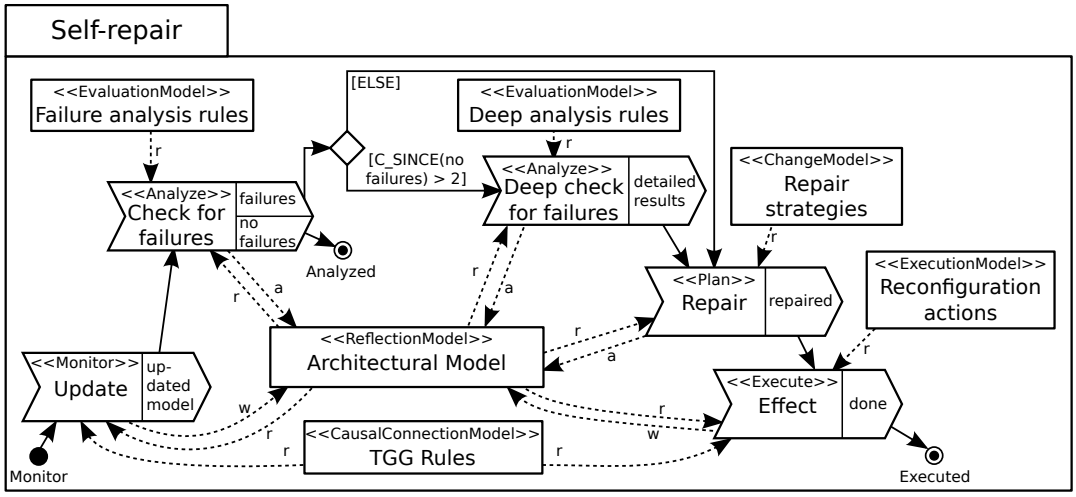


Figure 15: FLD for the Self-repair Feedback Loop.

Concerning the FLD in Figure 15, we modeled the self-repair feedback loop for mRUBiS by extending a scenario from our earlier work [22]. The Update and Effect operations use triple graph grammar rules (TGG Rules) that specify by means of model transformation rules how the Architectural Model reflecting the running mRUBiS system is synchronized with the system. Thus, monitoring the system, the Update operation keeps the Architectural Model up-to-date. The Check for failures operation performs analysis by applying Failure analysis rules on the Architectural Model. These rules define checks to identify occurrences of the different types of failures (*i.e.*, crash or destruction of a component, exceptions, and components being continuously affected by failures). If no failures are found, the feedback loop terminates in the state Analyzed. Otherwise, the identified failures are annotated in the Architectural Model such that they should be repaired by an adaptation. Furthermore, a decision is made whether further analysis is needed, which will be conducted by the Deep check for failures operation checking the Deep analysis rules on the Architectural Model. This is the case when the condition holds, which tests whether the last execution of the Check for failures operation that has identified no failures happened more than two consecutive executions in the past. Thus, the past three runs of the feedback loop's analysis has identified failures. After the analysis, the plan activity uses the analysis results annotated by one or both of the analyze operations to the Architectural Model to select suitable Repair strategies that define either a restart, redeployment, reconfiguration, or replacement of an affected component. The selected strategies are annotated to

the Architectural Model to prescribe an adaptation of the running mRUBiS system. This prescribed adaptation is executed by the Effect operation. Therefore, the Effect operation uses generic and basic Reconfiguration actions (*e. g.*, stopping, starting, deploying, undeploying, and wiring components) to adapt the Architectural Model as well as the TGG Rules to synchronize the adaptation of the model to the running system. The synchronization enacts the adaptation to mRUBiS, which terminates one run of the feedback loop.

This example as depicted in Figure 15 uses almost all elements of the EUREMA language with respect to FLDs such that we now give an overview of the concrete syntax for these elements (see Figure 16). *Initial* and *final states* are special operations that define the entry and exit points for executing a feedback loop instance. A *destruction state* is a final state that destroys the feedback loop instance when this state is reached. Adaptation activities are specified as *model operations* represented by hexagon block arrows and labeled with their names. A model operation has at least one named exit compartment, one for each return status of the operation. An exit compartment is the source of exactly one control flow link to another operation. At runtime, the implementation of the operation determines the return status and therefore which exit compartment is activated to continue the control flow. *Complex model operations* abstract from and invoke adaptation activities modeled in other FLDs, which will be discussed in Section 5.1.4.

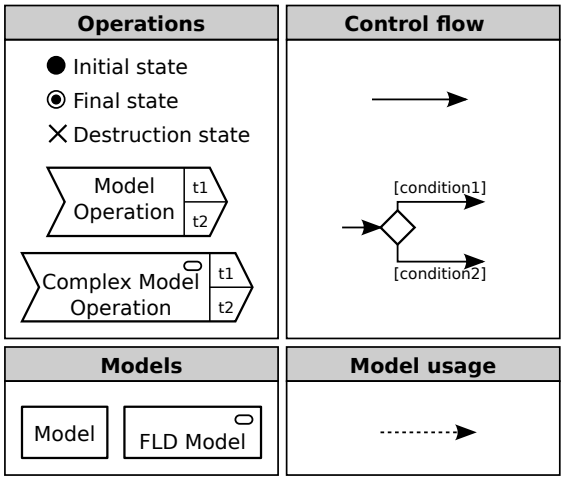


Figure 16: Concrete Syntax for FLDs.

The *control flow* between operations is specified by solid arrows connecting an exit compartment of an operation to another operation. The control flow can be exclusively branched by a decision node (diamond element). Each branch, that is, each of the decision node's outgoing control flow links, is labeled with a boolean condition. At run-time, the branch whose condition evaluates to true is taken. As a guideline, the conditions of the same decision node should be disjoint and complete, that is, exactly one of the conditions must evaluate to true. If an engineer cannot assure completeness (*i. e.*, none of the condition might evaluate to true), a default branch labeled with ELSE should be used. If more than one condition evaluates to true, the EUREMA interpreter will raise a runtime exception.

The language to express such conditions refers to counter and timing information about the execution of the feedback loop instance, particularly, how often and when an operation or a transition (*i. e.*, an operation's exit compartment with its outgoing control flow link to another operation) has been executed lastly, and for each transition, how often its source operation is consecutively executed without taking the transition. Such information allows

engineers to express conditions that refer to the time when and to the number of how often operations have been executed and individual return states (*i. e.*, exit compartments) of operations have been taken, either in absolute terms or with respect to the last consecutive execution runs. In a condition, such counter and timing information can be subject to arithmetic and boolean functions such that the condition eventually evaluates to true or false. The language to express such conditions is discussed in detail in Appendix A.3. Thus, conditions are generic and use only execution information related to elements of the EUREMA language to branch the control flow. By referring only to EUREMA elements, the conditions can be evaluated by the EUREMA interpreter without having any knowledge about the internals of user-defined model operations and runtime models.

In contrast, the different exit compartments of model operations and especially their activation may depend on application-specific information that is only known internally to user-defined runtime models and model operation implementations. For instance, the FLD in Figure 15 on Page 57 contains the Check for failures operation that analyzes the Architectural Model to identify failures. The result of the analysis determines the continuation of the control flow. Accordingly, the operation has two exit compartments indicating whether failures have been identified or not in order to branch the control flow. This analysis may depend on the various types of failures and content of the architectural model such that it cannot be generically performed by the EUREMA interpreter. Consequently, it must be performed by some application-specific logic and hence specified as a model operation with multiple exit compartments in the FLD. Such application-specific analyses cannot be captured by the generic conditions of decision nodes. Thus, application-specific or more complex cases of branching the control flow in FLDs must be realized with model operations and appropriate exit compartments. In general, engineers have to provide implementations for model operations—typically by providing adaptation models (*cf.* Section 4.1)—that may operate on arbitrary runtime models such that application-specific aspects can be addressed. At run-time, the implementation of a model operation determines which exit compartment of the operation is activated and should be taken by the EUREMA interpreter to continue the control flow in an FLD. Therefore, the application-specific aspects are transparent to the EUREMA language and interpreter.

As already mentioned, model operations work on runtime models. Such *models* are represented by rectangles and the *usage of models* is depicted by dotted arrows. An arrow connects a model operation and a model while its direction indicates the data flow and therefore, whether the model is an input or output of the operations. Arrows pointing from the model to the operation (from the operation to the model) indicate that the model is an input (output) of the operation. Finally, an *FLD model* describing a feedback loop can be used as a runtime model within another feedback loop specified by an FLD, which will be discussed in the context of layered feedback loops and reflection in Section 5.3. In general, runtime models are maintained and used across multiple runs of the same feedback loop instance such that they are not re-constructed or re-loaded in each run.

Additionally to the concrete syntax for FLD elements, labels and stereotypes can substantiate these elements. As already mentioned in Section 5.1.1, the usage of models by operations is substantiated to **c**reating, **d**estroying, **r**eading, **w**riting, and **a**nnotating models. Creating a model refers to producing the model from scratch. The repeated creation of a model within the same or in multiple runs of the feedback loop means the overwriting of the previously created model unless the model has not been explicitly destroyed. The destruction of a model removes the model from the feedback loop such that is not accessible any more by the operations. Instead of producing a model from scratch, it can also

be maintained and updated, which corresponds to reading and writing/annotating the model. While reading a model does not have any side effects, writes modify the model in a way that potentially affects the adaptable software (particularly in a controlled manner if the model is a reflection model and thus causally connected to the software), and annotations to a model enrich a model without affecting the adaptable software. Annotations are mainly used to enrich a model with working data to be shared among operations. For instance, the analyze operation of the Self-repair feedback loop shown in Figure 15 annotates the Architectural Model with the identified failures (*i. e.*, in general with the results of the analysis) that are afterwards addressed and removed by the plan and execute operations.

Following the MAPE-K blueprint, model operations are assigned to the typical steps of a feedback loop, which are ≪Monitor≫, ≪Analyze≫, ≪Plan≫, and ≪Execute≫. Refining the knowledge part of the MAPE-K blueprint to a set of runtime models as categorized in Section 4.1, models in FLDs are stereotyped with the category they fall in to indicate their purpose in the feedback loop. Based on the categorization, the following stereotypes ≪MonitoringModel≫, ≪ExecutionModel≫, ≪CausalConnectionModel≫, ≪ReflectionModel≫, ≪EvaluationModel≫, ≪ChangeModel≫, and ≪AdaptationModel≫ are possible.

Such labels and stereotypes substantiate the generic concepts of the EUREMA language. On the one hand, these labels and stereotypes support engineers in modeling and understanding FLDs since they classify the operations, runtime models, and the usage of these models by the operations based on the known domain concepts from the MAPE-K blueprint and Models@run.time field. For instance, stereotyping an operation with one of the MAPE steps clarifies the general task of the operation. Stereotyping a runtime model emphasizes the purpose of the model in the feedback loop. Labeling the link between a model and an operation details how the operation uses the model. Thus, the stereotypes and labels emphasize domain concepts in the EUREMA language.

On the other hand, using stereotypes and labels, we can keep the EUREMA language simple and limit the number of language elements. There are just four major language elements in FLDs, namely, operations, models, control flow, and model usage. This eases the comprehension of the language while supporting *optional* specializations of these elements by stereotypes and labels. In this context, the EUREMA language is extensible. Other stereotypes and labels can be integrated, for instance, to use other blueprints for feedback loops than MAPE-K such as sense-plan-act, learn-reason-act, or observe-decide-act (*cf.* Section 2.2.3) or another categorization of runtime models (*cf.* Section 4.1).

To specify how an instance of the Self-repair feedback loop is related to the adaptable software, specifically to the mRUBiS system, we use an LD as depicted in Figure 17.

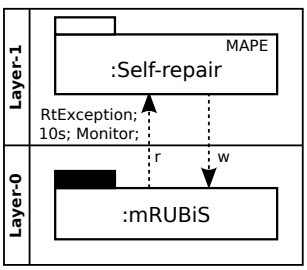


Figure 17: LD for the Self-repair Feedback Loop and mRUBiS.

This LD specifies that an instance of the Self-repair feedback loop as specified by the corresponding FLD (*cf.* Figure 15 on Page 57) is located at Layer-1 and directly senses and

effects the running mRUBiS instance at Layer-0. The sensing and effecting relationships are represented in the LD by arrows that are respectively labeled with **r**eading and **w**riting the mRUBiS instance. Since the FLD for the Self-repair feedback loop specifies all adaptation activities, that is, the monitor, analyze, plan and execute activities (MAPE), the instance of this feedback loop in the LD is labeled with MAPE. Inspired by [436], we use such labels to indicate which adaptation activities are specified in an FLD when using an instance of this FLD in the LD. In this context and depending on the applied blueprint for feedback loops, other labels such as SPA (sense-plan-act), LRA (learn-reason-act), or ODA (observe-decide-act) can be integrated and used in the EUREMA language.

Additionally, sense relationships are labeled with a triggering condition defining when the :Self-repair feedback loop should be executed. In this case, the triggering condition RtException;10s;Monitor; defines that the feedback loop instance is executed if the :mRUBiS emits an RtException event and if ten seconds since the end of the previous execution of the same instance have elapsed. Finally, Monitor points to the initial state of the feedback loop in which the execution should start (*cf.* initial state in the FLD in Figure 15 on Page 57).

The example LD in Figure 17 employs almost all of the language elements of LDs. Thus, we now discuss these elements together with their concrete syntax depicted in Figure 18.

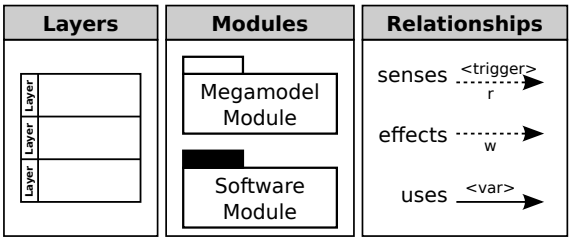


Figure 18: Concrete Syntax for LDs.

Partitions in LDs represent the *layers* of the self-adaptive software. Each layer has a name and contains *modules*. An instance of an FLD constitutes a *megamodel module* that encapsulates the details of a concrete feedback loop. Such a module is depicted as a package with a white tab in an LD since EUREMA provides white-box views of megamodel modules by means of the FLDs. In contrast, packages with a black tab represent *software modules* that are not specified by EUREMA such as an instance of the adaptable software. Thus, EUREMA entirely handles software modules in a black-box manner.

Sense and *effect relationships* between modules are reflected by dotted arrows, either labeled with r (for **r**eading the sensed module) or w (for **w**riting the effected module), respectively. The direction of such an arrow indicates the data flow such as the flow of sensor data or adaptation commands. A sense relationship is furthermore labeled with at most one triggering condition determining when the sensing module should be executed. The language and mechanism for triggers in EUREMA are discussed in detail in the next section.

Use relationships make the dependencies between modules explicit. Such dependencies occur because of three reasons.(1) Basic model operations contained in a megamodel module (*i. e.*, in an instance of the FLD specifying these operations) must be bound to their implementations provided by engineers. Such implementations are represented by software modules in LDs. Hence, the megamodel module containing the model operations depends on the software modules implementing these operations. (2) A dependency between megamodel modules exists if a feedback loop is specified by multiple FLDs and at run-time one megamodel module (*i. e.*, an FLD instance) invokes another one through

a complex model operation. In this case, the complex model operation contained in the invoking megamodel module must be bound to the invoked megamodel module. Hence, the invoking megamodel module depends on the invoked one. (3) Finally, a dependency occurs if a higher-layer feedback loop reflects on a lower-layer feedback loop. In this case, the higher-layer megamodel module uses the lower-layer megamodel module as a runtime model, that is, the higher-layer feedback loop instance operates on an FLD instance representing and specifying the lower-layer feedback loop instance. Consequently, the runtime model contained in the higher-layer megamodel module must be bound to the lower-layer module, which constitutes a dependency from the higher-layer to the lower-layer module.

To realize such bindings or generally the dependencies between the modules, engineers model use relationships in LDs. A use relationship is represented by a solid arrow pointing from the independent to the dependent module. Additionally, the relationship is labeled with a variable. Such a variable is the name of either the model operation, the complex model operation, or the runtime model that is contained in the source module and will be bound to the target module of a use relationship. The first two cases are further discussed in the context of modular feedback loops in Section 5.1.4 while the last case is detailed in Section 5.3 discussing the stacking of feedback loops.

Within the LD, the relationships between the modules and the structuring of these modules into different layers are related with each other. In layered architectures, a feedback loop at a certain layer may control (*i. e.*, sense and effect) a feedback loop at the layer directly below. To comply with this criterion of stacking feedback loops, *sense* (*effect*) relationships have to cross neighboring layers bottom-up (top-down) while the sensing (effecting) module is located at the layer directly above the sensed (effected) module. This criterion assures an acyclic structure of modules when following either the sense or the effect relationships. In this context, the EUREMA language enforces by construction that engineers respect the layering criterion such that they cannot specify cyclic structures (see Appendix A.1). This avoids the need to address termination and stability issues of cyclic adaptations among feedback loops due to infinite triggering of the feedback loops.

In contrast, the *use* relationships to bind model operations to implementations or complex model operations to megamodel modules (*cf.* cases (1) and (2) discussed previously) must not cross any layer. Such bindings are needed to compose a feedback loop within one layer but they do not establish any adaptation relationship to other feedback loops or to the adaptable software. These two cases are discussed in detail in Section 5.1.4. However, the binding of a runtime model at a certain layer to a megamodel module at the layer directly below (*cf.* case (3) discussed previously) relates to adaptation. The binding allows a higher-layer megamodel module to use a lower-layer megamodel module as a runtime model. Consequently, use relationships concerning the case (3) cross neighboring layers top-down and they come along with the sense and effect relationships among the corresponding modules to achieve procedural reflection (see Section 5.3 for a detailed discussion).

Summing up the basic modeling of feedback loops with EUREMA, an FLD encapsulates the behavioral specification of a feedback loop. At run-time, it is instantiated to a megamodel module. The LD considers such modules as black boxes and describes in which layers they are located and their relationships to other megamodel or software modules. Thus, an LD provides an abstract, structural view that supports the architectural design of self-adaptive software and that complements the behavioral FLDs. The LD forces engineers into specifying strictly layered architectures. In such architectures, feedback loops may only control other feedback loops at the layer adjacently below or eventually—if they are located that the lowest layer of the adaptation engine—the adaptable software.

5.1.3 *Triggering Conditions*

Besides specifying a feedback loop in an FLD, a triggering condition is required to determine *when* an instance of the feedback loop should be executed. In EUREMA, a feedback loop instance has exactly one triggering condition and we especially consider the occurrences of events as triggers. These events are emitted by instances of the adaptable software or of other feedback loops. For example, the adaptable software instance emits events notifying about changes such as a component crash that should be handled by self-adaptation.

Instances of feedback loops and of the adaptable software, which are megamodel and software modules, with their sense and effect relationships to each other are specified in an LD that therefore captures as well the triggering conditions. Hence, a megamodel module (*i. e.*, a feedback loop instance) may sense another module (*i. e.*, either another feedback loop instance or an instance of the adaptable software) and therefore the events emitted by this other module. Such events trigger the execution of the sensing feedback loop instance. In this context, a triggering condition for a feedback loop instance may only refer to events emitted from those modules that are sensed by this instance. This avoids that the feedback loop instance is triggered by events related to modules that are of no interest to the instance. Thus, we specify triggering conditions in LDs by annotating them to the corresponding *sense* relationships that reveal the flow of events from one module to another module.

This is exemplified by the LD in Figure 17 on Page 60 showing a triggering condition for the :Self-repair feedback loop that senses :mRUBiS. The condition RtException;10s;Monitor; defines that the feedback loop instance starts execution if :mRUBiS emits an event of type RtException notifying about a runtime exception, and when ten seconds since the last execution of this instance have expired. Finally, Monitor points to the initial state of the feedback loop (*cf.* FLD in Figure 15 on Page 57), in which the execution is going to start.

In general, EUREMA supports a simple language to express triggering conditions that consists of three parts: <events>; <period>; <initialState>;. The first part <events> refers to a comma-separated list of events. An event is characterized by a name and a type in a hierarchy of event types. It is consequently represented in a condition as EventType[EventName]. The event name can be omitted such that only the event type characterizes the event (*cf.* anonymous event). Such events and event types are modeled with EUREMA. At runtime, these events as part of triggering conditions are matched against events actually emitted by modules. An event matches another event if the latter is of the same type or of a subtype of the former and if both have the same name. For anonymous events, only the event types determine a match. If a match has been identified, that is, at least one of the events listed in the condition matches an emitted event, the trigger of the feedback loop instance is activated. Thus, the list of events in a triggering conditions are combined by a logical disjunction. For the given example condition, RtException represents an anonymous event of the given type. This event matches any emitted event if the latter is of the same type RtException or of a subtype of RtException. Such a match activates the trigger.

The second part <period> defines the minimal time period in seconds between two consecutive runs of the same feedback loop instance, which is measured as the time elapsed between the termination of the previous run and the beginning of the next run. Thus, if the required event that activates the trigger occurs before the specified time period has elapsed, the next run will be delayed until the period eventually has elapsed. Delaying the execution avoids thrashing effects due to the proliferation of events and it allows the adaptation being executed by the previous run to take effect in the adaptable software (*cf.* settling time [215]). Likewise, selecting specific events in the first part of the triggering

condition also serves as a filter that avoids the execution of a feedback loop instance for every event emitted by the sensed module.

Finally, the third part <initialState> just refers to the initial state specified in the FLD of the feedback loop, in which the feedback loop instance should start its execution.

The first two parts of a triggering condition are optional but one of them must be specified. If no events are specified, the period must be defined, which results in a trigger that periodically executes the feedback loop instance. If no period is defined, the events must be specified and the trigger initiates the execution of the instance when the corresponding events have occurred and the current run of the instance has terminated. In EUREMA, a feedback loop instance is not reentrant, that is, there are no concurrent executions of the same instance. All events that occur while the instance is running are queued. While an event-based trigger supports *reactive* adaptation, a periodical trigger enables *proactive* adaptation by executing a feedback loop before the adaptable software emits events. The grammar for the textual language to express triggering conditions is discussed in Section A.4.

5.1.4 *Modularity*

In the previous Sections 5.1.1 and 5.1.2, we specified each of the conceptual MAPE-K and the specific self-repair feedback loops in a single FLD. However, EUREMA supports a modular specification of a feedback loop by using multiple FLDs. Individual adaptation activities of a feedback loop are specified in distinct FLDs and afterwards composed to form a complete feedback loop. This composition happens in LDs where instance of these FLDs (*i. e.*, megamodel modules) are combined.

The motivation for such a modular specification is twofold. First, it provides further abstractions for engineers. Specifying a more complex feedback loop also makes the related FLD more complex and difficult to comprehend. To ease the modeling and perception of feedback loops in EUREMA, parts of a feedback loop can be abstracted, modeled in dedicated FLDs, and referenced by other FLDs. Second, a modular specification supports options for reuse and variability. Since parts of a feedback loop can be abstracted in dedicated FLDs and referenced by other FLDs, reusability of such parts in different feedback loops is supported. Additionally, such parts can be replaced by alternative parts specified by other FLDs, which leverages the modeling of variability for feedback loops in an adaptation engine as it will be discussed in the next section.

To discuss the modular specification, we use the Self-repair feedback loop specified by the FLD in Figure 15 on Page 57. For this feedback loop, we abstract from the analyze activities by specifying them in a distinct FLD. This FLD is called Self-repair-A. As shown in Figure 19, it describes the analyze operations of the self-repair feedback loop.

This FLD has one initial state (Start) and two final states reflecting whether failures have been identified (Failures) or not (OK) by the analyze operations. This FLD can be (re)used by other FLDs, which materializes at runtime by invocations between the corresponding megamodel modules (*i. e.*, FLD instances). Therefore, we introduce the concept of a *complex model operation* in EUREMA. Such an operation synchronously invokes a megamodel module by referring to the initial and final states of the corresponding FLD and to the runtime models that should be passed as parameters to the invoked module. These initial and final states as well as the runtime models are the signature of the invoked megamodel module as it is defined by the corresponding FLD. Complex model operations have to follow such signatures to invoke the corresponding megamodel modules. To (re)use the Self-repair-A FLD shown in Figure 19, the complex model operation is depicted in two variants in Figure 20.

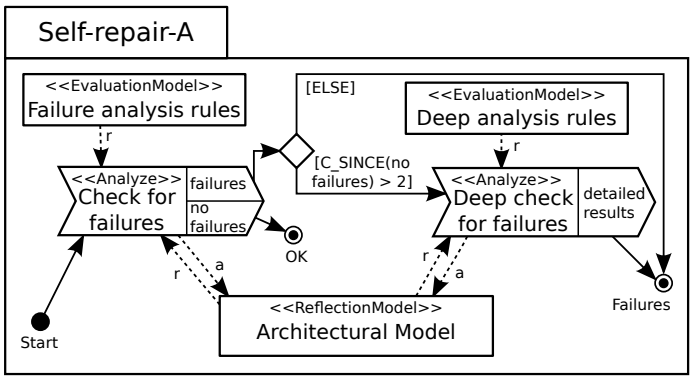


Figure 19: FLD for the Analyze Activities of the Self-repair Feedback Loop.

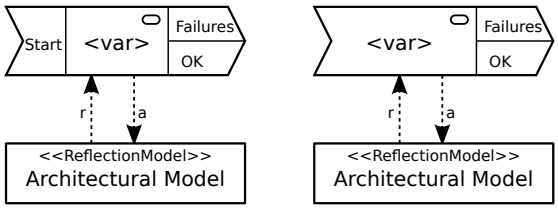


Figure 20: Complex Model Operation.

Based on the initial and final states of the Self-repair-A FLD, variant (a) has one entry compartment called Start and two exit compartments called Failures and OK. Thus, initial and final states of an FLD are mapped to entry and exit compartments of a complex model operation, respectively. This ensures that a feedback loop instance using a complex model operation can properly invoke an instance of the FLD by defining the initial state for starting the invocation. Moreover, it can properly resume execution after the invocation by referring to the final states and thus to the result of the invocation.

If an FLD to be (re)used specifies exactly one initial or final state, the entry or exit points for execution are uniquely defined such that the entry or exit compartments of the complex model operation can be omitted. This is illustrated by variant (b) in Figure 20. This variant is a complex model operation for the Self-repair-A FLD, which omits the explicit entry compartment since the FLD has exactly one initial state. In general, the entry (exit) compartments of a complex model operation have to be a subset of the initial (final) states of the corresponding FLD. However, all those final states that are reachable from the initial states selected for entry compartments must be selected for exit compartments. This ensures that the control flow can properly continue after an invocation.

Moreover, a complex model operation shows which runtime models have to be passed as parameters of the invocation to the invoked megamodel module. For the examples shown in Figure 20, it is the Architectural Model while the other runtime models used in the Self-repair-A module, namely the Failure analysis rules and Deep analysis rules, are encapsulated and provided by the invoked module itself.

Since we consider self-adaptive software that evolves throughout its lifetime, EUREMA adopts a dynamic typing approach in contrast to a static and explicit type system for FLDs and complex model operations. Thus, the signature of an FLD is not explicitly specified as it results from the current state of the megamodel module (*i. e.*, of the FLD instance) at runtime. In this context, the relevant state is the initial and final operations as well as

the runtime models that need to be passed as parameters. In EUREMA, the specification of a feedback loop in terms of the FLDs and the corresponding feedback loop instance in terms of the megamodel modules collapse since each feedback loop instance can be individually adapted. Hence, the initial and final operations as well as the runtime models can also be subject to adaptation, which accordingly modifies the signature to invoke the megamodel module. Consequently, a complex model operation must adhere to the signature determined by the current initial and final operations as well as the runtime models of the invoked module, which can be dynamically adapted and therefore not one-time statically defined. Nevertheless, the EUREMA interpreter checks at runtime whether a complex model operation conforms to the signature of the megamodel module to be invoked. If there is mismatch (*e. g.*, the entry compartment of the complex model operation cannot be mapped to an initial operation of the module to be invoked, or after the invocation, the final operation of the invoked module cannot be mapped to an exit compartment of the complex model operation), the EUREMA interpreter raises a runtime exception.

Concerning the concrete syntax, a complex operation is labeled with an icon, a small rounded rectangle. This icon distinguishes a complex model operation from a basic one in FLDs. Thus, the icon reveals that a model operation abstracts from another FLD and that it synchronously invokes an instance of this FLD at runtime when being executed. When using a complex model operation in an FLD, it must be given a name that declares the variable <var> (*cf.* Figure 20). The name must be unique within the FLD, in which the operation is used. Thus, the scope of the variable is the FLD and at runtime the megamodel module as the instance of this FLD. This variable is used later in the LD to bind the complex model operation to a concrete megamodel module that eventually should be invoked.

For example, to integrate the analyze fragment (see Figure 19) in the Self-repair feedback loop, engineers employ a complex model operation, in this case variant (a) of Figure 20, similar to a basic model operation. This results in the FLD shown in Figure 21. The used complex model operation is named Analyze, which declares a variable with the same name.

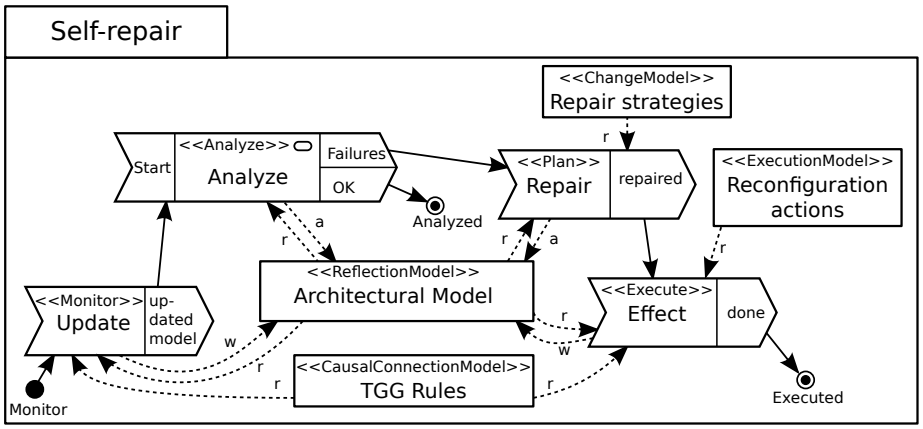


Figure 21: FLD Using the Self-repair-A FLD.

The binding of a complex model operation to a target for the invocation is specified in the LD, that is, at the instance level when the various megamodel modules (*i. e.*, FLD instances) are structured in the layered architecture. Specifically, such bindings are specified by use relationships between megamodel modules, which combine these modules and make the dependencies among them explicit. In general, a use relationship is reflected by a solid arrow and labeled with a variable. The corresponding variable has been declared

beforehand when defining and naming the complex model operation in an FLD. A use relationship with a variable and pointing from one megamodel module to another one binds the complex model operation, that declares this variable and that is contained in the former module, to the latter module. At runtime, this latter module is invoked when executing the complex model operation as part of the former module.

For the example, the corresponding LD with such a binding is shown in Figure 22. It describes that the :Self-repair module specified by the FLD in Figure 21 on the previous page senses and effects the :mRUBiS system and that it uses the :Self-repair-A module specified by the FLD in Figure 19 on Page 65. The use relationship is labeled with the variable Analyze that has been declared when employing and naming the complex model operation Analyze in the FLD of Figure 21. Thus, the use relationship binds this complex model operation contained in the invoking :Self-repair module to the :Self-repair-A module to be invoked.

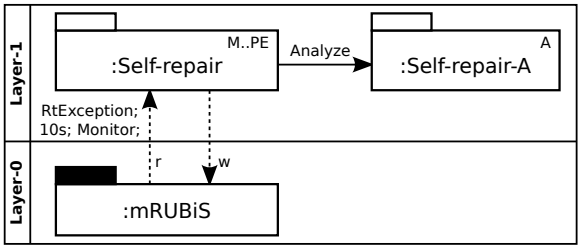


Figure 22: LD for the Self-repair and Self-repair-A Modules.

Additionally, the LD shows by the labels M..PE and A attached to megamodel modules that the :Self-repair module realizes the monitor, plan, and execute activities and the :Self-repair-A module the analyze activity. We use such labels to indicate which adaptation activities are realized by a megamodel module as defined in an FLD.

The same mechanism of binding a complex model operation to a megamodel module is used to bind basic model operations to their implementations. Such implementations are realized by engineers. They are black boxes for EUREMA and therefore modeled as software modules in LDs. An FLD specifies when a basic model operation should be executed, which runtime models are used as input and output, and the operation's return states. However, the concrete implementations of such operations have to be provided by software modules. When executing a basic operation as part of a megamodel module, EUREMA invokes the software module to which the operation is bound by a use relationship in the LD. Such a use relationship is labeled with the variable that has been declared when specifying the corresponding basic model operation in an FLD.

Such a binding is illustrated in Figure 23. The basic model operation Update of the :Self-repair feedback loop (*cf.* FLD in Figure 21 on the previous page) is bound to the software module :selfRepair.MonitorImpl by the use relationship labeled with Update. Thus, when the Update operation is executed, this software module is invoked with the runtime models that are specified as input of the operation in the FLD (*cf.* Figure 21).

Technically, software modules realizing basic model operations have to implement a specific interface prescribed by EUREMA. Such software modules and the interface will be discussed in detail in Chapters 7 and 8. The interface allows the EUREMA interpreter to invoke the software module, pass runtime models as parameters, and obtain the return status from the module after the invocation. Based on the return status, the interpreter selects the corresponding exit compartment of the executed operation to properly continue

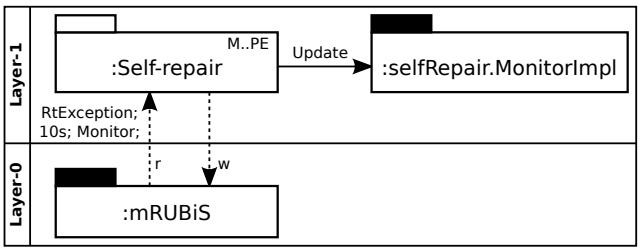


Figure 23: LD Illustrating the Binding of a Basic Model Operation to a Softare Module.

the control flow in the FLD instance and to execute the next operation. In the given example, the Update operation has only one return status and thus only one exit compartment.

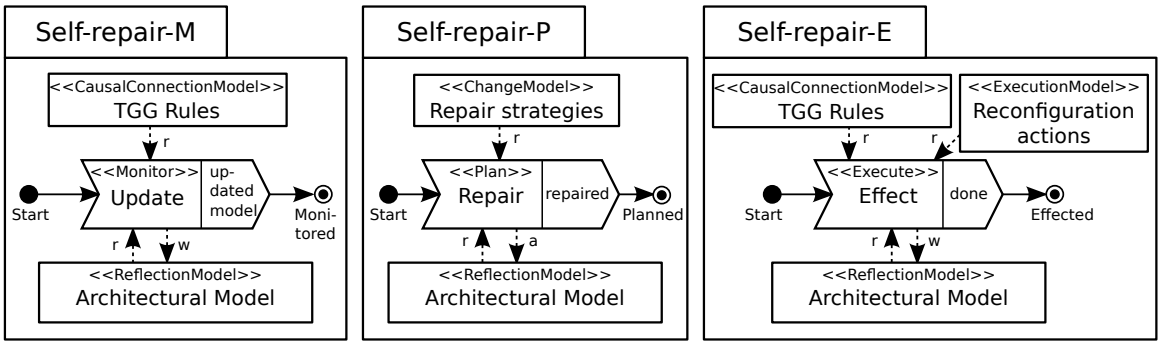
So far and in the following examples, we omit the modeling of software modules implementing basic model operations in LDs since they are just required for binding the operations to their implementations. This is relevant for the execution but they provide no further information relevant for the design of feedback loops. However, examples of such implementations will be discussed in detail in Section 9.3 when evaluating EUREMA.

In both cases, either binding a complex model operation to a megamodel module or a basic model operation to a software module, the corresponding use relationships combine modules to compose a feedback loop. Thus, such use relationships make the dependencies between the modules of a feedback loop explicit. Since modules use/invoke other modules, the former ones depend on the latter ones. Since a feedback loop is located in one layer and not spread across layers of the architecture, such use relationships that compose a feedback loop must not cross any layer in the LD. This constraint follows the principle of layered architectures that only adaptation relationships (*e. g.*, sense and effect) exist between modules at different layers. The EUREMA language assures this constraint by construction such that engineers cannot specify a feedback loop that is scattered across layers (*cf.* Section A.1).

Additionally, when composing a feedback loop by such use relationships, some modules do not require a triggering condition. Particularly, a megamodel module that is integrated as a fragment into a feedback loop and a software module that provides an implementation for a basic model operation do not need a triggering condition. In contrast, their execution is triggered by a call when executing the corresponding complex or basic model operation.

In general, concerning the modular specification of feedback loops, the two specifications of the self-repair feedback loop as the FLD of Figure 15 on Page 57 or the FLDs of Figure 19 on Page 65 and Figure 21 on Page 66 are equivalent considering functionality. The only difference is the number of FLDs used for the specification, which is motivated by design decisions concerning abstraction and modularity. For example, besides the analyze activity, each of the four MAPE activities can be specified in distinct FLDs and a high-level FLD integrates them to a feedback loop by using a complex model operation for each of them. This is illustrated in the following for the self-repair feedback loop.

While the analysis activity has already been specified in a distinct FLD (see Figure 19 on Page 65), the individual FLDs for the monitoring, planning, and executing activities are shown in Figure 24. These three activities have been cut from the previous FLD specifying the whole feedback loop (*cf.* Figure 21 on Page 66), copied in distinct FLDs, and enriched with appropriate initial and final operations. The high-level FLD that integrates the four MAPE activities using complex model operations is depicted in Figure 25. For each activity, a complex model operation is used that operates on the Architectural Model. This model is shared among all activities and therefore passed as a parameter to the individual activities.



(a) Monitor Activtiy. (b) Plan Activtiy. (c) Execute Activtiy.

Figure 24: FLDs for the Monitor, Plan, and Execute Activities of the Self-repair Feedback Loop.

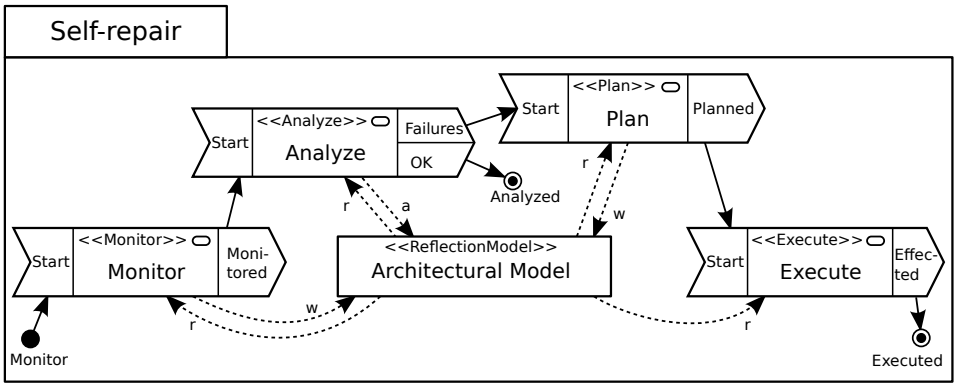


Figure 25: FLD of the Self-repair Feedback Loop Integrating the MAPE Activities.

Finally, the four complex model operations need to be bound to megamodel modules realizing the corresponding MAPE activities. This is done in the LD shown in Figure 26. Each FLD realizing one of the MAPE activities is instantiated to a megamodel module. Each of these modules is labeled either with M, A, P, or E to emphasize the activity it realizes. The complex model operations contained in the high-level :Self-repair module are bound to these modules by use relationships labeled with Monitor, Analyze, Plan, and Execute according to the specification of the operations in the FLD of Figure 25.

Considering this example, EUREMA does not restrict the depth of the abstraction for the FLDs and the related invocation relationships among the corresponding megamodel modules. This leverages different abstraction levels for modeling feedback loops and it further assists engineers in structuring and understanding feedback loops.

5.1.5 *Variability*

The modular specification of feedback loops discussed in the previous section enables identifying and modeling variability for feedback loops. Encapsulating feedback loops or individual adaptation activities defined by FLDs in megamodel modules and explicitly modeling such modules and their relationships in LDs reveals variation points in the adaptation engine. Such variability can be made visible in LDs and exploited to switch between variants, either during the design or dynamically at runtime.

In the example, we have specified the analyze fragment of the self-repair feedback loop in the dedicated FLD Self-repair-A and encapsulated it in the megamodel module :Self-repair-A

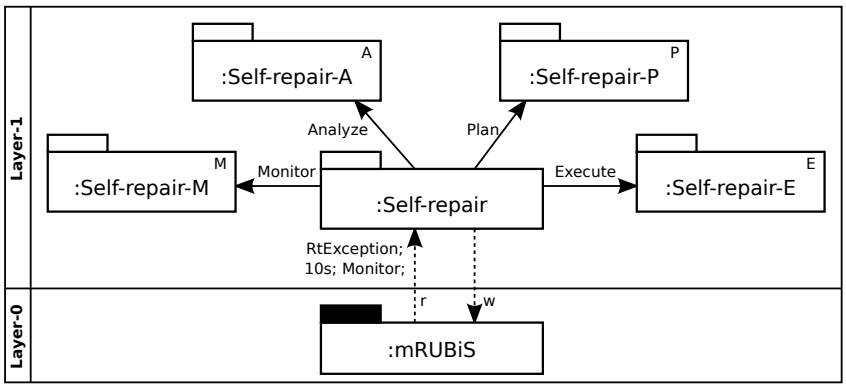


Figure 26: LD with Bindings of the MAPE Activities.

as discussed in the previous section. We now assume that we have modeled an additional analyze fragment, called Self-repair-A2, in a distinct FLD, which employs a different analysis technique than Self-repair-A. For instance, the techniques may differ in terms of runtime performance and accuracy of the analysis results. Both fragments are then alternative analyses that can be used by the self-repair feedback loop. This constitutes a variation point in the architectural design reflected in the LD in Figure 27.

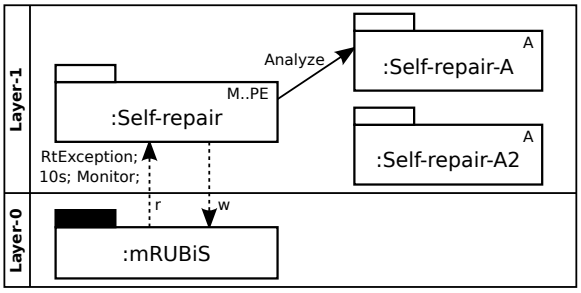


Figure 27: LD with Variability for Megamodel Modules.

If the FLDs of both alternatives have the same signature (*i. e.*, the same initial and final operations and the same runtime models as parameters), both of them can be used by the same complex model operation. In this case, to switch between these alternatives, it is sufficient to change the binding of the complex model operation to the invoked megamodel module, for instance, by re-routing the *use* relationship Analyze to point to the :Self-repair-A2 module instead of :Self-repair-A module in the LD in Figure 27. If the FLDs of both alternatives do not have the same signature, then the complex model operation within the invoking module has to be adapted to the signature of the actually used alternative. Since EUREMA follows a dynamic typing approach (see previous section), a complex model operation within a megamodel module can be replaced with another complex model operation that may have different entry and exit compartments as well as different runtime models as input and output. The EUREMA interpreter is able to handle such dynamic changes when executing the FLD instance containing the adapted complex model operation. How EUREMA supports dynamic changes will be discussed in detail in the context of stacked feedback loops and off-line adaptations in Sections 5.3 and 5.4.

In general, such variations points of megamodel modules reify architectural variability of feedback loops in LDs. This variability can be exploited to specify and evaluate alternative feedback loop designs or to adjust feedback loops at runtime by switching between variants.

Such variants correspond to arbitrary fragments of feedback loops that, however, contain usually at least one model operation as the behavior of an invoked fragment.

EUREMA applies the same idea to leverage variability for implementations of basic model operations. LDs capture alternative software modules implementing a basic operation and the binding of the operation to one alternative. Similar to changing a binding of a complex model operation to a megamodel module, the binding of a basic model operation to a software module providing the implementation can be changed. This is illustrated by the LD in Figure 28 showing that the Update operation of the :Self-repair feedback loop is bound to the software module :selfRepair.MonitorImpl while there is the alternative module :selfRepair.LightWeightMonitorImpl. By re-directing the use relationships named Update from one to the other alternative, the implementation of the Update operation will be exchanged.

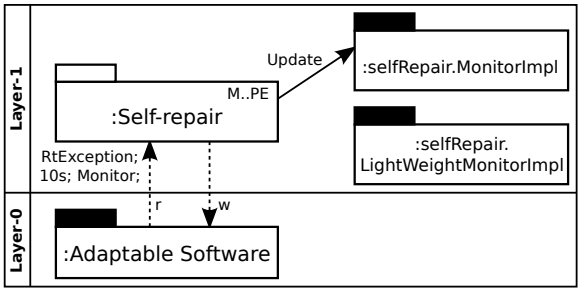


Figure 28: LD with Variability for Implementations of Basic Model Operations.

In general, such variations points at the level of megamodel modules specified by FLDs or software modules considered as black boxes reify variants for the design and implementation of feedback loops. Moreover, they can be leveraged at runtime to adjust feedback loops by switching between these variants such as between different analysis or monitoring techniques. Switching between such variants is eased in EUREMA since (complex) model operations should be stateless (*cf.* Section 6.7). The state in terms of the working data of operations is externalized and captured by runtime models used within the feedback loops. Consequently, if variants use the same runtime models—especially concerning the language in which these models are expressed—we can seamlessly switch between them. The replacing variant just starts execution with the same runtime models used by the replaced variant. In contrast, if the variants use different kinds of runtime models (*e. g.*, they are expressed in different languages), the models have to be translated. To achieve this kind of state translation, MDE techniques such as model transformation and synchronization (*cf.* Section 2.1.1) can be applied since the state is captured by models. Thus, before the replacing variant can start execution, the runtime models used by the replaced variant have to be initially transformed. Moreover, when variants that use different kinds of runtime models are simultaneously employed, these models have to be permanently synchronized to enable the integration of these variants. In general, the translation of state is eased in EUREMA by capturing the state in (runtime) models that are accessible by automated MDE techniques such as model transformation and synchronization.

Summing up, the modular specification of feedback loops supports the reuse of feedback loop fragments and implementations as well as the identification of variability for feedback loops. Such variability can be exploited by engineers during design to model variants of feedback loops, and at runtime to dynamically switch between the modeled variants.

5.2 MULTIPLE FEEDBACK LOOPS

Having discussed the modeling of a single feedback loop in Section 5.1, we now address the modeling of multiple feedback loops that are located in the same layer of the architecture. Employing multiple feedback loops, the relationships between them should be captured to identify potential interdependencies between them. Therefore, it should be made visible at the architectural level if multiple feedback loop operate independently from each other or in a coordinated manner. The coordination of multiple feedback loops is required if they address competing concerns and therefore compete with each other. Such interference between feedback loops are resolved by coordination.

In the following, we consider the mRUBiS example that employs two feedback loops, one for self-repairing failures and one for self-optimizing performance. Hence, two concerns are maintained by self-adaptation: The availability of mRUBiS that is affected by failures. The response time of mRUBiS that should be optimized. Each concern is handled by an individual feedback loop because each concern requires its specific runtime models and model operations. While the self-repair feedback loop has been discussed in Section 5.1, we now introduce the self-optimization feedback loop. The FLD specifying the Self-optimization feedback loop is shown in Figure 29. It analyzes the response time and the configuration of mRUBiS to identify performance issues. Such issues indicate the need for adapting the configuration to meet a certain target response time or to improve the response time. The feedback loop then resolves the identified issues by adapting the configuration of mRUBiS.

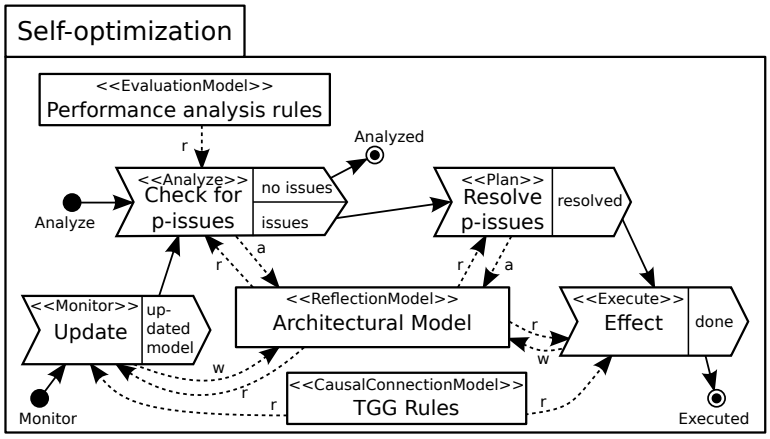


Figure 29: FLD for Self-optimization Feedback Loop.

Specifically, the feedback loop consists of four model operations. Similar to the self-repair feedback loop, the Update and Effect operations use triple graph grammar rules (TGG Rules) that specify by means of model transformation rules how the Architectural Model reflecting the running mRUBiS system is synchronized with the system. Thus, monitoring the system, the Update operation keeps the Architectural Model up-to-date. The Check for p-issues operation performs the analysis by applying Performance analysis rules on the Architectural Model. These rules define checks to identify occurrences of performance issues (*i. e.*, the response time is above or below a given threshold, and the current configuration of mRUBiS is not optimal with respect to performance). If no performance issues are found, the feedback loop terminates in the state Analyzed. Otherwise, the issues are annotated in the Architectural Model and adaptation is required to resolve them.

The planning activity[1] uses the performance issues annotated to the Architectural Model by the analyze operation to resolve them by selecting appropriate adaptation strategies (*e. g.*, removing optional components to improve performance). The selected strategies prescribing an adaptation of mRUBiS are annotated to the Architectural Model. The subsequent Effect operation executes the selected strategies and hence the prescribed adaptation using basic code-based actions such as adding, removing, or restructuring components in the Architectural Model. Moreover, it synchronizes the corresponding changes of the model to the running mRUBiS using the TGG Rules, which terminates one run of the feedback loop.

Finally, the self-optimization feedback loop has two initial states either initiating its execution with the monitor or the analyze activity. This will be exploited later when coordinating the self-repair and self-optimization feedback loops.

In general, EUREMA supports the specification of multiple feedback loops by distinct FLDs. In this context, employing multiple feedback loops raises questions of possible interferences between them. If there are no interferences, the two feedback loops are independent from each other and each of them may operate independently. However, if there are interferences between them, coordination between them is required to address these interferences. In the following, we discuss both cases, independent and (inter)dependent feedback loops, and how they are modeled with EUREMA to make the relationships between the feedback loops explicit in the design of self-adaptive software.

5.2.1 *Independent Feedback Loops*

If there are no interferences between multiple feedback loops, for instance, because the different concerns are not conflicting, the corresponding feedback loops can be specified and operated completely independent from each other. For instance, consider two feedback loops: One that maintains the performance of the adaptable software by adapting the software to the available bandwidth. And one that adjusts the software to country-specific settings (*e. g.*, multi-language support) by (de)activating existing components. Adapting the software to country-specific settings does not affect the bandwidth since these components already exist in the software. Otherwise, the adaptation would have to load dynamically the components, for instance, from a remote server, which would affect the bandwidth and therefore, interfere with the feedback loop managing the performance. On the other hand, adapting the software due to the available bandwidth does not affect the adaptation due to country-specific settings since for each of these settings a corresponding component must be activated and cannot be adapted to improve the performance. Consequently, both feedback loops do not affect each other such that they are independent of each other. Therefore, they can be specified and operated completely independent from each other.

Rather than modeling the independent feedback loops of this example, we stick to our running mRUBiS example and assume that the self-repair and self-optimization feedback loops are independent of each other in a similar manner—although this assumption will not hold in reality (*e. g.*, optimizing the performance of a failing system is not reasonable, or repairing a failure changes the architecture and configuration of the software, which impacts the performance). Nevertheless, we stick to the example and assumption to discuss the specification and operation of independent feedback loops in the following.

1 In contrast to the other operations, the planning activity is not specified and executed by a runtime model, particularly a change model (*cf.* Section 4.1) such as explicit adaptation strategies. It is a completely code-based operation to illustrate that EUREMA is able to integrate such operations and does not necessarily require that operations are specified and executed by runtime models.

We have already specified the self-repair and self-optimization feedback loops individually and independently from each other in distinct FLDs (*cf.* Figure 21 on Page 66 and Figure 29 on Page 72). To specify that these two feedback loops operate independently from each other, we use the LD that generally describes the relationships between feedback loop instances (*i. e.*, megamodel modules). The LD for the example is depicted in Figure 30. It shows that instances of the self-optimization and self-repair feedback loops are located at Layer-1 and both are sensing and effecting :mRUBiS located at Layer-0. Both instances have individual triggering conditions that might be activated at the same time such that they operate concurrently. While the triggering condition of the :Self-repair module has been discussed in Section 5.1.3, the :Self-optimization module should be triggered in its initial state Monitor when the load in :mRUBiS increases, causing an event of type LoadIncrease, and when 60 seconds since its last execution have elapsed. Based on the individual triggering conditions, both modules operate independently from each other and without any controlled interactions as these modules are not interrelated in the LD.

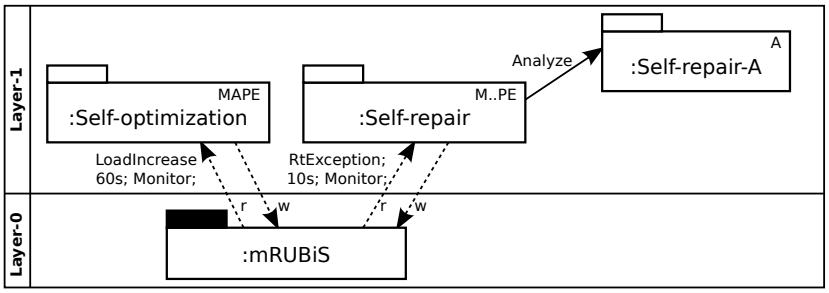


Figure 30: LD for Independent Feedback Loops.

However, it is conceivable that each of the two modules observes the adaptations performed to :mRUBiS by the other one. The :Self-repair and :Self-optimization modules individually maintain an Architectural Model reflecting :mRUBiS (*cf.* FLDs in Figure 21 on Page 66 and Figure 29 on Page 72). Monitoring the system and updating the architectural model, a feedback loop can observe the adaptation executed to the system by the other feedback loop, especially since both feedback loops perform architectural adaptations. However, we considers such observations of each other's adaptations to be implicit and irrelevant interactions since an adaptation performed by one of the feedback loops does not interfere with or even cause an adaptation performed by the other one. The reason for this consideration is that the two feedback loops are independent of each other. For the example, we assumed independence of the feedback loops while generally the independence must be ensured by engineers during development.

Compared to the LD in Figure 22 on Page 67 for the case of employing the single self-repair feedback loop, this LD just adds the :Self-optimization module in parallel to the :Self-repair module within the same layer. Consequently, the individual specification of independent feedback loops with FLDs continues when structuring instances of these feedback loops in an LD. However, running feedback loops completely independent of each other in EUREMA prevents any further coordination of them. For instance, they would have to be coordinated by a higher-layer feedback loop to adapt both of them safely (*cf.* Section 5.3). That is, the higher-layer feedback loop would have to intercept the execution of both feedback loops for adaptation, which is impeded due to uncoordinated and non-synchronized execution of these loops. Consequently, EUREMA does not support higher-layer feedback loops that control two or more independently running feedback loops.

In general, an LD explicitly shows if multiple feedback loop instances exists and if they operate independently from each other. This would not be visible by solely using FLDs since they focus on individual feedback loops and not on the whole adaptation engine.

5.2.2 *(Inter)Dependent Feedback Loops*

If we employ multiple feedback loops to address different concerns, there can be interferences between them that constitute dependencies or interdependencies between the feedback loops. This is particularly the case if the concerns are competing. In our running example, the concerns of availability (or failures) and response time are competing and causing likely interferences. On the one hand, optimizing the performance of a failing system is not reasonable because a subsequent repairing of the failures may contradict the optimizations. On the other hand, repairing a failure changes the architecture and configuration of the system, which may impact the performance (*e. g.*, a faulty component is replaced by an alternative component that consumes more resources than the faulty one). Such interferences cause (inter)dependencies and require coordination of the feedback loops.

In this context, we consider two aspects of coordination. The first aspect addresses the coordinated execution of the feedback loops in terms of synchronization to control concurrency. Such a coordination is required to avoid conflicts due to concurrency, for instance, when multiple feedback loops perform an adaptation at the same time. The second aspects covers reaching a consensus if conflicts occur, for instance, if contradicting adaptations have been planned and should be executed. A consensus determines a trade-off between the concerns and it can be reached by mechanisms from utility theory [156] (*e. g.*, utility functions to balance multiple concerns [115]) or coordination and agreement in distributed systems [126] (*e. g.*, voting to select adaptation actions for execution [369]). Such mechanisms typically require knowledge about the internals of the runtime models used in the feedback loops. For instance, to determine a trade-off among competing concerns requires knowledge about the current situation and goals of the concerns. Such knowledge is captured in the runtime models such as reflection models representing the current situation of the adaptable software and adaptation models operationalizing the goals (*cf.* Section 4.1).

Since the internals of runtime models are transparent for EUREMA, the mechanisms for achieving a consensus are not specified in EUREMA but they have to be realized by the implementations of the model operations and runtime models. Consequently, the mechanisms for achieving a consensus are also transparent for EUREMA. In contrast, EUREMA targets a higher abstraction level and addresses the interplay of the model operations and runtime models. Therefore, EUREMA focuses on the first aspect of coordination, that is, supporting the coordinated execution of multiple feedback loops.

In EUREMA, the coordination of the execution of multiple feedback loops is explicitly modeled with FLDs and LDs, the same language as used for specifying the individual feedback loops. At runtime, instances of these FLDs synchronize the operation of the (inter)dependent feedback loop instances while LDs make the coordinated execution explicit in the architectural design. In the following, we discuss two basic design alternatives for coordinating the execution of multiple feedback loops and we apply these alternatives to the mRUBiS example with the self-repair and self-optimization feedback loops. These two alternatives differ in the abstraction level at which coordination is addressed. The first alternative coordinates the execution at the level of complete feedback loops while the second alternative coordinates the execution at the level of individual adaptation activities.

Sequencing Complete Feedback Loops

A simple way to coordinate multiple feedback loops is to execute them sequentially, for instance, based on prioritizing one feedback loop over the other. Thus, the feedback loops do not interfere due to concurrency, which is ensured by EUREMA. However, if the feedback loops aim for competing goals and therefore permanently perform contradicting adaptations, their implementations have to provide a mechanism for achieving a consensus.

In EUREMA, the coordinated, sequential execution of multiple feedback loops is specified by FLDs and LDs. For the mRUBiS example, the sequential execution of the self-repair and self-optimization feedback loops is specified by an additional FLD shown in Figure 31.

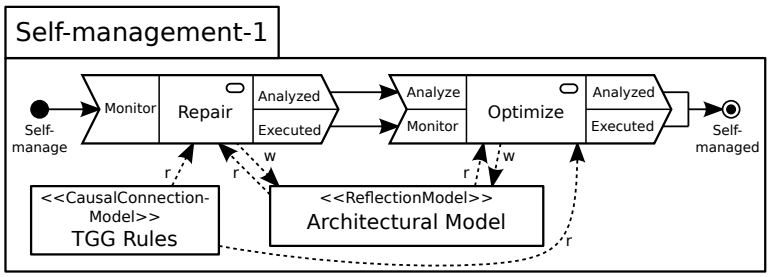


Figure 31: FLD for Sequencing Complete Feedback Loops.

We named this FLD Self-management-1 as it covers self-repair and self-optimization capabilities. It uses complex model operations to synchronously invoke the individual feedback loops. A higher priority is assigned to repairing failures than to optimizing performance since optimizing the performance of a failing system is only reasonable after the system has been repaired. Therefore, the self-repair is executed before the self-optimization loop.

Specifically, the complex model operations Repair and Optimize are bound respectively to the self-repair and self-optimization feedback loop instances, which is defined by the LD in Figure 32. Thus, the :Self-management-1 module invokes the :Self-repair and :Self-optimization modules one after the other. As a consequence, the invoked modules do not need any triggering condition in contrast to the :Self-management-1 module. This module is triggered if an event of the type RtException indicating potential failures or LoadIncrease indicating potential performance issues is emitted by :mRUBiS and when 35 seconds after the previous execution have elapsed. This triggering condition combines the individual triggering conditions of the self-repair and self-optimization feedback loops discussed before.[2] Taking over the coordination, the :Self-management-1 synchronizes the other modules in sensing and effecting :mRUBiS and when being triggered, it delegates to the other modules to perform the self-repair or self-optimization. Thus, the :Self-management-1 module itself does not perform any adaptation activity and has therefore no label in contrast to the labels MAPE for the :Self-optimization, M..PE for the :Self-repair, and A for the :Self-repair-A module.

The coordination of the self-repair and self-optimization feedback loops to sequentially execute them is specified in detail by the FLD (see Figure 31). The Repair operation invokes the self-repair feedback loop, that is specified by the FLD in Figure 21 on Page 66, to start execution in the initial state Monitor. Thus, the monitor and analyze activities of this feedback loop are carried out to update and check the Architectural Model for failures.

2 Coordinating multiple feedback loops with different periods (*i.e.*, execution frequencies) generally requires aligning the feedback loops to a common period. Accordingly, EUREMA requires that coordinated feedback loops have a common trigger attached to the coordinating megamodel module. Such a trigger may result from combining individual triggers, that is, by identifying a common period and merging individual events.

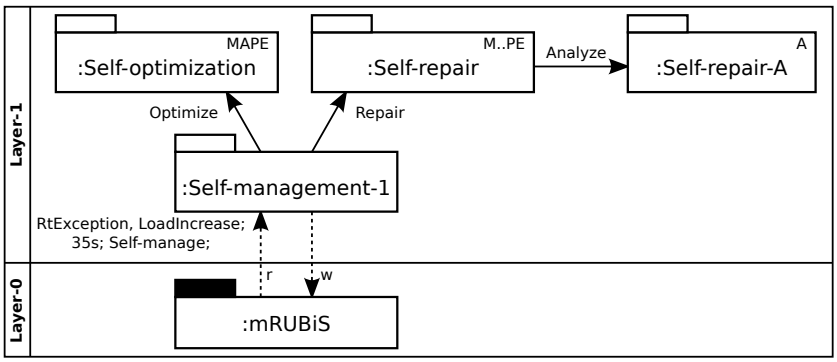


Figure 32: LD for Sequencing Complete Feedback Loops.

Depending on whether failures have been identified or not, the self-repair feedback loop either continues with the following adaptation activities or it terminates, respectively. This influences the subsequent execution of the self-optimization feedback loop.

If no failures are identified, the self-repair feedback loop does not plan and execute any adaptation and terminates in the final state Analyzed. In this case, the subsequent self-optimization feedback loop may immediately start with the analyze activity because the self-repair's monitor activity has already updated the shared Architectural Model and no adaptation of this model and of mRUBiS has been performed to repair any failures. As shown in the FLD in Figure 31, the Architectural Model and the TGG Rules are passed as parameters to both feedback loops when instances of these loops are invoked at runtime. Thus, both feedback loops share the same model and use the same rules for the monitor and execute activities to synchronize the model and mRUBiS in the same way.[3] Thus, the Optimize operation invokes the self-optimization feedback loop, that is specified by the FLD in Figure 29 on Page 72, to begin execution in the initial state Analyze (*cf.* the control flow connecting the exit compartment Analyzed of the Repair operation to the entry compartment Analyze of the Optimize operation in Figure 31). If no performance issues have been identified, the self-optimization feedback loop terminates in the final state Analyzed. Otherwise, it performs the plan and execute activities, and terminates in the final state Executed.

On the other hand, if the self-repair feedback loop identifies failures, it plans an adaptation of the mRUBiS architecture, which repairs these failures, executes this adaptation to mRUBiS, and terminates in the final state Executed. This requires that the subsequent self-optimization feedback loop performs monitoring to observe the effects of this adaptation, for instance, to obtain the mRUBiS architecture after the repair. Therefore, Optimize invokes the self-optimization feedback loop to begin execution in the initial state Monitor (*cf.* the control flow connecting the exit compartment Executed of the Repair operation to the entry compartment Monitor of the Optimize operation in Figure 31). After carrying out the monitor and analyze activities, the self-optimization loop either terminates or, if performance issues have been identified, plans and executes an adaptation addressing these issues.

3 The synchronization of a reflection model (*e. g.*, the Architectural Model) and an adaptable software (*e. g.*, mRUBiS) is the causal connection problem (*cf.* Section 2.1.5), that is, propagating changes from the software to the model by the monitor activity and vice versa by the execute activity. This problem can be generically solved (*cf.* Section 8.2) as we consider generic changes at the architectural level (*e. g.*, creating and deleting components and changing their interconnections and parameters) [258, 292]. Thus, changes of the software that are monitored or adaptations that are planned can be generically described as such architectural changes and used for the causal connection regardless of the addressed self-adaptation capability such as self-repair or self-optimization.

This coordination design synchronizes different feedback loops by sequentially executing them based on priorities and by using the adaptable software for synchronization. Thus, an adaptation performed by one feedback loop is executed to the software before another loop starts execution with the monitor activity to observe the effects of the adaptation. However, if a feedback loop does not perform any adaptation of the reflection model and adaptable software, the subsequent loop may skip the monitoring and start with the analyze activity since the previous loop already performed the monitoring to update the shared reflection model. Thus, explicitly coordinating multiple feedback loops makes visible which runtime models are shared by them and allows feedback loops to benefit from other loops, for instance, by reusing instead of recomputing the monitoring results. In EUREMA, such a coordination is explicitly specified by FLDs defining how the execution of the megamodel modules is coordinated and by LDs describing how the coordinating and the coordinated modules are related to each other at the architectural level.

For the given example, the implementation of the feedback loops does not have to provide any mechanisms for achieving a consensus since there are only basic interferences that are resolved by prioritizing and sequencing the feedback loops. On the one hand, the self-repair feedback loop only repairs the broken system, which may affect the performance but which is also a prerequisite for optimizing the system and its performance. On the other hand, we assume that the self-optimization feedback loop does not cause any failures in the system, which would otherwise require adaptations to be performed by the self-repair feedback loop. Consequently, there exists only a one-way dependency, that is, the self-optimization feedback loop depends on the self-repair feedback loop. Such one-way dependencies can be addressed by ordering the feedback loops accordingly.

Sequencing Adaptation Activities of Feedback Loops

The other design alternative for coordinating the execution of multiple feedback loops is more fine-grained than completely sequencing them. It intertwines the feedback loops by synchronizing them in shared monitor and execute activities and sequentially executing their individual analyze and plan activities. Modeling this coordination with FLDs and LDs, EUREMA ensures that the activities of the different feedback loops do not interfere due to concurrency. However, the implementations of the analyze and plan activities of the different feedback loops have to achieve a consensus since subsequent activities might contradict the analysis and planning results produced by the preceding activities. Both aspects will be discussed in the following using the running mRUBiS example.

In this example, the coordinated execution of the self-repair and self-optimization feedback loops at the level of adaptation activities is specified by the FLD in Figure 33.

The Update and Effect operations that perform respectively the monitor and execute activities synchronize the Architectural Model with the running mRUBiS by using the TGG Rules. This Architectural Model is shared by the individual, concern-specific analyze and plan activities of the self-repair and self-optimization feedback loops (*cf.* complex model operations RepairAP and OptimizeAP) that are specified in individual FLDs shown in Figure 34. These FLDs just describe the analyze and plan fragments of the self-repair and self-optimization feedback loops (*cf.* Figure 15 on Page 57 and Figure 29 on Page 72).

Thus, the analysis and planning for the self-repair are executed before the analysis and planning for the self-optimization. The Architectural Model is only modified by the self-repair's plan activity if the related analyze activity has identified failures. These modifications are a planned adaptation of the Architectural Model and of the running mRUBiS to repair the failures, that is, the adaptation has not been effected yet in the model and

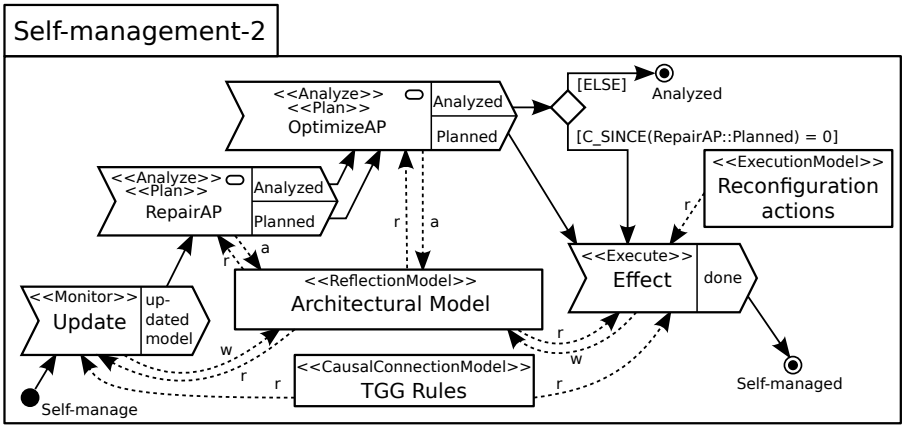


Figure 33: FLD for Sequencing Individual Adaptation Activities of Feedback Loops.

system. If no adaptation for self-repair has been planned, there are no conflicting adaptations possible when an adaptation is subsequently planned for self-optimization. Otherwise, the adaptations proposed for the self-repair must be handled by the subsequent analyze and plan activities for the self-optimization, which requires achieving a consensus. In this case, the adaptation proposed for the self-repair is considered as an invariant for the self-optimization, that is, the implementations of the analyze and plan activities for the self-optimization take potential adaptations planned in the Architectural Model for the self-repair into account and do not plan any conflicting adaptations. For instance, if a faulty component in mRUBiS should be exchanged by an alternative component to repair the system, the analyze and plan activities for the self-optimization will not prevent or undo this planned replacement although the replacing component might consume more resources than the replaced one and hence decrease the overall performance of the system.

Considering the Self-management-2 FLD in Figure 33, when the self-optimization's analyze and plan activities terminate, the Effect operation is executed if adaptations are proposed in the Architectural Model by the self-repair's or the self-optimization's plan activities. Thus, at least one of the complex model operations RepairAP or OptimizationAP must terminate in the state Planned. In this case, the Effect operation executes the planned adaptations in the Architectural Model and synchronizes them to the running mRUBiS. Otherwise, the Self-management-2 module terminates in the state Analyzed because no failures and no performance issues have been identified, which does not require any adaptation to be planned and thus to be executed in the Architectural Model and to mRUBiS.

Similar to the coordination mechanism discussed previously, we use an LD to bind the complex model operations to the invoked megamodel modules and to make all modules with their coordinated execution visible at the architectural level. The corresponding LD is shown in Figure 35. The :Self-management-2 module senses and effects :mRUBiS and it delegates to the :Self-repair-AP and :Self-optimization-AP modules when the analysis and planning steps for the self-repair and self-optimization should be performed. Therefore, the complex model operations RepairAP and OptimizeAP are respectively bound to the :Self-repair-AP and :Self-optimization-AP modules (*cf.* use relationships from the :Self-management-2 to the :Self-repair-AP and :Self-optimization-AP modules in the LD).

Similar to the :Self-management-1 module discussed previously in the context of the other coordination design, the :Self-management-2 module coordinates the other modules such that it requires a triggering condition in contrast to the coordinated, invoked modules. For this design, the same triggering condition as for the :Self-management-1 module is used.

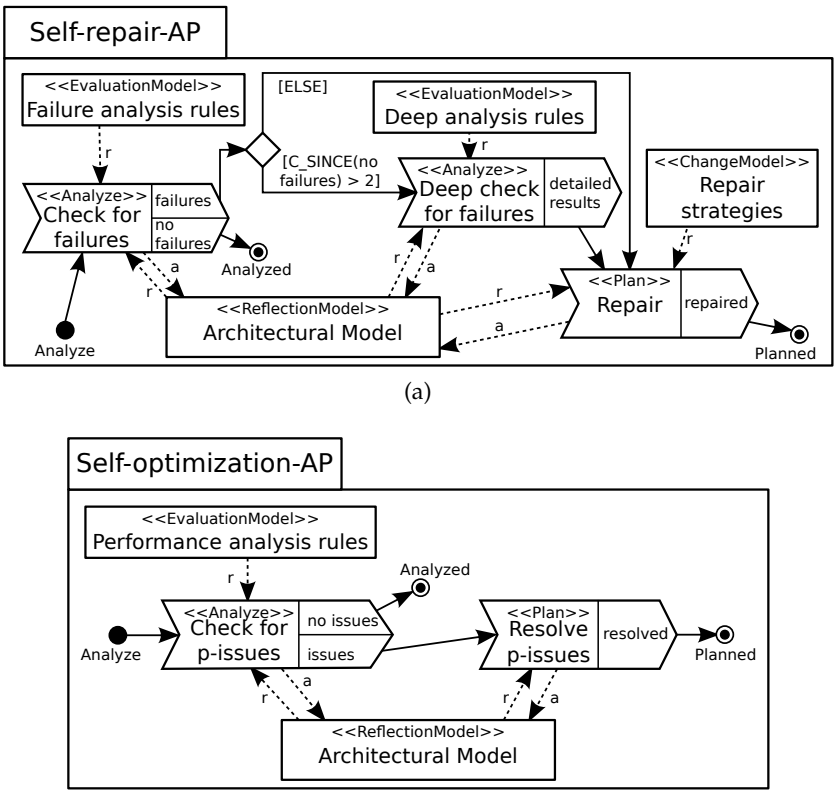


Figure 34: FLDs for the Analyze and Plan Activities of the (a) Self-repair and (b) Self-optimization.

However, in contrast to the other coordination design, the coordinating module does not only delegate to other modules but also performs itself adaptation activities. The :Self-management-2 module performs the monitor and execute activities such that it is labeled with M..E, and it delegates to the :Self-repair-AP and :Self-optimization-AP modules that realize the analysis and planning activities such that they are labeled with AP.

This example illustrates that coordinating multiple feedback loops at the level of adaptation activities supports sharing some activities such as monitoring and execution while having specialized activities such as analyze and plan for the specific concerns. In this context, EUREMA supports the coordinated execution of multiple feedback loops whose activities are essentially combined to one feedback loop. However, such a combination likely requires achieving a consensus among the activities of different feedback loops, for

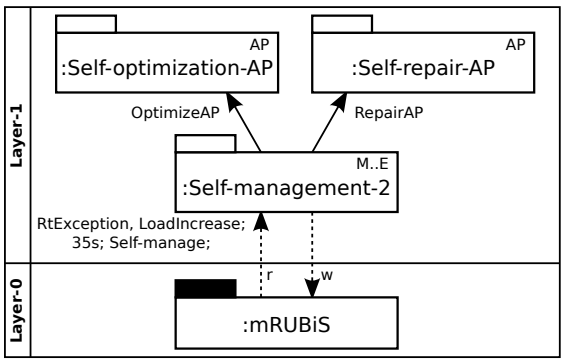


Figure 35: LD for Sequencing Adaptation Activities of Individual Feedback Loops.

instance, to avoid the planning of conflicting adaptations. This kind of coordination has to be addressed by the implementations of the activities since a consensus usually depends on internals of the runtime models and operations that are transparent for EUREMA.

Summing up, the coordination of multiple feedback loops can be specified with the same modeling concepts of EUREMA as used for specifying individual feedback loops. Thus, engineers do not have to learn new concepts to model the coordination. While FLDs are used to define how the coordination is done, LDs make the coordination visible in the architectural design. In this context, EUREMA focuses on the coordinated execution of multiple feedback loops either by synchronization at the level of complete feedback loops or at the level of individual adaptation activities of the different feedback loops. Explicitly specifying the synchronization, EUREMA enables the coordinated execution of multiple feedback loops in a controlled manner, which is eventually realized by the EUREMA interpreter.

Moreover, the explicit specification enables the sharing of reflection models (*e.g.*, the Architectural Model) or adaptation activities (*e.g.*, the monitor or execute activity) among multiple feedback loops. This potentially reduces the computation efforts at runtime. For instance, by sharing the monitor and execute activities as well as the reflection model, the causal connection among the reflection model and the adaptable software is maintained for all feedback loops rather than maintaining it redundantly for each feedback loop.

5.3 LAYERED FEEDBACK LOOPS

So far we have discussed the modeling of single and multiple feedback loops that are located within the lowest layer of the adaptation engine and that directly control the adaptable software. In this section, we discuss feedback loops that are located in different and higher layers of the adaptation engine and therefore, they operate on top of each other. That is, we stack feedback loops in a layered architecture. Such layered feedback loops are used to realize adaptive control architectures, in which a higher-layer feedback loop dynamically adapts a feedback loop at the layer directly below to dynamically adjust the adaptation logic. Though being adapted by a higher-layer feedback loop, a lower-layer feedback loop should not be aware of and dependent on the higher-layer loop. Thus, a lower-layer feedback loop should be able to operate without any higher-layer loop.

Similar to the adaptable software, a feedback loop should be generally adaptable by means of its parameters or structure. While for parameter adaptation it is often sufficient that a higher-layer feedback loop observes and adapts individual variables of the lower-layer feedback loop, structural adaptation requires architectural views. Such views can be provided by reflection models representing adaptable feedback loops. In the following, we discuss how layered feedback loops are modeled in EUREMA and how reflection models and the adaptation of feedback loops are supported by EUREMA.

Feedback loops developed with EUREMA are adaptable by construction because the EUREMA models as feedback loop specifications and visualized by FLDs are kept alive at runtime and they are directly executed by an interpreter. Particularly, the specification and the runtime instance of a feedback loop collapse such that the same model is used to specify, execute, and adapt the feedback loop. The EUREMA interpreter is able to cope with dynamic changes of EUREMA models, even while executing these models. EUREMA supports dynamic adaptation of feedback loops with respect to the FLD elements, namely, the runtime models, model operations, control flow, and usage of runtime models.

Thus, EUREMA supports dynamically adjusting the runtime models used within a feedback loop, for instance, to replace the evaluation and change models that define the plan

and analyze activities. Likewise, model operations can be adjusted by adding, removing, or replacing them. This typically requires further adaptation of the usage of runtime models and the control flow. Besides such structural adaptations, the control flow can be adjusted by parameter adaptation when decision nodes are used. Such nodes enable the conditional branching of the control flow, which can be adapted by changing the conditions of individual branches. Despite these various options to dynamically adapt a feedback loop, adjusting the runtime models is often sufficient since they serve as executable behavioral specifications for model operations and they contain the working data of the operations (*cf.* Section 4.2). Hence, the behavior of individual model operations and of the feedback loop can be easily adapted by changing the runtime models used within the loop.

Such adaptations of feedback loops are conducted by other feedback loops operating at higher layers. In EUREMA, the modeling of higher-layer feedback loops is similar to the modeling of feedback loops that are located at the lowest layer of the adaptation engine and that control the adaptable software. However, a particular aspect is the creation and maintenance of reflective views of adaptable feedback loops, which are used by higher-layer feedback loops as a basis for adaptation. Such views can be realized by reflection models representing adaptable feedback loops. To discuss such reflection models and the modeling of layered feedback loops with EUREMA, we use the following example.

The self-repair feedback loop applies a set of repair strategies to handle failures at runtime that occur in mRUBiS (*cf.* FLD in Figure 21 on Page 66). This set covers alternative strategies to repair failures. For instance, a component causing failures can be restarted, redeployed, reconfigured, or replaced with a another component providing the same functionality. Thus, a repertoire of repair strategies is available and a decision must be made which strategies should be applied. Each strategy has different characteristics, for instance, in terms of effectiveness and costs. Replacing a component can be more effective but also more costly than just restarting it. To manage the set of strategies and to decide which strategies should be applied by the self-repair feedback loop, we employ another feedback loop operating on a higher layer and on top of the self-repair loop. It analyzes the success of the applied repair strategies and if needed, it selects other strategies from the repertoire, which replace the currently applied strategies in the self-repair feedback loop. This constitutes an adaptation of the self-repair feedback loop by the higher-layer feedback loop. A similar scenario is conceivable for the analysis rules used by the self-repair feedback loop to identify failures. These rules might have to adapted, for instance, to improve the accuracy of the failure identification, which can be conducted by a higher-layer loop.

This example follows the reference model of Kramer and Magee [258] (*cf.* Section 2.2.3) that considers a higher-layer feedback loop that manages the adaptation plans (*i. e.*, strategies) of a lower-layer feedback loop. Using this example, we discuss in the following two variants of reflecting on feedback loops, which are supported by EUREMA. These variants correspond to the idea of *procedural* and *declarative* computational reflection [285] (*cf.* Section 2.1.5). In procedural reflection, the EUREMA models that specify and execute feedback loops are directly used as reflection models of the feedback loops for adaptation. In contrast, declarative reflection employs user-defined reflection models of feedback loops that are separated from the EUREMA models that specify and execute the feedback loops.

5.3.1 *Procedural Reflection: EUREMA Models as Reflection Models*

In procedural reflection, the program of a system implements the domain functionality and serves directly as the representation of itself for adapting it (*cf.* Section 2.1.5). Ap-

plying this idea to EUREMA means that the EUREMA models specifying and executing feedback loops are directly used as reflection models of the feedback loops for adaptation. Thus, a higher-layer feedback loop dynamically adjusts the EUREMA models as visualized by FLDs that specify and execute a lower-layer feedback loop to adapt this lower-layer loop. This approach to reflection is feasible since EUREMA models are kept alive at runtime and executed by an interpreter that can cope with dynamic changes of the models. Consequently, higher-layer feedback loops do not maintain a separate representation of a lower-layer feedback loop but one and the same representation in terms of the EUREMA model is used for the specification, execution, and adaptation of the lower-layer loop.

For the example, the higher-layer feedback loop that exploits procedural reflection to adapt the self-repair feedback loop is defined by the FLD in Figure 36. This feedback loop is called Select-repair-strategies. It is in charge of selecting repair strategies from a repertoire and providing them to the self-repair feedback loop by means of adaptation.

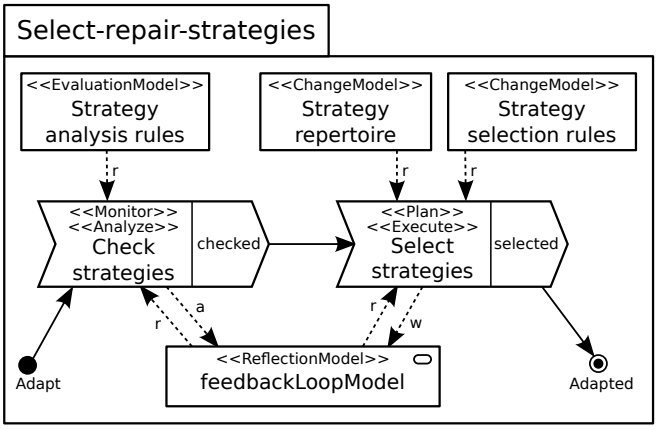


Figure 36: FLD for the Higher-Layer Feedback Loop with Procedural Reflection.

Therefore, the Select-repair-strategies feedback loop uses the runtime model, particularly, the reflection model called feedbackLoopModel. This model is labeled with an icon to highlight that it is an *FLD model*. Thus, this model is an EUREMA model as visualized by an FLD, which is directly used as a runtime model by the feedback loop. In the example, this model is the FLD instance specifying and executing the self-repair feedback loop (*cf.* Figure 21 on Page 66). The Select-repair-strategies feedback loop uses this FLD instance to check the success rate of the currently applied repair strategies. Particularly, the Check strategies operation identifies which strategies are currently used by the self-repair feedback loop and which failures have not been treated and thus still exist in mRUBiS.

Having access to the FLD instance specifying and executing the self-repair feedback loop, the Select-repair-strategies feedback loop may also access the runtime models used within the self-repair loop, especially, the Repair strategies and the Architectural Model to identify the currently used strategies and the untreated failures. Remember that the identified failures are annotated to the Architectural Model. If such failures exist, these strategies may not be able to cope with these failures such that they should be replaced. Accordingly, the Check strategies operation annotates the Repair strategies in the feedbackLoopModel. This basic analysis is specified by executable Strategy analysis rules.

The subsequent Select strategies operation identifies and replaces the annotated strategies directly in the feedbackLoopModel (*i. e.*, in the FLD instance specifying and executing the self-repair feedback loop) with alternative strategies from the Strategy repertoire. This

replacement is defined by Strategy selection rules that simply choose alternative strategies from the repertoire that should be tried out by the self-repair feedback loop to tackle the untreated failures.[4] Thus, the Select strategies operation directly changes the FLD instance specifying and executing the self-repair feedback loop by exchanging the runtime model defining the Repair strategies. The runtime model defining the currently used strategies is removed and the runtime model defining the new strategies is injected. Having replaced the strategies, the higher-layer feedback loop terminates and the self-repair feedback loop immediately uses the new strategies when continuing its execution.

Applying procedural reflection, the causal connection between the reflection model and the feedback loop under adaptation is ensured by construction. Thus, there is no need for explicit monitor and execute activities in a higher-layer feedback loop such as the Select-repair-strategies feedback loop whose monitor and execute activities are implicitly realized by the analyze and plan activities (*cf.* related stereotypes in the FLD in Figure 36).

To achieve procedural reflection, the FLD model reflecting the lower-layer feedback loop has to be bound at runtime to the specific megamodel module that encapsulates the FLD instance specifying and executing this feedback loop. This binding is defined by the LD in Figure 37 showing the :Select-repair-strategies module at Layer-2 that senses and effects the :Self-repair module at Layer-1. The binding of the FLD model, that is called feedbackLoopModel and contained in the :Select-repair-strategies module, to the :Self-repair module is defined by the use relationship having the same name as the model and pointing from the :Select-repair-strategies to the :Self-repair module. The mechanism for binding an FLD model to a megamodel module is similar to the binding of a complex model operation to the invoked megamodel module and of a model operation to the software module providing an implementation (*cf.* Section 5.1.4). Hence, when specifying an FLD model as part of a higher-layer feedback loop to reflect on a lower-layer feedback loop, the name of this model declares a variable. This variable is used in the LD to bind the model to the specific megamodel module specifying and executing the lower-layer feedback loop. In the example, the FLD model is called feedbackLoopModel (*cf.* Figure 36), which declares a variable that is used in the LD (*cf.* Figure 37) to bind the model to the appropriate module.

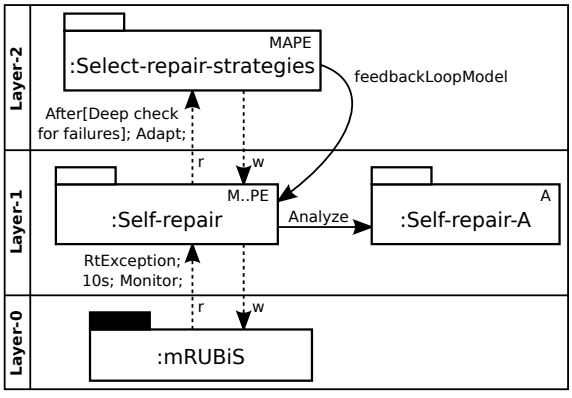


Figure 37: LD with Layered Feedback Loops and Procedural Reflection.

4 Since the focus of this section is on modeling layered feedback loops with EUREMA, we consider basic analysis and planning mechanisms for the higher-layer feedback loop, which simply select existing repair strategies in a trial and error fashion. In general, the analysis and planning mechanisms are out of scope of EUREMA since they are realized by implementations of model operations and runtime models. Consequently, from the EUREMA's point of view, advanced mechanisms such as dynamically synthesizing new repair strategies are conceivable. However, we do not elaborate such mechanisms or claim any contribution concerning them.

We name runtime models that are bound to megamodel modules *FLD models* as they refer to feedback loops specified by FLDs. When modeling a higher-layer feedback loop (*e. g.*, the Select-repair-strategies), engineers specify an FLD model (*e. g.*, the feedbackLoopModel) that materializes at runtime to an instance of the FLD specifying and executing the lower-layer feedback loop (*e. g.*, the Self-repair FLD shown in Figure 21 on Page 66). In general, an FLD model can be used for arbitrary purposes by a feedback loop although it is typically used as a reflection model to represent and control a lower-layer feedback loop.

To control and adapt the self-repair feedback loop, the :Select-repair-strategies module operates on the :Self-repair module (*i. e.*, on an instance of the FLD in Figure 21 on Page 66). Thereby, it also operates on the :Self-repair-A module (*i. e.*, on an instance of the FLD in Figure 19 on Page 65) because this module is used and therefore included by the :Self-repair module. Thus, reflecting on a megamodel module includes the reflection on other modules used by the megamodel module. In general, to dynamically adapt a lower-layer feedback loop, a higher-layer feedback loop changes the FLD instance that specifies and executes the lower-layer loop. Thus, all FLDs shown in this thesis are *initial* specifications of feedback loops since they are kept alive at runtime, they directly serve as the runtime instance for execution, and they might be dynamically changed at runtime.

The triggering conditions of higher-layer feedback loops are quite similar to the conditions for feedback loops at the lowest layer of the adaptation engine that are directly controlling the adaptable software (*cf.* Section 5.1.3). They refer to events emitted by the sensed module and define the initial state of the higher-layer feedback loop in which the execution should start. Here, the sensed module is the lower-layer feedback loop specified and executed by EUREMA. In general, events are synchronously emitted by the EUREMA interpreter when executing a feedback loop in terms of an FLD instance. There exists three types of events:(1) Before[opName] events are emitted *before* any operation is executed. (2) Similarly, After[opName] events are emitted *after* any operation is executed. Thereby, opName refers to the name of the specific operation that is executed. Finally, (3) OnTransition[tName] events are emitted when a transition in terms of a control flow link between two operations is executed. Thereby, tName refers to the name of the operation's exit compartment that is the unique source end of the executed transition. In contrast to operations, there is no behavior associated to transitions. Therefore, we do not have to distinguish between the state before and after executing a transition and we consider only the OnTransition event type.

All events of these three types are emitted synchronously, which supports intercepting the execution of feedback loops. Particularly, EUREMA supports intercepting the execution of a (lower-layer) feedback loop to synchronously execute another (higher-layer) feedback loop. Consequently, feedback loops that operate on top of each other do not run concurrently to avoid conflicts such as the higher-layer feedback loop adapts the repair strategies of the lower-layer loop while this lower-layer loop currently applies these strategies.

To ensure that layered feedback loops do not run concurrently, triggering conditions of higher-layer feedback loops must not use the period concept to influence the frequency of execution. The reason is that the whole adaptation engine would be blocked if lower-layer loops are intercepted and higher-layer loops are delayed until the period has elapsed. This distinguishes triggering conditions for feedback loops at the lowest layer of the adaptation engine from conditions for higher-layer feedback loops since the adaptable software emits events asynchronously while the EUREMA feedback loops synchronously. In other words, the adaptable software is not intercepted and blocked to execute a feedback loop at the lowest layer of the adaptation engine while any EUREMA feedback loop is intercepted and blocked to execute a feedback loop at the next higher layer.

Therefore, a triggering condition of a higher-layer feedback loop consists of only two parts: <events>; <initialState>;. The first part is a list of events of the types Before, After, or OnTransition that define the interception points. The individual events of this list are combined by logical disjunction such that the trigger is activated when the lower-layer feedback loop reaches one of the interception points defined by the list of events. Thus, the list must contain at least one event. The second part of the triggering condition defines the initial state in which the triggered higher-layer feedback loop will start execution.

Triggering a higher-layer feedback loop by intercepting the lower-layer feedback loop avoids inconsistencies due to concurrency. Particularly, intercepting a feedback loop at the given execution states (*i. e.*, before or after executing an operation or when executing a transition) enables a safe adaptation similar to a quiescent state (*cf.* [259]). On the one hand, a feedback loop is not intercepted and adapted while it is executing an operation and thus any behavior. An operation works on runtime models and it may modify them. Therefore, when intercepting a model operation during its execution, the runtime models might be in an inconsistent state since the operation has not finished its work yet. Consequently, we consider an operation as an atomic unit of execution. This consideration and the fact that operations should be stateless (*i. e.*, their state is externalized to the runtime models) ensures that the runtime models are in a consistent state *after* an operation has been executed and *before* the next operation will be executed (*cf.* After and Before event types). This state between executing two operations is the execution of a transition connecting these two operations. Since transitions do any have any associated behavior, we may also intercept the execution of a transition (*cf.* OnTransition event type) for safe adaptations. On the other hand, the execution of a feedback loop in EUREMA is the sequential, non-concurrent execution of operations. Consequently, when intercepting a feedback loop between the execution of two operations, the feedback loop does not have any active behavior, that is, no operation is running. This means that the feedback loop is quiescent. Thus, ensuring consistent runtime models and intercepting a (lower-layer) feedback loop allows the higher-layer feedback loop to safely adapt the lower-layer feedback loop.

In this context, the specific interception point that is used to adapt a feedback loop can be selected by engineers. With the triggering condition of the higher-layer feedback loop, engineers specify when the lower-layer feedback loop should be intercepted and therefore adapted. This allows engineers to take into account dependencies between operations such as two consecutive operations use the same runtime model while both of these operations should either use the original or the adapted version of the runtime model (*cf.* version consistency [284]). Hence, the adaptation should take place either before or after the execution of both operations but not in between the execution of both. Engineers may respect such constraints by specifying corresponding triggering conditions.

The adaptation of a lower-layer feedback loop is only restricted concerning the interception point. A higher-layer loop must not remove the interception point of the lower-layer loop as part of an adaptation. Otherwise, the EUREMA interpreter looses the position to which the program counter[5] of the lower-layer loop points and it cannot continue the execution of this loop after the execution of the higher-layer loop. The interception point of a feedback loop is either before or after executing a specific operation or while executing a specific transition. When intercepting a feedback loop, the program counter points to the specific operation or transition of the interception point. Thus, an adaptation must not re-

5 For each feedback loop instance, the EUREMA interpreter maintains a program counter pointing to the currently executed operation or transition. The EUREMA language reflects this counter as the relationship current pointing from the instance's execution context to the currently executed operation or transition (*cf.* Section A.1).

move this operation or transition. A detailed discussion of safely adapting feedback loops at runtime is given in Section 6.6 in the context of the EUREMA execution semantics.

For the example, the triggering condition of the :Select-repair-strategies module (see Figure 37) refers to the event After[Deep check for failures] that is emitted when the EUREMA interpreter executes the :Self-repair and :Self-repair-A modules, particularly, after having executed the Deep check for failures operation (*cf.* FLD in Figure 19 on Page 65). This is the case if more than two consecutive runs of the self-repair feedback loop has identified failures in mRUBiS, which indicates the need for new or other repair strategies. At this point, the execution of the :Self-repair and :Self-repair-A modules are intercepted and the higher-layer :Select-repair-strategies module synchronously starts execution in its initial state Adapt. This higher-layer module adapts the repair strategies used by the :Self-repair module. After it has finished execution, the :Self-repair module continues execution at the point it has been intercepted and starts now using the adapted strategies for the planning. Consequently, the interception point has been selected to be after the analysis but before the planning activities of the self-repair feedback loop. In a run of the feedback loop, the repair strategies are only used by the planning activity that will now use initially the adapted strategies.

Using the interception point After[Deep check for failures], the :Select-repair-strategies module must not remove the operation Deep check for failures from the :Self-repair-A module. The EUREMA interpreter's program counter points to this operation when intercepting the self-repair feedback loop. The interpreter requires keeping this operation to properly continue execution after executing the :Select-repair-strategies. Nevertheless, other kinds of adaptations effecting the operation such as adapting its wiring to other operations (*i. e.*, the control flow), its usage of runtime models, and its used runtime models are possible.

The advantage of directly using EUREMA models as reflection models of feedback loops is that the causal connection is ensured by construction. This avoids the development of monitor and execute activities for higher-layer feedback loops, which create and maintain reflection models of lower-layer feedback loops. However, by using the same model of a feedback loop for executing as well as adapting it, the adaptation cannot be decoupled from the execution. Thus, any adaptation performed by the higher-layer feedback loop is instantaneously enacted to the lower-layer feedback loop. This avoids previewing the adaptation, for instance, to analyze a projected adaptation before enacting it. Moreover, using the same model for execution and adaptation means that also the same abstraction level is used for both. Consequently, a higher-layer feedback loop has to work at the abstraction level of the execution to adapt the lower-layer loop. Thus, a higher-layer feedback loop does not create and work on higher-level, adaptation-specific abstractions of the lower-layer loop.

5.3.2 *Declarative Reflection: User-defined Reflection Models*

In declarative reflection, the program of a system implements the domain functionality and a separate representation of the program is maintained and used for adapting it. This requires the synchronization of the program and the representation, that is, the causal connection as discussed in Section 2.1.5. Thus, user-defined reflection models expressed in an arbitrary language can be used to represent the program under adaptation.

Applying this idea to EUREMA means that a higher-layer feedback loop employs a user-defined reflection model representing the lower-layer feedback loop. Thus, two representations of the lower-layer loop are used at runtime: The EUREMA models for specifying and executing it and that are kept alive by the EUREMA interpreter. The user-defined reflection model for adapting it, which is maintained by the higher-layer feedback loop.

For the example, the higher-layer feedback loop that follows the idea of declarative reflection to adapt the self-repair feedback loop is defined by the FLD shown in Figure 38. The task of this Select-repair-strategies-2 feedback loop is to select repair strategies from a repertoire and provide them to the self-repair feedback loop by means of adaptation.

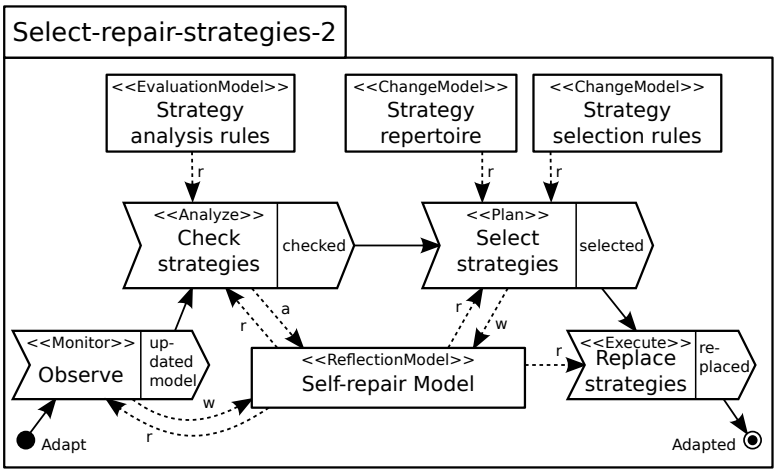


Figure 38: FLD for the Higher-Layer Feedback Loop with Declarative Reflection.

Therefore, the Select-repair-strategies-2 feedback loop performs the similar function as the Select-repair-strategies feedback loop discussed in the previous section (*cf.* the FLD in Figure 36 on Page 83). Particularly, both perform the same analyze and planning activities to determine if and which repair strategies should replace the strategies currently used by the self-repair feedback loop. However, since the Select-repair-strategies-2 feedback loop maintains its own reflection model of the self-repair feedback loop, it requires corresponding monitor and execute activities. As depicted in Figure 38, the Observe operation monitors the self-repair feedback loop and accordingly keeps the Self-repair Model up-to-date. This model is a reflection of the underlying self-repair feedback loop. Using this reflection model, the analyze and plan activities are carried out. The replacement of the repair strategies is prescribed in the Self-repair Model. The execute activity, called Replace strategies, finally uses this model to enact the prescribed replacement in the self-repair feedback loop.

The corresponding LD that shows the layered feedback loops using declarative reflection is depicted in Figure 39. Similar to the case of procedural reflection (*cf.* Figure 37 on Page 84), this LD shows that the :Select-repair-strategies-2 megamodel module at Layer-2 senses and effects the :Self-repair module including the :Self-repair-A module at Layer-1. Moreover, the same triggering condition is used for the :Select-repair-strategies-2 module as for the :Select-repair-strategies-1 module in the previous case.

However, a particular aspect in this case is that the higher-layer feedback loop maintains a user-defined reflection model (*cf.* Self-repair Model in the FLD in Figure 38) instead of directly using the EUREMA models specifying and executing the self-repair feedback loop. Thus, the reflection model does not require a binding to any EUREMA model, which would have been defined in the LD. In this context, a reflection model is user-defined because its metamodel can be user-defined and it is maintained by user-defined model operations for the monitor and execute activities (*cf.* Observe and Replace strategies operations in the FLD in Figure 38). Thus, engineers may decide which information about the lower-layer feedback loop is covered by the reflection model as well as the abstraction level of the model. This supports adaptation-specific reflection models. However, the engineers must

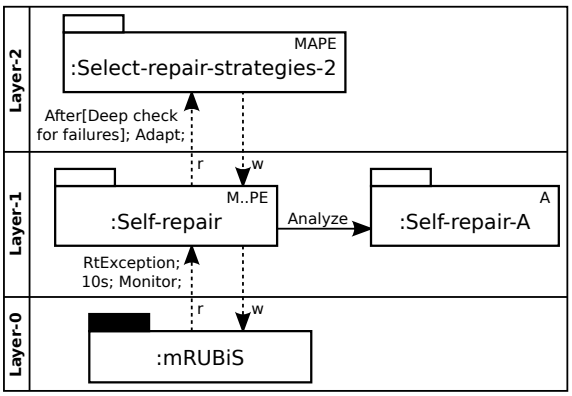


Figure 39: LD with Layered Feedback Loops and Declarative Reflection.

ensure the causal connection between the reflection model and the reflected feedback loop by defining and implementing model operations for the monitor and execute activities.

For this purpose, sensors and effectors provided by EUREMA can be used to observe and adjust feedback loops by means of the EUREMA models. For sensing EUREMA models, they can be queried and events notifying about the execution and changes of these models are emitted by the EUREMA interpreter. For effecting EUREMA models, basic means to change models are provided such as changing attribute values, or adding and removing nodes and relationships. Engineers can use these sensors and effectors to implement model operations for the monitor and execute activities that realize the causal connection.

The advantage of user-defined reflection models is that the higher-layer feedback loop may run decoupled from the lower-layer feedback loop since the reflection model is kept separate from the EUREMA models specifying and executing the lower-layer loop. Thus, an adaptation can be planned, analyzed, and even reverted in the reflection model without affecting the lower-layer feedback loop. However, the disadvantage is that both representations have to be synchronized to each other to ensure the causal connection. Nevertheless, this synchronization can be simplified since both representations are models conforming to MDE principles, that is, they have potentially different metamodels but the same metametamodel (*cf.* Section 2.1). Thus, one-to-one copies of EUREMA models can be directly provided as reflection models or MDE techniques such as model synchronization to keep the EUREMA model and the user-defined reflection model consistent to each other can be used. We have shown the applicability of such techniques for the monitor and execute activities of a feedback loop to maintain multiple reflection models at runtime [22, 28]. Finally, user-defined reflection models can be at a higher level of abstraction than the EUREMA models and thus emphasize adaptation-specific concerns such as the performance or success rate of the lower-layer feedback loop. Thus, a higher-layer feedback loop does not have to work on the EUREMA model that is also concerned with the execution of feedback loops but it may focus on adaptation-specific concerns.

Overall, LDs explicitly describe the feedback loops at the individual layers of the architecture and how the feedback loops are interrelated and particularly stacked by sense and effect relationships. This makes the adaptive control strategy visible in the architectural design of the self-adaptive software. In this context, LDs make further visible whether procedural or declarative reflection is used. If a reflection model is bound to a megamodel module in the LD, procedural reflection is used, otherwise declarative reflection.

In general, the number of layers is not restricted by EUREMA such that engineers may stack feedback loops as often as required. The stacking of feedback loops is only restricted

with respect to controlling concurrently running feedback loops (*cf.* Section 5.2). EUREMA does not allow a higher-layer feedback loop to control two or more lower-layer feedback loops that run concurrently. Otherwise, this would require to intercept two or more feedback loops in a synchronized manner. That is, *all* controlled feedback loops have to reach their designated interception points until the higher-layer feedback loop can be executed to perform safe adaptations. This synchronized interception may cause major disturbances of the adaptation engine until all controlled feedback loops are intercepted (*i. e.*, intercepted feedback loops are blocked until the other feedback loops reach their interception points). It may even result in a deadlock because some feedback loops will never reach their interception points, for instance, when a certain branch of a feedback loop is never executed and the interception point refers to operations or transitions in this branch. At the costs of this restriction, EUREMA automatically supports safe adaptations of feedback loops in layered architectures without requiring from engineers to implement any functionality that enables the interception and quiescence of feedback loops. This functionality is generically realized and provided by EUREMA. Thus, engineers are not concerned with such issues but can focus exclusively on developing the adaptation logic of the feedback loop.

In this section, we have discussed the modeling of layered feedback loops with EUREMA. Similar to the modeling of multiple feedback loops and their coordinated execution (*cf.* Section 5.2), the same language concepts used for modeling single feedback loops are used for modeling layered feedback loops. Thus, there is no need for engineers to learn additional concepts to specify adaptive control architectures by stacking feedback loops in EUREMA. Moreover, EUREMA supports developing such architectures because EUREMA-based feedback loops are adaptable by construction and EUREMA models can be directly used as reflection models (*cf.* procedural reflection) or they are amenable for MDE techniques to create and maintain user-defined reflection models (*cf.* declarative reflection).

Finally, based on the EUREMA triggering mechanism, the EUREMA interpreter ensures that layered feedback loops do not run concurrently, which avoids any inconsistencies between the layered feedback loops and which supports safe adaptations of feedback loops. Hence, EUREMA does not burden engineers with synchronizing the execution of layered feedback loops. Engineers only have to specify the triggering conditions for these feedback loops, based on which EUREMA realizes the synchronized execution of the feedback loops.

5.4 OFF-LINE ADAPTATION

Self-adaptation envisions that software is able to adjust itself according to changes in its own state, context, or requirements by automating and taking over some of the adaptation activities that are otherwise performed off-line in the context of maintenance. However, we cannot expect that self-adaptive software is able to cope with *all* evolution needs itself and thus, to fully automate and take over all kinds of off-line adaptation activities. This requires the typical maintenance of self-adaptive software. Thus, self-adaptive software has to undergo adaptation due to maintenance steps that are analyzed and planned off-line by engineers and then executed on-line to the running software. Consequently, self-adaptation and maintenance together address the long-term evolution of the software.

On the one hand, this calls for the co-existence of on-line and off-line adaptation, that is, the coordination of activities that are performed by feedback loops (*i. e.*, internally to the self-adaptive software in the runtime environment) respectively by engineers (*i. e.*, externally to the self-adaptive software in the development environment) to properly evolve the software. Of particular interest is the execution of an adaptation that has been ana-

lyzed and planned off-line to the running self-adaptive software. As the execution is done on-line while at the same time feedback loops might be operating for self-adaptation, the *coordination* of both kinds of adaptations is necessary.

On the other hand, this calls for a maintenance *interface* provided by the adaptation engine to support engineers in observing the running self-adaptive software and executing adaptations developed off-line to the running software. Obtaining runtime information about the software, engineers use this information in the development environment to analyze and plan an adaptation, for instance, to fulfill a change request. The resulting adaptation must then be executed, for instance, by updating the software.

In general, since we consider self-adaptive software following the external approach, maintenance may target the adaptation engine or the adaptable software. For the adaptation engine, we may conceive the following maintenance scenarios:(1) adding or removing a feedback loop at different layers in the running adaptation engine, (2) changing an existing and running feedback loop in the adaptation engine, and (3) supporting the adaptation concerning legacy feedback loops. Finally, the last maintenance scenario targets the adaptable software, that is, (4) the adaptable software is directly changed while being operational and without changing the feedback loops already running in the adaptation engine.

In the following, we use these four scenarios to discuss how EUREMA supports off-line adaptation and thus the maintenance of self-adaptive software by enabling the co-existence and coordination of on-line and off-line activities, and providing a maintenance interface of the running software. Particularly, we discuss how EUREMA is used to model an adaptation that is analyzed and planned off-line by engineers and how the resulting models are used for executing the adaptation on-line.

The rationale of EUREMA's approach to off-line adaptation is based on two principles. First, we interpret a maintenance process as a feedback loop (*cf.* [272, 274]) that is effectively split up into on-line monitor and execute and off-line analyze and plan activities.

Monitor: On-line monitoring is supported by observing EUREMA models including the runtime models that are used within the feedback loops. Since EUREMA models as visualized by FLDs and LDs are directly used for executing adaptation engines, they innately reflect the running feedback loops and the runtime architecture of the self-adaptive software. Moreover, runtime models used within feedback loops either reflect the adaptable software (*e. g.*, the architectural model reflecting mRUBiS) or specify the self-adaptation (*e. g.*, change models such as the repair strategies). Thus, they describe knowledge about the adaptable software and the self-adaptation logic of individual feedback loops.

Analyze: Using the maintenance interface provided by the EUREMA interpreter, engineers in the development environment may retrieve snapshots of these models and use them to analyze off-line the running system, its context, and requirements. For this purpose, runtime models have to be used since they contain the latest information about the running system, context, and requirements that may dynamically change at runtime, which is not reflected in the design-time models.

Plan: If an adaptation is needed, for instance, due to a change request, engineers use these models to plan off-line an adaptation. In this context, an adaptation may refer to changes according to the scenarios (1)-(4). In general, an adaptation planned off-line should be applicable despite intermediate self-adaptations that have occurred on-line. Therefore, engineers should have knowledge about the configuration space of the self-adaptive software to anticipate potential states of the software and to make an off-line adaptation applicable

to these states. Having planned an adaptation, engineers use the EUREMA language to specify the adaptation and its enactment to the running self-adaptive software.

Execute: The resulting EUREMA models are sent to the EUREMA interpreter through the maintenance interface for on-line execution. The interpreter executes these models at the designated layer of the adaptation engine, which enacts the adaptation developed off-line to the running self-adaptive software. This execution accomplishes a maintenance cycle such that these models can be optionally removed from the adaptation engine afterwards.

This maintenance process illustrates that we consider EUREMA models and the contained runtime models as carriers of knowledge about the self-adaptive software between the runtime and development environments. By seamlessly exchanging models, on-line and off-line adaptation activities interact and both environments are integrated. Moreover, the maintenance process illustrates that some of the activities are performed off-line by engineers in the development environment while other activities are modeled with EUREMA and performed on-line by the EUREMA interpreter in the runtime environment. Concerning the activities performed by engineers in a concrete maintenance cycle, EUREMA does not prescribe the maintenance process. Thus, the process may consists of more refined activities from the software evolution field, for instance, impact analysis, release planning, requirements analysis, and requirements update instead of the more abstract analyze and plan activities. Concerning the on-line activities, the maintenance process shows that the EUREMA language can be used to specify maintenance activities besides feedback loop activities for self-adaptation. Similar to feedback loops modeled with EUREMA, maintenance activities that are specified by FLDs are treated at runtime as megamodel modules.

The second principle we exploit for EUREMA's approach to off-line adaptation is the layered architecture of the self-adaptive software. We consider layers of feedback loops that operate on top of the lowest layer which is the adaptable software. Thereby, a layer may adapt the layer directly below while the lower layer should not be aware of and dependent on any higher layer. To enable the maintenance of the self-adaptive software, the layered architecture is exploited to execute an off-line adaptation to the self-adaptive software in coordination with the running feedback loop instances in the software. Therefore, the megamodel module executing an off-line adaptation will be usually placed at the layer above the topmost layer of the self-adaptive software. Thus, the module may intercept the currently running feedback loop instances to avoid any concurrency and thus conflicts concerning the execution of adaptations. This ensures that an off-line adaptation is executed without interfering or being interfered by feedback loop instances performing self-adaptation. Such a coordinated execution is explicitly specified with LD to describe at which layer a megamodel module performing the off-line adaptation will be placed and which other feedback loop instances will be intercepted. Additionally, the execution of an off-line adaptation and the execution of existing feedback loop instances can be coordinated similarly to the coordinated execution of multiple feedback loops (*cf.* Section 5.2). Thus, we may specify the coordinated execution in an FLD and properly place an instance of this FLD as a megamodel module in the LD, which enacts the coordination.

To discuss EUREMA's support for off-line adaptation by considering maintenance processes as feedback loops and by exploiting layered architectures, we use the four scenarios discussed previously. To avoid repetitions, the first scenario—adding or removing a feedback loop—is discussed in more depth to introduce all relevant details of EUREMA's approach to off-line adaptation, which apply to all scenarios.

5.4.1 *Adding or Removing a Feedback Loop*

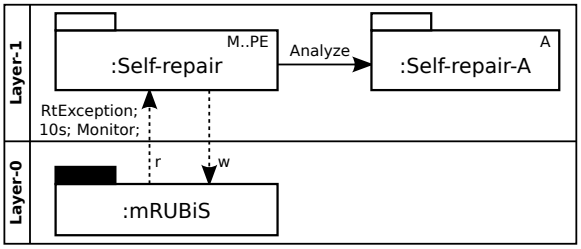
The first maintenance scenario is to add or remove a feedback loop at various layers of the adaptation engine. That is, a feedback loop can be added to a layer at which feedback loops are already operating (*e. g.*, an engine employing a self-repair feedback loop can be additionally equipped with a self-optimization feedback loop, which results in multiple feedback loops operating at the same layer as discussed in Section 5.2) or it can be added to a new layer on top of the existing layers of the engine (*e. g.*, an engine employing a self-repair feedback loop can be additionally equipped with a higher-layer feedback loop managing the repair strategies as discussed in Section 5.3). Likewise, feedback loops can be removed from an arbitrary layer of the engine. It is further conceivable to add or remove a fragment of a feedback loop from the adaptation engine. For instance, an engineer may only specify and add a monitoring activity to the engine that maintains a certain runtime model representing the adaptable software. This model can then be transferred to the development environment where it is analyzed by the engineer. However, in the following we discuss the scenario of adding a complete feedback loop to the adaptation engine.

We now consider our running example that employs the self-repair feedback loop to automatically heal failures in mRUBiS. The corresponding layered architecture is represented by the LD in Figure 40a. As discussed in Section 5.3, the self-repair feedback loop uses a set of repair strategies that might have to be adjusted from time to time by selecting appropriate strategies from a repertoire for this set. This task can be done by a higher-layer feedback loop. Thus, an engineer decides to develop such a higher-layer loop and to evolve the self-adaptive software from a two-layer to a three-layer architecture shown in Figure 40c. We refer to Section 5.3 for a detailed discussion of the higher-layer loop and the purpose of the three-layer architecture. Hence, this example addresses the maintenance scenario of adding a higher-layer feedback loop on top of another feedback loop.

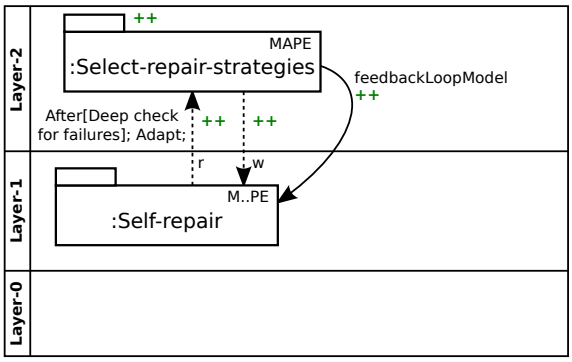
To perform such a change, an engineer proceeds as follows. At first, she retrieves snapshots of the runtime models that are currently employed in the system from the EUREMA interpreter. In particular, these models are the EUREMA models specifying and executing the self-repair feedback loop (*i. e.*, instances of the FLDs in Figure 21 on Page 66 and Figure 19 on Page 65) and the layered architecture (*i. e.*, the LD in Figure 40a). These models allow the engineer to observe and analyze the deployed self-repair feedback loop and the contained runtime models used within the loop (*e. g.*, the reflection model describing the runtime architecture of mRUBiS, or the change models specifying the employed repair strategies). Based on such knowledge about the running feedback loops and mRUBiS, the engineer can make an informed decision. For instance, based on the employed repair strategies and the untreated failures in the reflection model, she may identify the need to replace the repair strategies. Frequently identifying such a need, she may decide to develop a higher-layer feedback loop that automatically selects appropriate strategies and replaces the currently used strategies with the selected ones in the self-repair feedback loop.

The development of the higher-layer feedback loop is done with EUREMA, that is, the engineer specifies the loop with an FLD and provides implementations for the model operations and runtime models used within the loop. For the example, this results in the FLD called Select-repair-strategies shown in Figure 36 on Page 83.

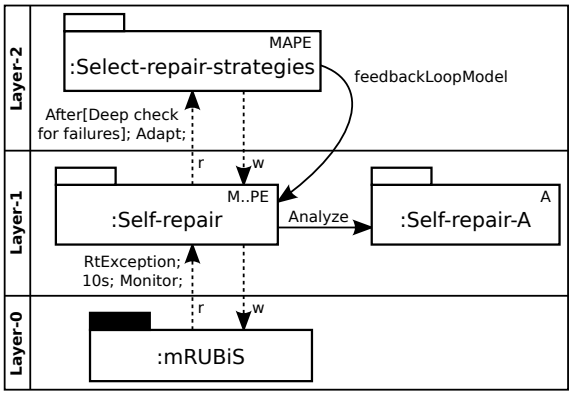
In addition, the engineer has to specify how the developed feedback loop as modeled with the FLD should be integrated as a megamodel module into the running system. This is done by modeling an adaptation rule describing how the layered architecture of the system should be adjusted. An example of such a rule is shown in Figure 40b that follows a



(a) LD Before the Off-line Adaptation.



(b) Adaptation Rule to Adjust the LD.



(c) LD After the Off-line Adaptation.

Figure 40: An Off-line Adaptation Adding a Feedback Loop.

graph transformation approach to modify a model in-place.[6] This rule specifies how the current layered architecture reflected by the LD in Figure 40a should be changed to obtain the target architecture shown in Figure 40c. In general, the elements of a graph transformation rule that are annotated with ++ (- -) are added to (removed from) the LD if a match for its elements having no or a - - annotation is found in the LD. For the example, applying the rule identifies a match for the :Self-repair module at Layer-1, it consequently adds the module :Select-repair-strategies (*i. e.*, an instance of the FLD modeled by the engineer) at Layer-2, establishes the sense and effect relationships between both modules including the triggering condition, and binds the feedbackLoopModel used in the :Select-repair-strategies module to the :Self-repair module. This results in the desired three-layer architecture (*cf.* Figure 40c).

6 EUREMA does not define or prescribe the language for such adaptation rules. Theoretically, any mechanism can be used that specifies and executes changes of a model, in this context of the LD. We apply a mechanism based on graph transformations [57, 8] that is suitable for structural changes of models. Technically, the EUREMA interpreter integrates an existing execution engine for graph transformations, which can be easily replaced with an alternative mechanism and engine.

In general, an adaptation rule does not only specify the addition and removal of megamodel and software modules from certain layers as well as the sense (including a triggering condition) and effect relationships between modules. It additionally has to specify different kinds of bindings. Complex and basic model operations contained in an added megamodel module must be bound to megamodel and software modules via use relationships between the corresponding modules (*cf.* Section 5.1.4). Furthermore, but only for procedural reflection between feedback loops, a reflection model contained in an added megamodel module must be bound to the adapted megamodel module (*e. g.*, the feedbackLoopModel contained in the :Select-repair-strategies module must be bound to the :Self-repair module).

To execute the off-line adaptation, the engineer loads up the modeled FLD and the adaptation rule through the maintenance interface to the EUREMA interpreter. The interpreter instantiates the FLD to a megamodel module (*e. g.*, the :Select-repair-strategies module) and applies the rule to adapt the layered architecture (*e. g.*, from a two-layer to a three-layer architecture). Having adapted the architecture, the EUREMA interpreter executes the added megamodel module when the corresponding triggering condition is fulfilled such that the off-line adaptation is integrated as well as eventually executed in the running system.

While a module is integrated into (or removed from) the adaptation engine, the other modules may operate without any impact except of delays. Such delays are caused by supporting consistent architectural adaptation in a quiescent state (*cf.* [259]). In this context, adapting the layered architecture requires that all affected megamodel and software modules are quiescent, which is achieved by the EUREMA interpreter that blocks initiating new executions of feedback loops at the lowest layer of the adaptation engine and that waits until the current executions terminate. If these feedback loops have finished their execution and are quiescent, all higher-layer feedback loops are quiescent as well since they are only executed by intercepting running lower-layer feedback loops. As soon as all feedback loops are quiescent, that is, non of them are currently running, the layered architecture can be safely reconfigured by applying an adaptation rule such as the one shown in Figure 40b. The EUREMA interpreter applies such a rule while the adaptation engine is quiescent. After that it breaks the quiescent state, that is, it disables any blocking of feedback loops and allows initiations of feedback loop executions. A detailed discussion of quiescence with respect to the whole adaptation engine is given in Section 6.6.

Afterwards, the execution of a newly added module is coordinated with the other modules by a triggering condition defined in the adaptation rule (*e. g.*, the triggering condition of the added :Select-repair-strategies module in Figure 40b). Such a triggering condition for a megamodel module that senses another megamodel module enables the exclusive execution by intercepting the execution of the sensed lower-layer module to run the higher-layer module (*cf.* Section 5.3). This enables the exclusive and thus synchronized execution of the modules to coordinate the execution of an off-line adaptation with running feedback loops.

In general, EUREMA's approach to off-line adaptation is technically feasible due to the LD that is kept alive at runtime and used as a procedural reflection model of the self-adaptive software. Thus, the LD is used by the EUREMA interpreter to execute the feedback loops and it can be directly modified to dynamically adapt the feedback loops. Such modifications are done by applying an adaptation rule for maintenance purposes (*cf.* Figure 40b). Moreover, it is conceivable to employ another adaptation engine on top of the engine that is reflected by the LD. The higher-layer engine then operates on the LD.

The following dynamic changes of LDs and therefore of adaptation engines are supported. Layers and megamodel modules can be added and removed from the engine. Since sense, effect, and use relationships between modules can be interpreted as dependencies,

a module depends on another module if the former senses, effects, or uses the latter. Thus, a module can be added if the modules it depends on already exist in the running self-adaptive software.[7] Likewise, a module can be removed if there does not exist any other module that depends on it. Considering the example in Figure 40c, the :Self-repair-A cannot be removed since the :Self-repair module uses and depends on it. Likewise, the :Self-repair module cannot be removed since it is sensed and effected by the :Select-repair-strategies module. However, the :Select-repair-strategies module can be removed since no other module depends on it. To remove this module, it is required to specify and upload just an adaptation rule defining the removal of this module from the LD. For this purpose, an FLD is not needed since no new functionality is added. In general, the adaptation rule for removing a module must also remove the module's sense and effect relationships to other modules and the bindings of the module's operations and models to other modules. For instance, removing the :Select-repair-strategies module requires among others the removal of the sense and effect relationships to the :Self-repair module as well as the binding of the feedbackLoopModel to the :Self-repair module. In general, the sense and effect relationships as well as the bindings can also be individually added or removed from an LD.

A module that already exists in the adaptation engine can be adapted by changing the bindings of its complex model operations to other megamodel modules, and of its basic model operations to software modules implementing these operations. This has been discussed in Section 5.1.5 in the context of variability that can now be exploited at runtime.

The changes discussed so far alter the number and composition of modules in flexible layers of the adaptation engine. Finally, LDs reflect the triggering conditions of megamodel modules that can be dynamically changed by adapting their constituent parts (*cf.* Section 5.1.3). Thus, events whose occurrences trigger a megamodel module can be added or removed from the trigger specification, the period of a trigger can be adjusted, and finally, a different initial state of the feedback loop can be chosen for initiating the execution.

All these kinds of changes particularly modify the composition of modules and thus the structuring of feedback loops in a layered architecture. We consider such changes as more extensive and less frequent adaptations than adjusting the internals of a feedback loop (*e. g.*, replacing the adaptation strategies). Therefore, such changes are addressed by off-line adaptation while the internals of feedback loops can be adjusted by self-adaptation in layered architectures (see Section 5.3) but also by off-line adaptation (see next section).

Summing up this example illustrating the maintenance scenario of adding a feedback loop to the running self-adaptive software, it shows how EUREMA supports adaptation that has not been anticipated when initially developing and deploying the software, such as evolving the software from a two-layer to a three-layer architecture (*cf.* Figure 40).

5.4.2 *Changing a Feedback Loop*

In this section, we discuss the second maintenance scenario, which is changing an existing and running feedback loop in the adaptation engine. Particularly, the internals of the feedback loop in terms of the model operations, control flow among the operations, runtime models, and usage of the models are the subject of change. We may consider such changes

7 Issues due to cyclic dependencies do not matter since cyclic dependencies are either not allowed (*i. e.*, the principle of layered architectures that an element at a certain layer can only adapt an element at the adjacently lower layer prevents any cycles concerning either sense or effect relationships) or they should be generally avoided for functional dependencies (*e. g.*, two modules should not use each other's functionality, which otherwise would cause a cyclic dependency).

as patches of running feedback loops. Thus, we use EUREMA to specify a patch process. The resulting FLD is loaded up and executed once in the running system to perform the patch. This contrast with the previous scenario, in which an FLD specifies a feedback loop that remains in the system until it might be removed by a future off-line adaptation.

We now consider the same example that employs the self-repair feedback loop to automatically heal failures in mRUBiS and that requires maintaining the repair strategies from time to time by developing new or selecting other strategies. In the previous scenario, the focus was on selecting strategies from a repertoire, which can be easily automated and thus realized by self-adaptation. In contrast, we now focus on developing novel strategies for the self-repair feedback loop since some types of failures have not been anticipated when initially deploying the feedback loop with its strategies. Since automating the development of strategies is far more difficult than just selecting strategies, this task is not automated but performed by an engineer. Having developed off-line the novel strategies, the engineer patches afterwards the running feedback loop to integrate these strategies.

To perform such an update of the running self-repair feedback loop, the engineer proceeds as follows. She retrieves snapshots of the EUREMA models and the contained runtime models from the EUREMA interpreter to see which strategies are used and which failures have occurred and not been repaired in the reflection model representing mRUBiS. By analyzing these models, the engineer identifies the need for new repair strategies since untreated failures exist that cannot be handled by the currently deployed strategies. She develops these new strategies using the same formalism as employed for the existing strategies in the self-repair feedback loop. That is, strategies are expressed in a certain language and specified in a model that is kept alive at runtime (*cf.* the runtime model Repair strategies in the FLD in Figure 21 on Page 66). Thus, the new strategies are expressed in the same language, which results in the model New repair strategies. To add this new model and hence the new strategies to the running self-repair feedback loop, the engineer models a patch process with EUREMA as shown by the FLD in Figure 41 to enact the off-line adaptation.

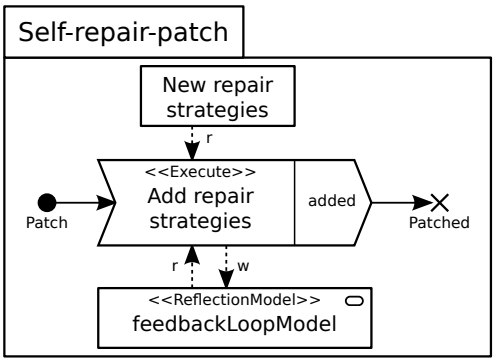
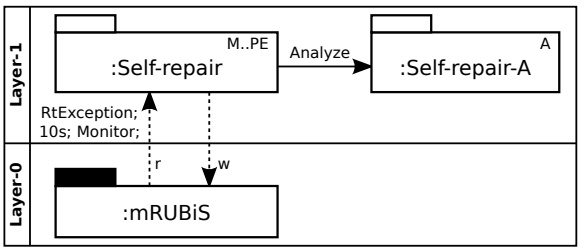
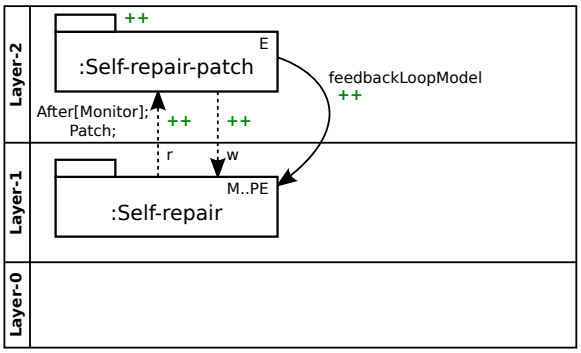


Figure 41: FLD for Patching a Feedback Loop.

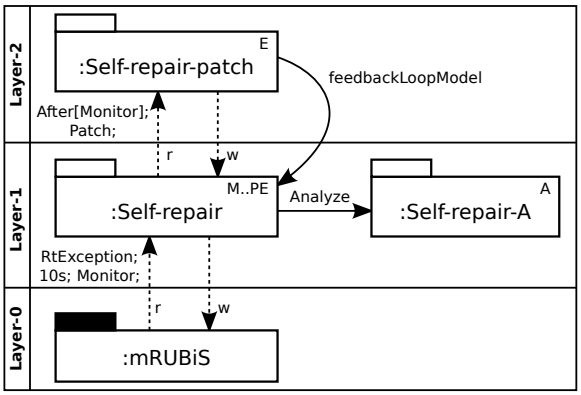
This patch process consists of a single model operation that adds the New repair strategies to the feedbackLoopModel reflecting the self-repair feedback loop. Particularly, the operation merges the New repair strategies into the existing Repair strategies used in the self-repair loop. Besides specifying this patch process, the engineer has to implement the single operation. In this example, a code-based implementation is used as the FLD does not show any further runtime model that is consumed by the operation. Finally, the patch process terminates in a destruction state, denoted by a cross, which is a special final state of an FLD. It defines



(a) LD Before and After the Off-line Adaptation.



(b) Adaptation Rule to Adjust the LD.



(c) LD During Executing the Off-line Adaptation.

Figure 42: An Off-line Adaptation Patching a Feedback Loop.

that the megamodel module as an instance of the FLD is executed exactly once and when it reaches the final state, it is destroyed and erased from the adaptation engine.

Similar to the previous scenario, to integrate the FLD developed off-line as a megamodel module into the adaptation engine, the engineer specifies an adaptation rule as shown in Figure 42b. This rule and the FLD shown in Figure 41 are loaded up to the EUREMA interpreter. The interpreter instantiates the FLD to the :Self-repair-patch module and applies the rule on the LD reflecting the running self-adaptive software as shown in Figure 42a. This adaptation temporarily evolves the two-layer to a three-layer architecture as shown in Figure 42c. The :Self-repair-patch module is added to the layer above the :Self-repair module as the former module adapts the latter. Therefore, the :Self-repair-patch module can intercept the :Self-repair module to perform a safe adaptation of the repair strategies.

Using the uploaded adaptation rule, the EUREMA interpreter safely adapts the layered architecture in a quiescent state. Afterwards, the :Self-repair-patch module can be activated by its trigger After[Monitor]; Patch; that enables the coordinated execution of the :Self-repair-patch and :Self-repair modules. The execution of the :Self-repair module is inter-

cepted after this module has executed its initial operation called Monitor (*cf.* Figure 21 on Page 66) in order to run the :Self-repair-patch module starting in its initial state Patch (*cf.* Figure 41). After the patch module has added the new repair strategies into the :Self-repair module, the :Self-repair module continues execution and uses the new strategies for the next planning. The :Self-repair-patch module will be executed exactly once and it will be destroyed and removed from the adaptation engine afterwards. The resulting architecture is the same as the one before applying the patch (*cf.* Figure 42a) except that new repair strategies that have been developed off-line are now used by the self-repair feedback loop.

This scenario exemplifies that based on dynamic and flexible layers, feedback loops running at a certain layer $1..n$ can be dynamically changed by megamodel modules that realize adaptations (*e. g.*, patches developed off-line) and that temporarily operate at the next higher layer $2..(n+1)$ until the off-line adaptation has been achieved.

5.4.3 *Support for Legacy Feedback Loops*

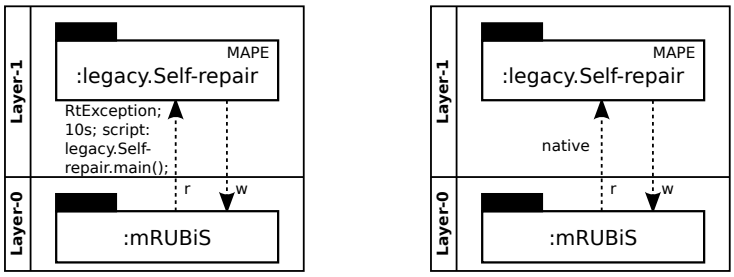
We have discussed so far how megamodel modules, that is, feedback loops specified with FLDs can be adjusted by off-line adaptation. Particularly, we have considered adding, changing, and removing megamodel modules from the adaptation engine, which is feasible since the EUREMA interpreter is in charge of managing and executing such modules.

In this context, EUREMA provides basic support for managing and executing legacy feedback loops. Such feedback loops are not specified with EUREMA. Therefore, they are handled as black boxes and described as software modules (*i. e.*, packages with black tabs) in LDs (*cf.* Section 5.1.2). Consequently, management and execution support is not provided for the internals of legacy feedback loops but only for those feedback loops as a black box.

On the one hand, representing legacy feedback loops in LDs and their sense and effect relationships to other modules makes them visible in the architecture of self-adaptive software. Thus, they can be included in the design and evolution of the software. On the other hand, EUREMA may take over the responsibility of activating legacy feedback loops if they can be triggered in a similar way as EUREMA feedback loops (*cf.* Section 5.1.3), that is, periodically or by events. Thus, triggering conditions are specified for the software modules implementing legacy feedback loops in LDs and the EUREMA interpreter initiates the execution of these feedback loops if the triggering conditions are satisfied. Controlling the triggering of legacy feedback loops provides the flexibility for basic off-line adaptations.

This is illustrated by the LD in Figure 43a. The :legacy.Self-repair module is an instance of a legacy feedback loop that senses and effects :mRUBiS and that realizes self-repair capabilities. It is triggered based on the condition RtException;10s;script: legacy.Self-repair.main(). Thus, the EUREMA interpreter initiates the execution of this module if :mRUBiS emits an event of type RtException and if ten seconds since the end of the previous execution of this module have elapsed by invoking the method legacy.Self-repair.main() on the module. The grammar defining the textual language to express such triggering conditions for legacy as well as EUREMA feedback loops is discussed in Appendix A.4.

Being in charge of triggering instances of legacy feedback loops, the EUREMA interpreter can decommission such legacy modules to migrate the adaptation engine to megamodel modules that are specified by EUREMA. Such a migration can be realized by off-line adaptations, especially by removing the legacy modules and adding the megamodel modules (*cf.* Section 5.4.1 discussing the addition and removal of feedback loops). For the given example, an off-line adaptation may specify the removal of the :legacy.Self-repair module such that the EUREMA interpreter will not trigger this module any more. Removing the



(a) EUREMA Trigger. (b) Native Trigger.

Figure 43: LD with a Legacy Feedback Loop and (a) an EUREMA or (b) a Native Trigger.

actual binaries of the legacy module is out of scope of EUREMA since it can be specific to employed technologies. Subsequently, an off-line adaptation may specify the addition of a self-repair feedback loop as specified with EUREMA in Section 5.1. In this example, there is only a short time interval between the removal of the legacy and the addition of the EUREMA feedback loop, when there is no feedback loop operating in the adaptation engine. Moreover, the EUREMA feedback loop has to start from scratch if it does not reuse any knowledge from the legacy feedback loop. Despite these issues, we omit further details for this example since the addition and removal of feedback loops has been generally discussed in Section 5.4.1. The discussion there also applies to legacy feedback loops except of modeling feedback loops with FLDs since legacy feedback loops are black boxes and not specified by EUREMA. Moreover, legacy feedback loops are not a key aspect of EUREMA.

Thus, EUREMA can support the migration from legacy to EUREMA feedback loops by controlling the activation and decommission of megamodel and software modules that implement the feedback loops. Similarly, it is conceivable to deploy a self-adaptive software with no feedback loop employed in the adaptation engine but equipped with the EUREMA interpreter. Then, an off-line adaptation may add a feedback loop to the running engine, which supports the bootstrapping of feedback loops after the initial deployment.

In contrast, if it is not possible to trigger a legacy feedback loop using EUREMA's triggering concept, the EUREMA interpreter cannot control the activation of the legacy feedback loop. Hence, no triggering condition can be specified for the corresponding software module. However, EUREMA still allows the engineer to model such software modules in LDs as illustrated in Figure 43b. Thus, legacy feedback loops whose triggering is not controlled by EUREMA can still be made visible in the architecture of the self-adaptive software.

Similar to the LD in Figure 43a, this LD shows an instance of a legacy feedback loop, namely the :legacy.Self-repair module that senses and effects :mRUBiS. However, instead of an EUREMA triggering condition, the LD defines the native triggering of the legacy module. Such a native trigger indicates that the triggering is controlled by some glue code that hardwires the legacy module and :mRUBiS. Consequently, the EUREMA interpreter cannot interfere and support off-line adaptation of legacy feedback loops with native triggers.

5.4.4 *Changing the Adaptable Software*

The last maintenance scenario we are discussing does not address changes of the adaptation engine but directly of the adaptable software. Such a scenario is needed if the employed feedback loops are not able to perform the desired change and if it is not worthwhile to automate the change with a feedback loop since such changes are not needed frequently.

For instance, a specific component of the adaptable software should be replaced as an improved version has been developed off-line due to change requests by users. In the context of mRUBiS, users might request novel authentication mechanisms to authenticate on the marketplace by using third-party services. This calls for replacing the authentication component deployed in mRUBiS with a novel version of it. The employed self-repair feedback loop is not able to perform such a specific change such that off-line adaptation is used.

To perform such an off-line adaptation of mRUBiS, the engineer proceeds as follows. At first, she develops the new version of the component based on the change requests. Then, she obtains snapshots of the EUREMA models including the contained runtime models that are currently used on-line. Hence, she obtains the model reflecting the runtime architecture of mRUBiS as well as the repair strategies employed by the self-repair feedback loop. Knowing the runtime architecture as well as the repair strategies, she is aware of the system's configuration space. Accordingly, she identifies a reconfiguration that integrates the newly developed component into the system that may self-adapt within this configuration space. This reconfiguration process is specified with an FLD as shown in Figure 44. It is developed off-line and will be executed on-line to enact mRUBiS.

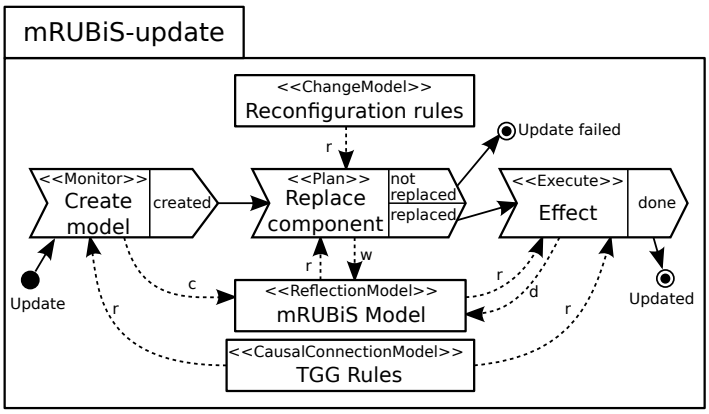


Figure 44: FLD for Updating the Adaptable Software.

The reconfiguration process consists of three model operations. The Create model operation monitors the running mRUBiS and uses the TGG Rules to create the architectural mRUBiS Model that is reflecting and causally connected to mRUBiS. The Replace component operation plans the exchange of the specific component by applying Reconfiguration rules that change accordingly the architectural model. If the component cannot be replaced in the model, for instance, due to architectural failures that might have occurred meanwhile in mRUBiS, the process terminates in the final state Update failed. If the component can be replaced in the architectural model, the Effect operation loads the new component, enacts the reconfiguration prescribed in the model to the running mRUBiS, and destroys the model. Then, the process terminates in the final state Updated.

This reconfiguration process directly changes mRUBiS and it will be therefore placed at the lowest layer of the adaptation engine. However, the self-repair feedback loop is already running at the lowest layer of the engine and potentially adapting the mRUBiS at the same time. Consequently, to avoid any conflicts due to concurrent adaptations performed by the reconfiguration process and the self-repair feedback loop, both have to be coordinated. This is a similar same issue as coordinating multiple, (inter)dependent feedback loops, which

we have discussed in Section 5.2. Thus, we also use here an FLD to specify the coordination of both, the reconfiguration process and the self-repair feedback loop (see Figure 45).

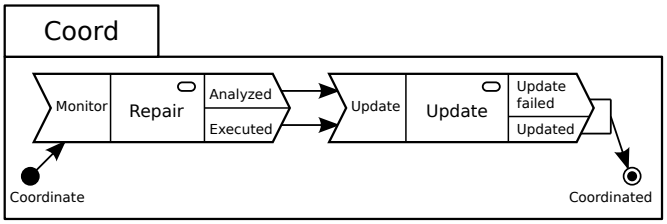
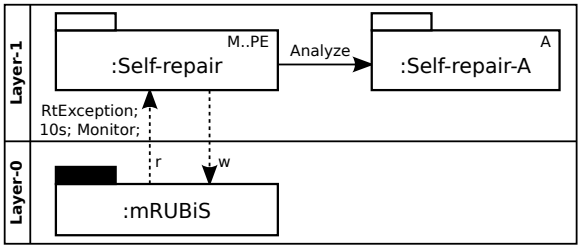


Figure 45: FLD for Coordinating the Self-repair and the Update.

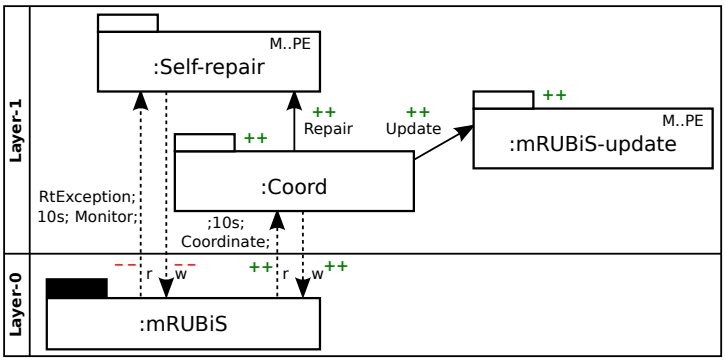
Particularly, the coordination sequences the execution of the self-repair feedback loop and reconfiguration process by invoking them one after the other through the complex model operations Repair and Update. Hence, this coordination module takes over the triggering of the self-repair feedback loop in terms of invoking this loop such that the triggering condition of the loop has to be disabled while the coordination module is deployed. We discuss this required change in the following when outlining how the coordinated execution of the reconfiguration process and feedback loop is enacted on-line.

Therefore, the engineer has to specify an adaptation rule describing how instances of the FLDs mRUBiS-update (see Figure 44) and Coord (see Figure 45) should be integrated as megamodel modules into the adaptation engine. The corresponding rule is depicted in Figure 46b. It changes the current layered architecture of the self-adaptive system shown in Figure 46a to the target architecture shown in Figure 46c to perform the coordinated reconfiguration. The rule matches the :mRUBiS and :Self-repair modules as well as the sense and effect relationships between these modules. All the other elements of the rule are annotated with ++ such that they will be added to the architecture and do not have to be matched. Prior to this, the elements annotated with -- are removed from the architecture. Thus, the sense and effect relationships between the :mRUBiS and :Self-repair modules are removed, which disconnects the self-repair feedback loop from mRUBiS. Thereby, the triggering condition attached to the sense relationship is removed as well such that the self-repair feedback loop will no longer be directly triggered by events emitted from mRUBiS. However, it will be triggered by invocations from the :Coord module that is added by the adaptation rule. More precisely, based on the rule the EUREMA interpreter instantiates the Coord and mRUBiS-update FLDs and adds the corresponding megamodel modules to the architecture. Moreover, it wires these modules to the beforehand existing :mRUBiS and :Self-repair modules. First, the complex model operations Repair and Update of the :Coord module are bound respectively to the :Self-repair and :mRUBiS-update modules such that the :Coord module can invoke these modules. Then, the sense and effect relationships are established between the :Coord and :mRUBiS modules, which also enables the triggering condition of the :Coord module. This module is time-triggered to execute the off-line adaptation without depending on any event emitted by the :mRUBiS module.

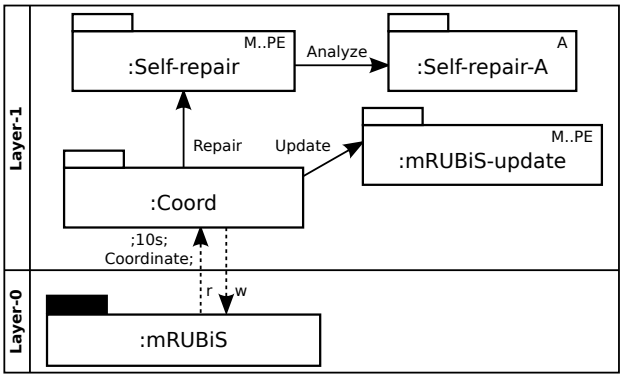
The :Coord module starts execution in its initial state Coordinate and it invokes the :Self-repair module and afterwards the :mRUBiS-update module (*cf.* FLD in Figure 45). Thus, the self-repair feedback loop and the reconfiguration process are sequentially executed based on the coordination of the :Coord module. This synchronization avoids any interferences due to concurrent executions of adaptations to mRUBiS. Moreover, executing the



(a) LD Before and After the Off-line Adaptation.



(b) Adaptation Rule to Adjust the LD.



(c) LD During Executing the Off-line Adaptation.

Figure 46: An Off-line Adaptation Updating the Adaptable Software.

self-repair feedback loop before the reconfiguration process makes it less likely that failures occur in between such that the process may reconfigure a repaired mRUBiS.

Similar to the other maintenance scenarios, to enact the off-line adaptation, the engineer loads up the modeled FLDs (see Figures 44 and 45) and the adaptation rule (see Figure 46b) to the EUREMA interpreter. The interpreter ensure a quiescent state and then applies the rule as discussed previously, which changes the layered architecture and enables the coordinated execution of the self-repair feedback and reconfiguration process. Thus, the EUREMA interpreter enacts the adaptation of mRUBiS that has been developed off-line by engineers and consequently finishes the maintenance cycle.

In this example, the `:mRUBiS-update` module realizing the enactment of the off-line adaptation remains in the adaptation engine until the engineer removes it in the context of another off-line adaptation. Therefore, the engineer observes the EUREMA models and the contained runtime models provided by the maintenance interface of the EUREMA interpreter to get feedback on the reconfiguration process and its success.

When the reconfiguration process has successfully replaced the desired component in mRUBiS, the engineer removes the :Coord and :mRUBiS-update modules from the adaptation engine by conducting another off-line adaptation cycle. Particularly, she specifies and loads up an adaptation rule that reverts the changes of the layered architecture done in the context of the previous off-line adaptation. The corresponding rule is shown in Figure 47. This rule removes the :Coord and :mRUBiS-update modules as well as their wiring to the other modules, and it establishes the original sense and effect relationships between the :Self-repair and the :mRUBiS modules. Applying this rule at runtime reverts the current layered architecture (*cf.* Figure 46c) to the original one (*cf.* Figure 46a). The only difference is that mRUBiS is now equipped with the updated component.

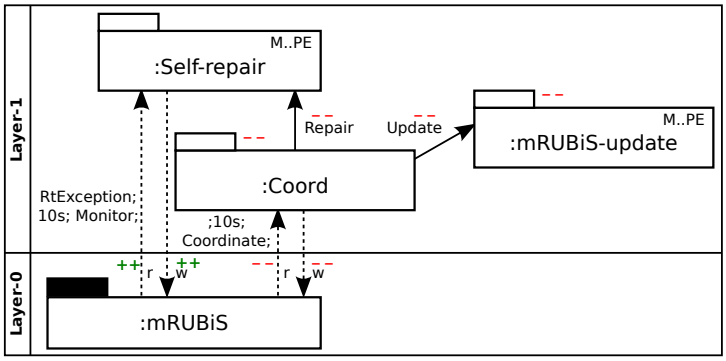


Figure 47: Adaptation Rule to Adjust the LD after the Update.

This maintenance scenario, which we discuss as the last one in this thesis, has shown how the adaptable software can be maintained after the initial deployment of the self-adaptive software while the other scenarios address the maintenance of the adaptation engine.

As illustrated by all scenarios for maintaining self-adaptive software, EUREMA supports off-line adaptation to evolve the software, especially in a way that has not been anticipated when initially deploying the software. Thereby, the same language concepts of EUREMA to model single and multiple feedback loops in layered architectures are used to specify an off-line adaptation and its enactment to the running software. Thus, having specified the self-adaptive software with EUREMA during the design and development, no other language concepts are needed to address the maintenance of the software. This is possible since we consider maintenance processes as feedback loops (*cf.* [272, 274]) that are split into on-line and off-line activities while the on-line part is modeled with the EUREMA language and executed by the EUREMA interpreter. The EUREMA models and the contained runtime models, some of which are created by on-line monitoring, can be used off-line by engineers to analyze and plan an adaptation. The planned adaptation is modeled and developed off-line with EUREMA and finally integrated into the running system to be executed on-line. This on-line execution is coordinated with the feedback loops running in the system and performing self-adaptation. Moreover, off-line adaptation is enabled by EUREMA's support for layered architectures in which feedback loops can be dynamically (un)loaded as megamodel modules at various layers and in which such modules can be executed in coordination with each other. Technically, this is achieved by using the LD as a runtime model that specifies and executes the self-adaptive software and that can be dynamically and directly changed to adapt the software. Together with the maintenance interface provided by the interpreter to exchange models between the runtime and development environments, this supports the co-existence of self-adaptation and maintenance.

6 EXECUTION

Having discussed the use of the EUREMA modeling language to specify feedback loops in self-adaptive software in the previous chapter, we now discuss the execution of the corresponding models. At first, we discuss our motivation for an interpreter-based instead of a generative approach for the execution (see Section 6.1). Then, we discuss the execution semantics by first sketching EUREMA's general model of computation (see Section 6.2) that we refine subsequently. Particularly, we discuss the semantics for executing concurrent megamodel modules (*i. e.*, concurrently running feedback loops) in Section 6.3, for executing a single megamodel module (*i. e.*, a single feedback loop) in Section 6.4, and for the interactions between multiple modules in Section 6.5. Based on this semantics, we discuss in Section 6.6 how EUREMA addresses safe adaptations of feedback loops and adaptation engines by supporting two notions of quiescence, one for adapting FLD models (*i. e.*, the feedback loops) and one for adapting LD models (*i. e.*, the adaptation engine). Finally, we discuss the requirements that EUREMA imposes on the implementations of model operations such that they comply with the EUREMA execution semantics (see Section 6.7).

The execution semantics we discuss in this chapter generally defines how FLD models are executed with the help of the LD model—regardless of how these models are used, for instance, to specify a single feedback loop (*cf.* Section 5.1.2), multiple feedback loops (*cf.* Section 5.2), layered feedback loops (*cf.* Section 5.3), or off-line adaptation (*cf.* Section 5.4).

6.1 INTERPRETER VS. CODE GENERATOR

We designed EUREMA to be an *executable* DSL since we aim for supporting the engineering as well as the *execution* of feedback loops in self-adaptive software. Hence, EUREMA models capture structural as well as behavioral aspects. Especially, the former aspects are addressed by LDs and the latter by FLDs. Besides designing an executable DSL, means to implement and to support the execution are required.

In general, interpretation and code generation are two major approaches to implement executable DSLs [136], especially if the language has been newly invented and if the models should be subject to "domain-specific analysis, verification, optimization, parallelization, and transformation" [302, p. 331]. In this case, the domain concepts should be first-class entities of the language and not be embedded and mixed with general concepts of a GPL to enable analysis, verification and so on at the level of the domain concepts.

This applies well to the EUREMA language. On the one hand, we created a novel language to specifically address the domain of self-adaptive software and feedback loops, which cannot be easily achieved by typical general-purpose software engineering paradigms and languages (*cf.* [386]). On the other hand, we want to enable engineers to specify, analyze, optimize, and adapt (transform) feedback loops by using and focusing only on domain-specific concepts such as MAPE-K, runtime models, and layered architectures.[1] Consequently, we may choose an interpretation or code generation approach.

These two approaches are briefly discussed by Mernik et al. [302]. An interpreter parses an executable model to identify, decode, and execute the DSL constructs. Instead of directly

1 A discussion of EUREMA from a language engineering perspective will be given in Chapter 9.

executing the constructs, a generator translates them to constructs of an existing GPL that is used for the execution. Hence, a generator approach introduces at least one additional step, the code generation, compared to the interpreter approach. According to Mernik et al. [302] and Voelter [425], an interpreter approach is suitable if dynamic changes are required because it supports controlling the execution in a flexible manner. In contrast, a generator approach is less flexible at runtime but may achieve a better execution performance.

We adopt an interpreter approach for EUREMA since dynamic changes are inherent in self-adaptive software. For instance, we discussed dynamic changes of feedback loops in layered architectures and by off-line adaptation in Sections 5.3 and 5.4. An interpreter approach eases supporting such dynamic changes as it dynamically identifies, decodes, and executes the DSL constructs in the model without requiring a static representation of the feedback loop at the code level. Thus, an interpreter avoids the re-generation including potential re-compilation and re-packaging steps of the changed feedback loops. This allows a seamless integration of dynamic changes into the execution of feedback loops.

In general, an interpreter realizes the execution semantics of a modeling language, that is, how the models expressed in this language are executed (*cf.* Section 2.1.1). Consequently, the EUREMA interpreter realizes the execution semantics of the EUREMA language. However, the execution semantics remain *implicit* if it is only specified within the implementation of an interpreter [94]. Such implicit semantics can only be accessed by testing models expressed in the corresponding language using the interpreter [42]. This deters engineers from understanding the language and how the corresponding models are executed. Therefore, we additionally aim for an *explicit* specification of the execution semantics as it is implemented in the interpreter. This explicit specification is discussed in the following.

6.2 SEMANTICS AND MODEL OF COMPUTATION

There exists different means to specify the (execution) semantics of a modeling language although there is no commonly used formalism for such a specification, which makes the specification challenging. The different means are operational, translational, and denotational semantics that can be specified formally or informally (*cf.* Section 2.1.1).

In this section, we precisely discuss the execution semantics of EUREMA using an operational approach. That is, we discuss the computations that are performed when executing the individual concepts of the EUREMA language. This discussion will be rather informal to ease readability and understandability. According to Selic [383, p. 316], such "[i]nformal but structured and precise specifications of semantics are best suited for human consumption". For instance, Harel and Naamad [206] apply such an approach to describe the semantics of statecharts. Additionally, we will discuss a more formal specification that is based on graph transformations and that can be consumed by a computer in Appendix B.

Specifying the execution semantics of a modeling language requires selecting a *model of computation*, that is, a paradigm of how to execute the models expressed in the language [383]. Selic [383] discusses two general paradigms. The first paradigm are behavior-dominant models that consider stateless computational elements consuming some input and producing some output while these elements are combined by control flow or data flow relationships. The second paradigm are structure-dominant models that consider computational elements that are potentially stateful and structured in a network within which these elements collaborate and exchange information. In practice, these two paradigms are often mixed (*e. g.*, in object-oriented computing the network of objects is structure dominant while the implementation of the objects' methods is behavior dominant) [383].

In fact, EUREMA combines as well these two paradigms. The FLDs are behavior dominant as they capture a flow of stateless operations (*i. e.*, adaptation activities) that form a feedback loop and that have runtime models as input and output. The necessary state of the feedback loop such as working data is captured externally to the operations in the runtime models. The operations are connected to each other by control flow relationships and the operations and models are connected by data flow relationships. In contrast, the LDs are structure dominant as they describe a structural network of modules encapsulating FLD instances. Such a module is typically stateful as it encapsulates the runtime models used within the feedback loop. Moreover, these modules collaborate in terms of using, sensing, or effecting each other, which may involve the sharing of runtime models.

Based on these two paradigms, we consider self-adaptive software as reactive systems, that is, the feedback loops react to stimuli. Particularly, modules are triggered by events according to the specification of triggering conditions in the structure-dominant LD. When a module is triggered, the behavior-dominant FLD instance that is encapsulated in the module is executed. Consequently, the LD determines the triggering of modules taking the use, sense, and effect relationships into account while the FLDs determine the behavior of the individual modules. More generally, the LD provides the required structural information and the FLDs provide the required behavioral information to execute the feedback loops.

In the following, we discuss in detail the execution semantics of EUREMA, that is, how EUREMA models as modeled with the LDs and FLDs are executed. For this purpose, we refine the two general paradigms of structure- and behavior-dominant models and discuss how the individual elements of the EUREMA language are executed as well as their interplay to achieve the execution an adaptation engine with multiple feedback loops.

6.3 EXECUTING CONCURRENT MEGAMODEL MODULES (FEEDBACK LOOPS)

If we employ multiple feedback loops in a self-adaptive software, they may run concurrently. We discussed the modeling of such concurrent feedback loops in Section 5.2.1. The corresponding scheme for two feedback loops is shown in Figure 48. This LD specifies that two megamodel modules (*i. e.*, two feedback loops) are located at the same layer and both are sensing and effecting the underlying adaptable software. Each megamodel module has its own trigger such that each of them is executed independently of the other one. If both triggers are activated at the same time, the different megamodel modules run concurrently.

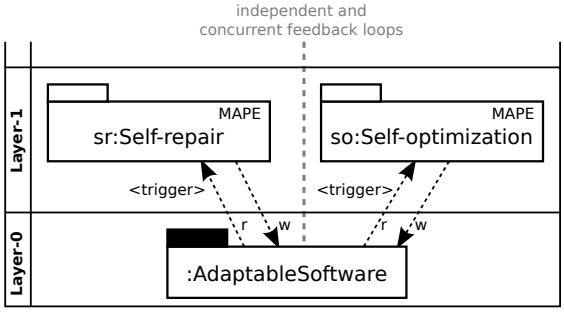


Figure 48: LD Scheme for Independent and Concurrent Feedback Loops.

EUREMA supports the concurrent execution of feedback loops if they are independent of each other. Conceptually, there must not be any interferences between the feedback loops, which otherwise would require coordination. Technically and in terms of EUREMA, mod-

ules that belong to different feedback loops must not be directly or indirectly interrelated by use, sense, or effect relationships in the LD—except of sense and effect relationships to the software modules representing the common adaptable software. This is illustrated by the dashed line in Figure 48 that clearly separates both feedback loops at the upper layers.

Consequently, both feedback loops do not share any behavior or runtime models, that is, modules belonging to different feedback loops do not invoke a shared megamodel module realizing a fragment of a feedback loop, and thus, they also do not pass runtime models as parameters of invocations to other modules. This would otherwise constitute a dependency between the feedback loops. Such dependencies of invoking shared modules or even invoking each other's modules are visible in the LD by means of use relationships that bind modules as targets of invocations (*cf.* Section 5.1.4). Thus, modules of different feedback loops must not be connected directly and indirectly by use relationships in the LD.

Moreover, concurrent megamodel modules as shown in Figure 48 must not be controlled by a common higher-layer megamodel module. Such a higher-layer module would sense and effect both of the concurrent modules (*cf.* Section 5.3), which introduces a dependency between them since the higher-layer module would have to coordinate both concurrent modules for safe adaptations. A common higher-layer module would indirectly connect the modules of the different concurrent feedback loops by sense and effect relationships.

Therefore, we may conclude that the execution of a feedback loop is independent of the execution of another feedback loop if both of them are specified as concurrent feedback loops following the scheme in Figure 48. In other words, concurrently executing multiple megamodel modules does not impact the execution semantics of the individual modules. Thus, we can define the execution semantics of a single feedback loop (*i. e.*, megamodel module) independent of the other feedback loops that run concurrently in the adaptation engine. Thereby, the execution of each megamodel module follows the same semantics.

6.4 EXECUTING A SINGLE MEGAMODEL MODULE (FEEDBACK LOOP)

In the following, we describe the execution semantics of a single megamodel module the realizes a feedback loop. In the course of the discussion, we also consider modular feedback loops, that is, when multiple modules realize a feedback loop. Therefore, we first describe the interface of a megamodel module to other modules (see Section 6.4.1), then triggers for executing a megamodel module (see Section 6.4.2) as well as the execution of the FLD instance encapsulated in the triggered module (see Section 6.4.3). Considering the latter aspect, we further discuss the maintenance of execution information (see Section 6.4.4) and runtime models (see Section 6.4.5) while executing an FLD instance.

6.4.1 *Interface of a Megamodule Module*

To discuss the execution semantics of a megamodel module, we first describe the interface of such a module. This interface shows the possibilities of how the module can be combined with other modules in an LD, which influences the execution of the module.

Figure 49 shows an LD fragment with a megamodel module and its potential relationships to other modules. The gray text gives these relationships an easily accessible meaning from the perspective of the depicted module. Assuming that the megamodel module fl:FeedbackLoop realizes a feedback loop, it *senses* and *effects* at least one other module that is located at the underlying layer. This underlying module can be a megamodel module realizing another feedback loop, or a software module representing the adaptable software.

In the former case, the module fl:FeedbackLoop is located at a higher layer, in the latter case at the lowest layer of the adaptation engine. A megamodel module may have at most one *trigger* that activates the module. Since we consider reactive feedback loops, the execution of the (sensing) module depends on events that are emitted by the sensed module. Thus, the trigger is annotated to a sense relationship along which the events can be observed.

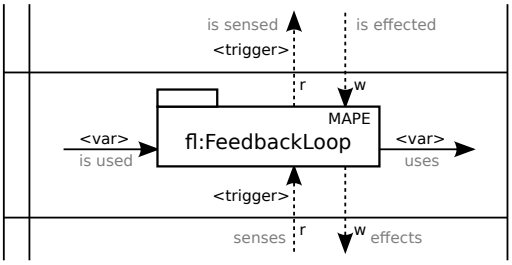


Figure 49: LD Fragment Showing a Megamodel Module and its Relationships to other Modules.

Similar to sensing and effecting a module, a megamodel module such as fl:FeedbackLoop itself can be *sensed* and *effected* by other megamodel modules that are located at the layer on top of it. These higher-layer modules require as well a *trigger* that activates them based on events emitted by the sensed module such as the fl:FeedbackLoop module in this case.

Moreover, a megamodel module *uses* other modules that are located at the same layer because of two reasons. First, the module contains a complex model operation that synchronously invokes another megamodel module when being executed. Second, the module uses software modules that are the implementations of its basic model operations. Executing a basic model operation, the corresponding implementation is synchronously invoked. In both cases, naming the operation in the FLD declares a variable *var*. The variable and the corresponding operation are then bound to the invoked module by the *use* relationship labeled with *var* in the LD. Thus, both cases describe the synchronous integration of behavior provided by other modules into the executing megamodel module. This supports modular feedback loops (*cf.* Section 5.1.4). The execution semantics of basic and complex model operations will be discussed in the context of executing an FLD instance in Section 6.4.3.

Similar to using another megamodel module, a megamodel module itself can be *used*. In this case, the used module describes a feedback loop fragment that is synchronously invoked and integrated into an overall feedback loop. A particular issue here is that a megamodel module must not be used and have a trigger at the same time. Otherwise, the module might be invoked and triggered concurrently, which is not supported. Megamodel modules are not reentrant such that at most one single thread of control is active within the module. Hence, the execution of a megamodel module can be initiated either by a trigger or by an invocation within the execution context of another module. The triggering of modules is discussed in the following and the invocation of modules in Section 6.4.3.

6.4.2 *Triggering the Execution of a Megamodel Module*

EUREMA supports event-based and timed triggering of megamodel modules. Event-based triggering addresses the reactive execution of feedback loops initiated by occurrences of events. Timed triggering enables the periodical execution, which does not depend on the occurrences of events and can therefore support proactive approaches to self-adaptation.

Both types of triggers can be composed such that the initiation of the execution depends on the occurrences of events and on the elapsing of time between two consecutive executions.

The specifics and semantics of a trigger varies with respect to the layer in which the megamodel module to be triggered is located. We distinguish two cases here:(1) The module is located at the lowest layer of the adaptation engine and directly controlling the adaptable software. (2) The module is located at a higher layer of the engine and controlling another megamodel module (*i.e.*, feedback loop) at the underlying layer. The variations concern the synchronous or asynchronous triggering and whether timed triggering is allowed.

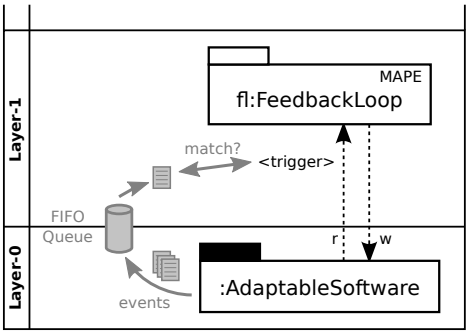


Figure 50: Trigger for Megamodel Modules at the Lowest Layer of the Adaptation Engine.

(1) Megamodel Modules at the Lowest Layer of the Adaptation Engine. The triggering of such a module is illustrated in Figure 50. The megamodel module fl:FeedbackLoop directly senses and effects the :AdaptableSoftware. A *trigger* is figuratively attached to the sensing relationship that reveals a flow of events from the adaptable software to the feedback loop. Such a trigger is expressed in a textual language (that is defined in Section A.4) and it specifies three aspects: <events>, <period>, and an <initialState>.

1. The <events> specify a comma-separated list of at least one event that is potentially emitted by the adaptable software. The actual runtime occurrence of such an event activates the trigger and thus the execution of the megamodel module. In the following, we call events that are specified in a trigger as *trigger events* and events that actually occur at runtime as *runtime events*.

   An event of both kinds is defined by an optional name and a mandatory type in a hierarchy of event types. This event type hierarchy does not support multiple inheritance such that each event type either has no or at most one super type.

   The scheme and examples of a trigger event are shown in Table 3. An event is defined by its type such as RtException and an optional name such as Permission denied. If the name is omitted, the event is anonymous and only characterized by the type. The type hierarchy is reflected by fully qualified types such as Exception::RtException where Exception is the super type of RtException. The depth of the type hierarchy is not restricted, which results in a scheme of arbitrary length Type:: ... Type::Type[Name].

2. A <period> defines the time in seconds that should elapse between two consecutive runs of the module, which allows the adaptation performed by the last run to take effect in the adaptable software before starting the next run (*cf.* settling time [215]).

3. Finally, the <initialState> refers by name to the initial operation in which the triggered megamodel module should start execution. This initial operation is the entry point for executing the FLD instance encapsulated in the module.

Table 3: Overview of Trigger Events.

Scheme	Example
Type[Name]	RtException[Permission denied]
Type	RtException
SuperType::Type	Exception::RtException
SuperType::Type[Name]	Exception::RtException[Permission denied]

Based on these aspects, there exist three types of triggers for megamodel modules (see Table 4). An event- and time-based trigger defines events that must occur at runtime as well as a period that must elapse between two consecutive runs of the module. An event-based trigger omits specifying a period such that only the occurrence of events is required. A time-based trigger only specifies a period, which results in periodically executing the module. Each trigger must specify the initial state (operation) for starting the execution.

Table 4: Overview of Trigger Types.

Type	Scheme
Event- and Time-Based Trigger	<events>;<period>;<initialState>;
Event-Based Trigger	<events>; ;<initialState>;
Time-Based Trigger	;<period>;<initialState>;

In the following, we first discuss event- and time-based triggers and then the other two types of triggers. At runtime, the EUREMA interpreter uses a trigger to determine when a megamodel module should be executed. This is illustrated in Figure 50. The module fl:FeedbackLoop senses and effects the :AdaptableSoftware and it will be activated based on the trigger attached to the sensing relationship that reveals a flow of runtime events from the software to the module. The adaptable software emits runtime events to a First-In, First-Out (FIFO) queue provided by the EUREMA interpreter. This queue decouples the adaptable software and the feedback loops such that the interpreter can asynchronously process these events. Thus, the adaptable software is not blocked while the interpreter evaluates the triggers and executes the feedback loops. The software can continue executing its domain logic and providing service to users or other systems. A feedback loop only disturbs the execution of the software when it effects the software by enacting an adaptation, which might additionally require quiescence of (parts of) the software [259, 20].

To process the runtime events from the queue (see Figure 50), the EUREMA interpreter repeats the following behavior consisting of multiple steps that are performed sequentially:

$Step_a$ If the queue is empty, wait until the adaptable software emits a runtime event to the queue. Otherwise, continue with the next step.

$Step_b$ Consume a single runtime event from the FIFO queue.

$Step_c$ Match the trigger events against the consumed runtime event. A successful match of the trigger events and a runtime event is determined by the following aspects:

- At least one trigger event matches successfully the runtime event. Trigger events are combined by logical disjunction and as soon as a successful match has been identified, the remaining trigger events, which have not been tested yet for a match, do not have to be tested.

- A single trigger event matches the runtime event if:
  - The type of the runtime event is the same or a subtype of the trigger event type.
  - Both events have the same name. However, if the trigger event is anonymous (*i. e.*, it does not define a name), it does not further restrict the matching such that only the previous condition on event types must hold.

  If a successful match has been found, go to $Step_d$, otherwise to $Step_f$.

$Step_d$ Execute the megamodel module, that is, the FLD instance encapsulated in the module. This step will be discussed in detail in Section 6.4.3.

$Step_e$ Wait until the period has expired.

$Step_f$ Continue with $Step_a$.

Performing these steps sequentially and considering an event- and time-based trigger, Figure 51 shows an example trace of the interpreter's behavior of evaluating such a trigger and executing the corresponding module. The adaptable software continuously adds runtime events, e_1 to e_5 in this example, to the queue. The EUREMA interpreter consumes these events one after the other in the same order as they have been added to the queue (FIFO). For the example trace, we assume that emitting and adding runtime events to the queue, consuming such events from the queue, and matching of trigger events against a runtime event do not take any time—although they consume a negligible amount of time in reality—in contrast to executing a megamodel module and waiting for a runtime event to occur or for a period to elapse. Consequently, we consider a logical time scale in Figure 51.

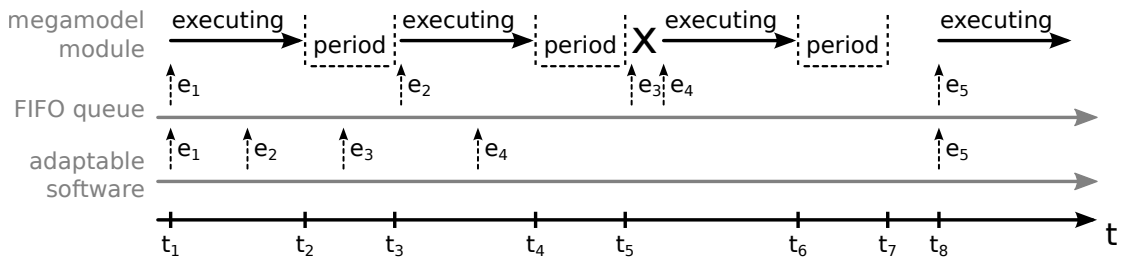


Figure 51: Example Trace for an Event- and Time-Triggered Megamodel Module at Layer 1.

We assume that the queue is initially empty. At t_1, the interpreter consumes the runtime event e_1 ($Step_b$) as soon as e_1 has been added to the queue, and matches the trigger events against e_1 ($Step_c$). We assume a successful match of the events such that the interpreter is executing the module ($Step_d$). Having finished this execution at t_2, the interpreter is waiting until the period defined in the trigger has passed at t_3 ($Step_e$). While the interpreter is executing the module or waiting for the period to elapse (from t_1 to t_3), the adaptable software adds e_2 and e_3 to the queue. At t_3, the interpreter repeats its behavior ($Step_f$).

Thus, it consumes the first event from the queue ($Step_b$), which is e_2, and matches the trigger events against e_2 ($Step_c$). Assuming a successful match, the interpreter is executing the module ($Step_d$) until t_4. After that, the interpreter waits until the period has expired ($Step_e$). This happens at t_5 and the interpreter repeats its behavior ($Step_f$). While the module was running, the adaptable software has emitted e_4. At t_5, the interpreter consumes e_3 ($Step_b$). However, it cannot match successfully the trigger events against e_3 ($Step_c$), which is denoted by the cross in Figure 51. Consequently, it does not execute the module but continues ($Step_f$) with consuming the next event, which is e_4 ($Step_b$). The trigger events are

successfully matched against e_4 ($Step_c$) such that the interpreter is executing the module ($Step_d$) until t_6 followed by waiting for the period to expire ($Step_e$). This happens at t_7 and the interpreter repeats its behavior ($Step_f$) and tries to consume an event from the queue.

However, the queue is empty such that the interpreter is waiting until the adaptable software adds the next event to the queue ($Step_a$), which happens for e_5 at t_8. The interpreter immediately consumes e_5 ($Step_b$) and matches the trigger event against e_5 ($Step_c$)—in this case successfully such that it is executing again the module ($Step_d$).

This example trace is similar for an event-based trigger, that is, a trigger that does not specify any period. Consequently, the interpreter skips $Step_e$, that is, it does not have to wait until the period expires after executing the module and before consuming the next event from the queue. This results in a corresponding example trace as shown in Figure 52.

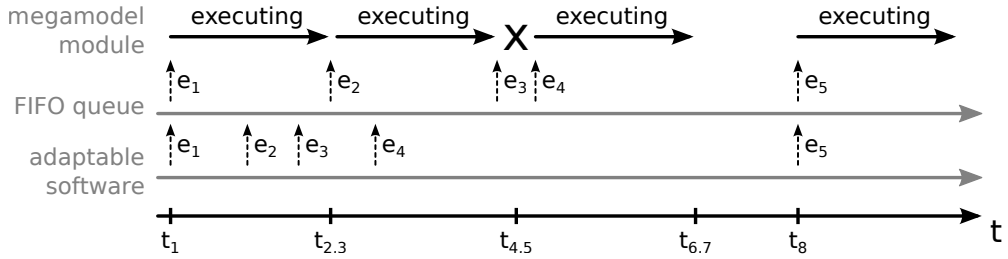


Figure 52: Example Trace for an Event-Triggered Megamodel Module at Layer 1.

This trace describes the same scenario as the previous one. Skipping $Step_e$, the interpreter immediately re-executes the module as long as matching runtime events are available in the queue (see the first three executions of the module in Figure 52). Otherwise, the interpreter is waiting for new runtime events and their successful matching to start the next execution of the module (see the fourth execution of the module in Figure 52).

Finally, a time-based trigger does not specify any trigger events but only a period. Thus, the triggering does not depend on any runtime events but only on the time between two consecutive executions of the module. Consequently, the interpreter does not have to process any runtime events such that it skips the steps of waiting for runtime events ($Step_a$), consuming runtime events ($Step_b$), and matching trigger events against runtime events ($Step_c$). In contrast, the interpreter just executes the module ($Step_d$) and then waits for the period to expire ($Step_e$). The interpreter continuously loops these two steps ($Step_f$). A corresponding example trace for the given scenario is shown in Figure 53. Since the interpreter does not process any runtime events, we do not require a queue for timed-based triggers.

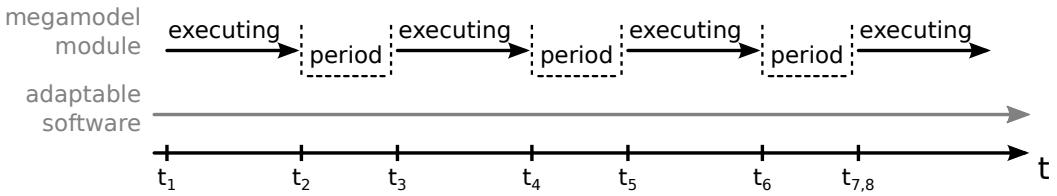


Figure 53: Example Trace for a Time-Triggered Megamodel Module at Layer 1.

The examples traces shown in Figures 51-53 illustrate that a megamodel module is not re-entrant such that there are no multiple overlapping executions of the same module. The interpreter runs the executions of the same module consecutively by queuing up the runtime events and considering the period as a break between two consecutive runs.

So far we have discussed the case of triggering megamodel modules that are located at the lowest layer of the adaptation engine. In the following, we discuss the other case of triggering megamodel modules that are located at higher layers of the engine.

(2) Megamodel Modules at a Higher Layer of the Adaptation Engine. The triggering of such megamodel modules is different than of modules at the lowest layer of the engine. The reason is that higher-layer modules sense and effect other megamodel modules—located at the adjacently lower layer—and not the adaptable software. This is illustrated in Figure 54 showing two megamodel modules that are located relatively to each other at adjacent layers of the engine: the fl:FeedbackLoop at the adjacently higher and the al:AdaptableLoop at the adjacently lower layer. The former module senses and effects the latter module and each of them realizes a feedback loop. In the following, we discuss the triggering of the module fl:FeedbackLoop based on a *trigger* that is attached to the sensing relationships that reveals a flow of runtime events from the al:AdaptableLoop to the fl:FeedbackLoop module.

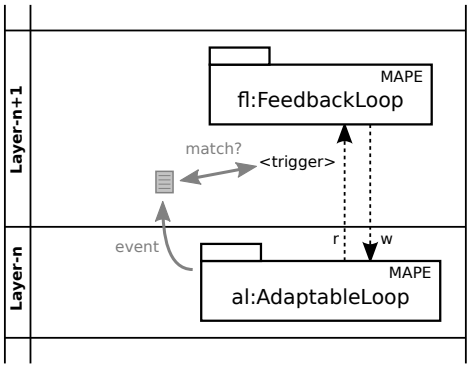


Figure 54: Trigger for Megamodel Modules at Higher Layers of the Adaptation Engine.

In absolute terms, the al:AdaptableLoop module can be located at the lowest layer of the adaptation engine. The triggering of such modules that sense and effect the adaptable software has been discussed previously. However, the module can also be located at a higher layer of the engine such that it senses and effects another megamodel module. In this case, the al:AdaptableLoop module is triggered similarly to the fl:FeedbackLoop module.

A trigger for a higher-layer megamodel module is expressed in a textual language (that is defined in Section A.4). It describes two aspects: <euremaEvents> and <initialState>.

1. The <euremaEvents> specify a comma-separated list of at least one event that is potentially emitted by the EUREMA interpreter when executing the FLD instance encapsulated in the sensed lower-layer megamodel module. The actual runtime occurrence of such an event activates the trigger and thus the execution of the higher-layer megamodel module. As done previously, we call events that are specified in a trigger as *trigger events* and events that actually occur at runtime as *runtime events*.

   Trigger and runtime events are defined by a mandatory name and type. The name corresponds to the name of an operation or transition contained in the lower-layer module. Hence, an event references an operation or transition by name and is thus defined by the engineer. The types of events are pre-defined and listed in Table 5.

   According to such trigger events specified by engineers, the higher-layer megamodel module is triggered when the interpreter emits corresponding runtime events while executing the FLD instance encapsulated in the lower-layer megamodel module.

Table 5: Overview of Event Types Predefined by EUREMA.

Scheme	Description
Before[opName]	An event of type Before is emitted *before* an operation is executed. The name of the event (opName) is the name of the operation executed next.
After[opName]	An event of type After is emitted *after* an operation has been executed. The name of the event (opName) is the name of the executed operation.
OnTransition[tName]	An event of type OnTransition is emitted when a transition (*i. e.*, a control flow link) between two operations is executed. The name of the event (tName) is the name of the currently executed transition. Since transition names are not represented in the concrete syntax (*i. e.*, the FLDs), the name of the operation's exit compartment that is the unique source end of the executed transition can be used instead (*cf.* Section A.4).

2. The <initialState> refers by name to the initial operation in which the triggered higher-layer megamodel module should start execution. This initial operation is the entry point for executing the FLD instance encapsulated in the module.

Both aspects of the trigger are mandatory such that there exists only one type of triggers for higher-layer megamodel modules. Such triggers are evaluated at runtime by the EUREMA interpreter to determine when a higher-layer (sensing and effecting) megamodel module should be executed in the scope of executing a lower-layer (sensed and effected) megamodel module. As shown in Figure 54 on the previous page, we consider the triggering of the higher-layer module fl:FeedbackLoop, for which a trigger is defined and which senses and effects the lower-layer module al:AdaptableLoop.

To trigger and execute a higher-layer megamodel module, the EUREMA interpreter repeats the following behavior. Particularly, it sequentially performs the following steps:

Step$_a$ Execute (or continue executing) the lower-layer megamodel module (*e. g.*, the module al:AdaptableLoop). Along the way, intercept the execution and emit synchronously a corresponding runtime event

- before executing an operation,
- after having executed an operation, and
- when executing a transition (*cf.* Table 5).

Details of this step will be discussed in Section 6.4.3 when describing the execution of an FLD instance encapsulated in a megamodel module that can be located at any layer of the adaptation engine.

Step$_b$ Synchronously process the emitted runtime event by matching the trigger events against it. A successful match of the trigger events and the runtime event is determined by the following aspects:

- At least one trigger event matches successfully the runtime event. Thus, the trigger events are combined by logical disjunction and as soon as a successful match has been identified, the remaining trigger events, which have not been tested yet for a match, do not have to be tested.
- A single trigger event matches the runtime event if both events are of the same type and if both events have the same name.

If a successful match has been found, go to *Step*$_c$, otherwise to *Step*$_d$.

Step$_c$ Execute the megamodel module (*e. g.*, fl:FeedbackLoop) that senses and effects the module whose execution has been intercepted in *Step*$_a$. During this step, the execution of the intercepted module remains intercepted.

Step$_d$ Continue with *Step*$_a$.

Since a runtime event is emitted (*Step*$_a$) and processed (*Step*$_b$ and *Step*$_c$) synchronously, there is no need for a queue storing such events to decouple these steps (*cf.* Figure 54 on Page 114). Particularly, a runtime event notifies about having reached an interception point when executing a megamodel module. As specified by trigger events, such interception points are used for synchronously executing a higher-layer megamodel module. To illustrate the triggering of a higher-layer module, an example trace of the interpreter's behavior is shown in Figure 55. It describes a specific instance of executing a lower-layer megamodel module (*e. g.*, al:AdaptableLoop), which causes the runtime events e_1 to e_6 that are used to trigger the execution of a higher-layer megamodel module (*e. g.*, fl:FeedbackLoop).

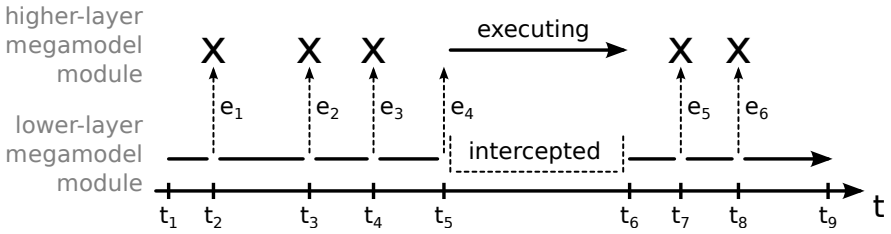


Figure 55: Example Trace for a Triggered Higher-Layer Megamodel Module.

The interpreter starts executing the lower-layer module at t_1. During the execution, it synchronously emits runtime events when it reaches the corresponding points of the executed FLD instance (*cf.* Table 5). At t_2, it interrupts the execution of the lower-layer module and emits e_1 (*Step*$_a$). This event is synchronously processed by the interpreter that matches the trigger events—specified for the higher-layer module—against it (*Step*$_b$). The interpreter cannot successfully match the trigger events against e_1, which is denoted by a cross in Figure 55, such that it goes back (*Step*$_d$) and continues executing the lower-layer module (*Step*$_a$). The interpreter repeats this behavior at t_3 and t_4 when e_2 and e_3 are emitted but not successfully matched. At t_5, it interrupts the execution of the lower-layer module to emit e_4 (*Step*$_a$) and successfully matches the trigger events against e_4 (*Step*$_b$). Thus, it further intercepts the execution of the lower-layer module to execute the triggered higher-layer module (*Step*$_c$). At t_6, the execution of the higher-layer module has finished such that the interpreter goes back (*Step*$_d$) and resumes executing the lower-layer module (*Step*$_a$). While executing this module, it emits further events, e_5 and e_6, which are not matched successfully. Eventually, the execution of the lower-layer module finishes at t_9.

This example trace illustrates that a higher-layer module does not run concurrently with the lower-layer module it senses and effects. The execution of the lower-layer module is intercepted to synchronously execute the higher-layer module. After the higher-layer module has finished execution, the lower-layer module resumes its execution at the point it has been intercepted. In this context, we do not allow the use of a period in a trigger for a higher-layer module. Otherwise, a period might delay the execution of the higher-layer module while the execution of the lower-layer module is intercepted. This might block the whole adaptation engine. This aspect distinguishes triggers for modules at the lowest layer of the adaptation engine from triggers for higher-layer modules since the adaptable software emits events asynchronously while the megamodel modules synchronously.

However, a common aspect is that multiple higher-layer modules that sense and effect the same lower-layer module run concurrently if they are triggered by the same runtime events (*cf.* corresponding discussion of concurrently running megamodel modules located at the lowest layer of the adaptation engine in Section 6.3).

Finally, the triggering of a higher-layer megamodel module by intercepting a lower-layer module through events can be implemented differently and more efficiently. For instance, the interpreter only intercepts a module and emits a corresponding runtime event at those execution points for which trigger events for higher-layer modules are actually specified. Considering Figure 55, this avoids emitting all of the runtime events except of e_4. Moreover, the interception can be even realized without any runtime event. However, for defining the execution semantics, we aim for the same paradigm of event-based triggers as we used for modules located at the lowest layer of the adaptation engine and sensing the adaptable software. This contributes to a unified design of the EUREMA language.

6.4.3 *Executing an FLD Instance Encapsulated in a Megamodel Module*

Having triggered a megamodel module, regardless whether it is located at the lowest or at a higher layer of the adaptation engine, the FLD instance encapsulated in this module will be executed. Such an FLD instance is created once and it is reused if the module is triggered multiple times. However, as discussed in Section 6.4.2, there are no concurrent triggers and thus no concurrent executions of the same module. We discussed the modeling of FLDs in Section 5.1. In the following, we describe the execution semantics of the FLD, that is, how an FLD instance is executed. The executable FLD elements are shown in Figure 56.

In general, an FLD describes a flow of operations while exactly two operations are connected by a transition (*i. e.*, a control flow link). Hence, the EUREMA interpreter starts executing an FLD instance with an initial operation (see Figure 56a) and then alternately executes a transition (see Figure 56f) and an operation (see Figures 56b–56d) until it reaches a final operation (see Figure 56e). If an operation has more than one outgoing transition, the execution of the operation determines exactly one out of all these transitions, which should be taken and hence executed. While executing an FLD instance, the interpreter maintains a *model counter*—similar to a program counter—that points to the currently executed operation or transition. Such a counter is maintained for each FLD instance.[2] The initial operation, in which the interpreter starts executing the FLD instance, is determined by the trigger for the corresponding megamodel module (*cf.* Section 6.4.2). Hence, we may consider the megamodel module and the initial operation as parameters for the interpreter's behavior of executing the FLD instance encapsulated in the module while the return value is the final operation in which the execution terminates. This behavior is defined as follows:

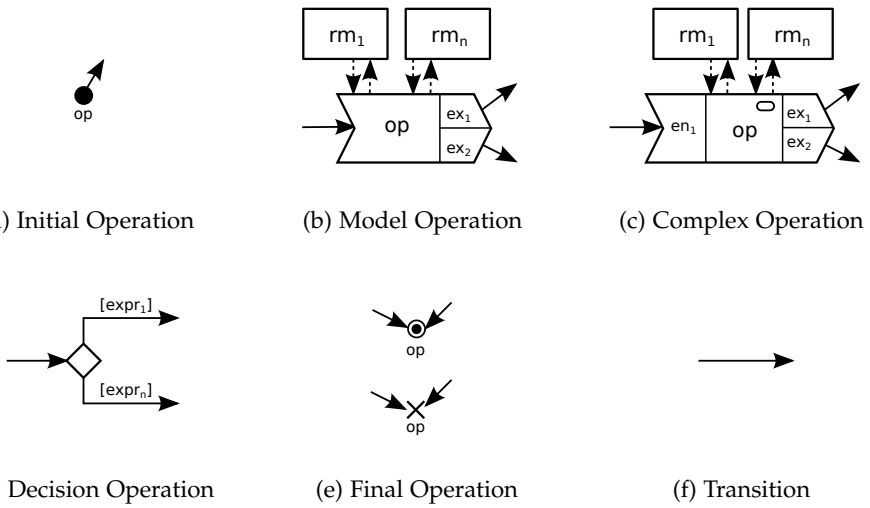
fo = Execute(mm, io)

- Parameters: the megamodel module mm encapsulating the FLD instance that is going to be executed, and the initial operation io in which to start the execution.
- Return value: the final operation fo, in which the execution terminates.

Step$_a$ Obtain the single FLD instance encapsulated in mm and continue with *Step*$_b$ to execute the initial operation io (i.e., op is set to io in *Step*$_b$).

Step$_b$ Execute the operation op as follows:

2 This model counter is captured in the EUREMA language by the relationship named current that points from the megamodel module's execution context to the currently executed operation or transition (*cf.* Section A.1).



(a) Initial Operation (b) Model Operation (c) Complex Operation

(d) Decision Operation (e) Final Operation (f) Transition

Figure 56: Executable FLD Elements.

Step$_{b1}$ Set the modelCounter$_{mm}$ to op and then emit synchronously the runtime event Before[op.name].

Step$_{b2}$ If op is an *initial operation* (see Figure 56a):

- Obtain the single outgoing transition t.

Remark: An initial operation does not contain any behavior as it just defines the entry point of executing the FLD instance. Thus, no behavior is executed. Moreover, it has exactly one outgoing transition[3], that will be executed next.

Step$_{b3}$ If op is a *model operation* (see Figure 56b):

- Retrieve the runtime models $\{rm_i\}$ that are the inputs of op.
- Identify the software module sm to which op is bound (*i. e.*, the implementation of the operation). This binding is defined in the LD by a use relationship pointing from the currently executed megamodel module mm containing op to the software module sm. The relationship is labeled with the name of op.
- Synchronously invoke sm passing $\{rm_i\}$ as a parameter.
- Wait until the execution of sm eventually finishes and returns a status s as well as the runtime models $\{rm_o\}$ that are the outputs of op.
- Identify the exit ex of op with the given name s. An exit of an operation is the unique source end of a single transition. Obtain this transition t for ex.

Remark: A model operation has runtime models as inputs and outputs. The input models capture the working data and may even specify the behavior of the operation. The actual behavior of processing or executing the input models and of producing or updating the output models must be implemented by engineers. Such an implementation is a software module to which the operation is bound. It must realize a prescribed interface, which is used by the interpreter to synchronously invoke the module with the input models as parameters. The invocation returns the output models as well as a status that is mapped to exactly one exit of the executed operation. Thus, the user-defined implementation

3 Such constraints on the EUREMA language are precisely defined on the metamodel in Section A.1.

determines which exit and therefore which outgoing transition of the operation is taken. The output models are maintained by the interpreter such that they can be the inputs of other operations.

Step$_{b4}$ If op is a *complex model operation* (see Figure 56c):

- Identify the megamodel module mm' to which op is bound. This binding is defined in the LD by a use relationship labeled with the name of op and pointing from the currently executed megamodel module mm containing op to the megamodel module mm'.
- Based on the entry en of op that has been taken to enter op, identify the initial operation $\text{io}_{\text{mm'}}$ of mm' whose name equals the name of en.
- Synchronously invoke mm' (*i.e.*, the FLD instance encapsulated in mm') and start executing it with the initial operation $\text{io}_{\text{mm'}}$. That is, perform $\text{Execute}(\text{mm'}, \text{io}_{\text{mm'}})$, which returns $\text{fo}_{\text{mm'}}$, that is, the final operation in which the execution of mm' has terminated.
- Identify the exit ex of op with the name of $\text{fo}_{\text{mm'}}$. An exit of an operation is the unique source end of a single transition. Obtain this transition t for ex.

Remark: A complex model operation as part of an FLD instance synchronously invokes another FLD instance. The taken entry of the complex operation determines the initial operation for starting the execution of the invoked FLD instance. This execution of the invoked FLD instance terminates in a final operation that determines exactly one exit of the complex operation to be activated and hence, the outgoing transition to be executed next in the invoking FLD instance. When invoking another FLD instance, the interpreter dynamically checks whether the entries and exits of the complex operation can be matched by name to the initial and final operations of the invoked instance. If the checks fails, the interpreter throws a runtime exception. To execute an invoked FLD instance, the interpreter performs the same behavior as of executing the invoking instance.

Finally, a complex model operation may have runtime models as inputs. These models are passed lazily as parameters to the complex operation, that is, they are loaded when executing the basic model operations of the invoked FLD instance, which actually use these models.

Step$_{b5}$ If op is a *decision operation* (see Figure 56d):

- Evaluate all expressions of the outgoing transitions of op. An expression evaluates either to true or false.
- Throw a runtime exception if more than one expression evaluated to true.
- If exactly one expression evaluated to true, obtain the corresponding transition t labeled with this expression.
- If no expression evaluated to true, look for a default expression, that is, an ELSE branch. If there is such an ELSE expression, obtain the corresponding transition t. Otherwise, throw a runtime exception.

Remark: A decision operation is used to conditionally branch the control flow between operations. To deterministically determine the branch that should be taken, we require that exactly one of all expressions for a decision operation evaluates to true, or if none of them evaluates to true, a default ELSE branch exists. Otherwise, the interpreter throws a runtime exception.

The expressions are specified with the condition expression language and evaluated by the corresponding parser (*cf.* Section A.3). The EUREMA language integrates this language and the EUREMA interpreter uses the parser to evaluate the expressions. In general, the expression language is exchangeable in EUREMA.

$Step_{b6}$ If op is a *final operation* (see Figure 56e):

- Do nothing.

  Remark: A final operation denotes the termination of executing the FLD instance. There is no behavior associated to a final operation.

  There is a semantic variation distinguishing a regular (depicted by an encircled black circle) and a destructive (depicted by a cross) final operation. Both of them have no associated behavior and just denote the termination of executing an FLD instance. However, the destructive final operation specifies that the FLD instance together with its encapsulating megamodel module should not be executed any more but removed from the adaptation engine if the instance has terminated with such a final operation.

- If the final operation is destructive, remove the megamodel module encapsulating the executed FLD instance from the adaptation engine.

$Step_{b7}$ Emit synchronously the runtime event After[op.name].

$Step_c$ Set $\text{modelCounter}_{\text{mm}}$ to null and return op if op is a final operation. Otherwise, continue with $Step_d$ to execute transition t that has been determined either by $Step_{b2}$, $Step_{b3}$, $Step_{b4}$, or $Step_{b5}$.

Remark: Reaching the final operation, the interpreter has finished executing the FLD instance. It returns this final operation to notify about the status of this execution (in general, an FLD can have multiple final operations). Otherwise, the interpreter will execute the transition determined by the execution of the operation op.

$Step_d$ Execute the transition t (see Figure 56f) as follows:

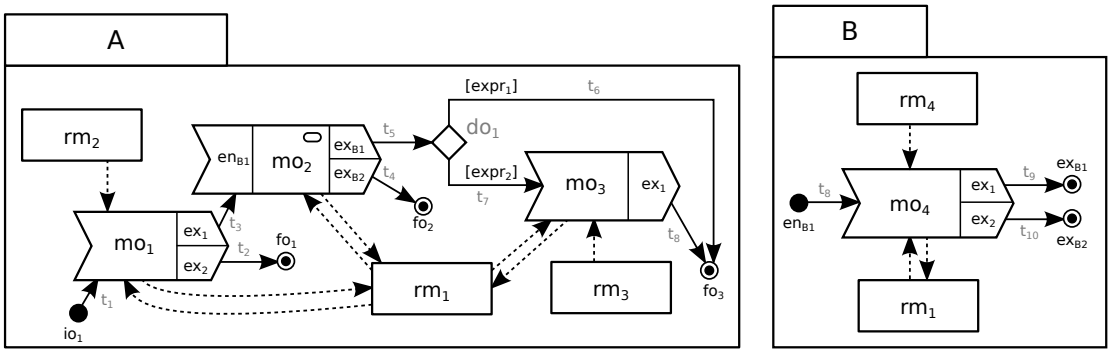
$Step_{d1}$ Set the $\text{modelCounter}_{\text{mm}}$ to t and then emit synchronously the runtime event OnTransition[t.name].

$Step_{d2}$ Obtain the operation op that is the single target end of t. Continue with $Step_b$ to execute op.

Remark: There is no behavior associated to a transition such that only the runtime event is emitted and the operation to be executed next is obtained.

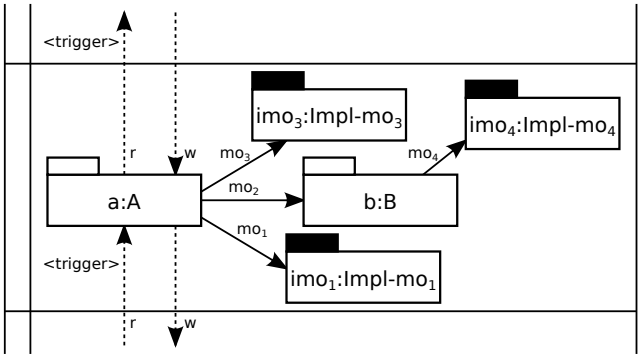
As described by this behavior, the interpreter requires structural information from the LD to execute the FLD instance. Specifically, it requires the bindings of basic and complex model operations to software respectively megamodel modules such that it knows how to actually execute the operations. When executing a basic or complex model operation, the interpreter the either invokes the corresponding software (see $Step_{b3}$) or megamodel module (see $Step_{b4}$). Moreover, when executing a module with its FLD instance, the interpreter synchronously emits runtime events of type Before (see $Step_{b1}$), After (see $Step_{b7}$), and OnTransition (see $Step_{d1}$). Such events denote interceptions points of the currently executed module to trigger another megamodel module that senses and effects the intercepted module—as defined in the LD. This has been discussed in the context of triggering megamodel modules located at higher layers of the adaptation engine in Section 6.4.2. These cases illustrate how the structure-dominant LD and the behavior-dominant FLD complement one another to enable the execution of feedback loops.

Example. In the following, we illustrate the execution of an FLD instance using the generic example shown in Figure 57. The example consists of two FLDs, A and B as shown in Figures 57a and 57b, while A invokes B. Both FLDs are instantiated and encapsulated in megamodel modules that are located at the same but arbitrary layer of the adaptation engine (see LD in Figure 57c). According to the LD, we consider that the megamodel module a:A is triggered such that an instance of the FLD A is going to be executed. We assume that the trigger defines io_1 as the initial operation to start execution with.



(a) FLD A.

(b) FLD B.



(c) LD.

Figure 57: Example to Illustrate the Execution of FLD Instances.

Thus, the interpreter performs the behavior Execute(a:A, io_1) as specified above. To illustrate the execution semantics of FLDs, we now describe succinctly a scenario of the interpreter's behavior executing the megamodel module a:A. This scenario is further visualized by the sequence diagram in Figure 58, which focuses on the major steps performed by the interpreter. These steps are enumerated in the following text and in the sequence diagram to ease the mapping between the text and the diagram. The scenario is as follows:

(1) $Step_a$ The interpreter obtains the FLD instance encapsulated in the module a:A as well as the initial operation io_1 to be executed.

(2) $Step_b$ It executes io_1 by setting the $modelCounter_{a:A}$ to io_1, emitting the runtime event Before[io_1.name] ($Step_{b1}$), obtaining the single outgoing transition t_1 to be executed next ($Step_{b2}$), and emitting the runtime event After[io_1.name] ($Step_{b7}$).

(3) $Step_c$ It continues execution since io_1 is not a final operation.

(4) $Step_d$ It executes t_1 by setting the $modelCounter_{a:A}$ to t_1, emitting the runtime event OnTransition[t_1.name] ($Step_{d1}$), and obtaining the target of t_1, which is mo_1 ($Step_{d2}$).

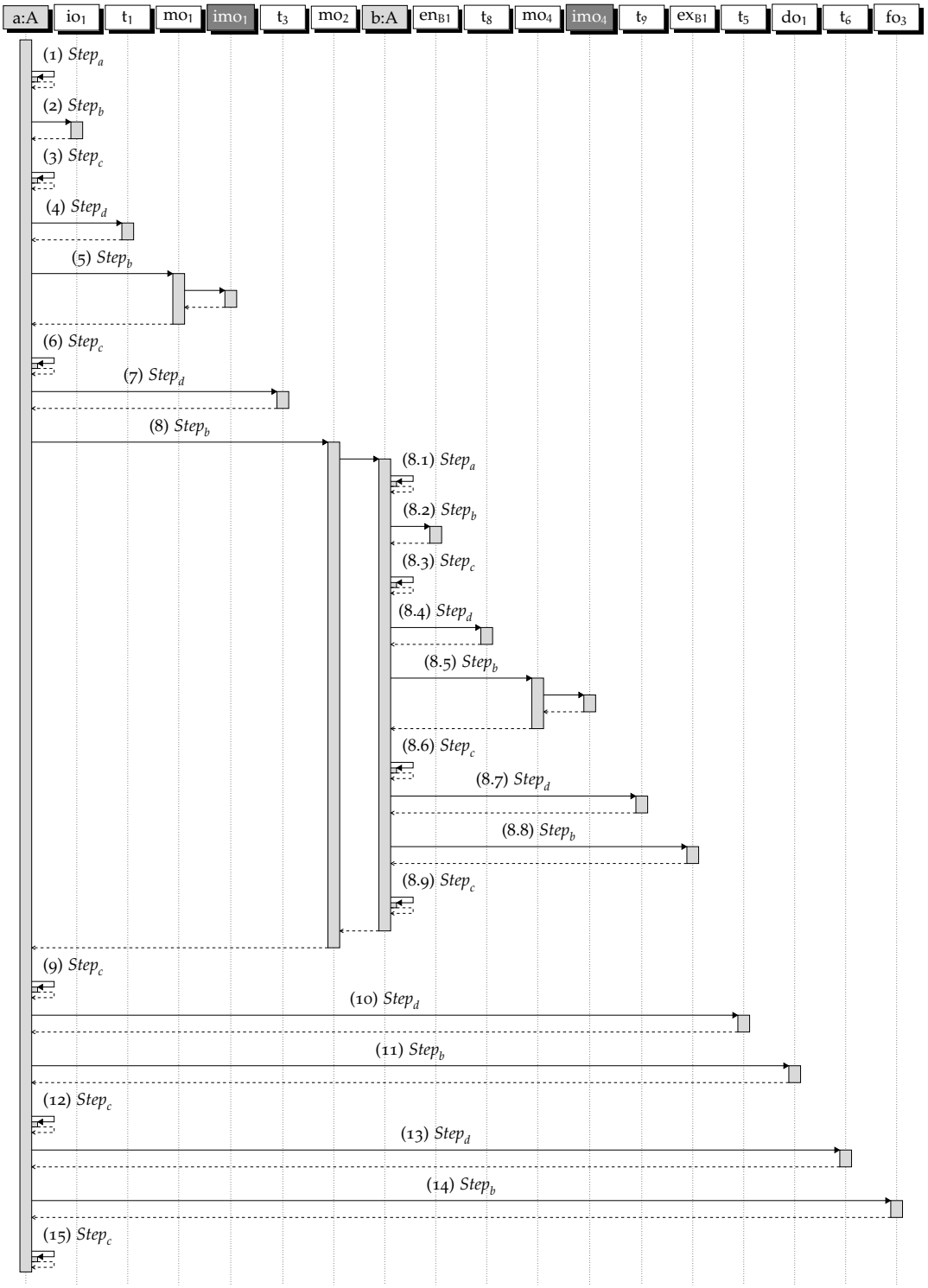


Figure 58: Interaction Diagram Showing the Execution Scenario of the Example.

(5) $Step_b$ It executes mo_1 by setting the $modelCounter_{a:A}$ to mo_1 and emitting the runtime event Before[mo_1.name] ($Step_{b1}$). Since mo_1 is a model operation ($Step_{b3}$), the interpreter retrieves the input runtime models {rm_1, rm_2} of mo_1 and uses the LD to identify the software module imo_1 implementing mo_1. It invokes imo_1 with the parameters {rm_1, rm_2}, which eventually returns a status, either "ex_1" or "ex_2", that is mapped by the interpreter to an exit of mo_1. We assume that the returned status is "ex_1". Thus, the interpreter activates the exit ex_1 and obtains the corresponding transition t_3 to be executed next. Having executed mo_1, the interpreter emits the runtime event After[mo_1.name] ($Step_{b7}$).

(6) $Step_c$ It continues execution since mo_1 is not a final operation.

(7) $Step_d$ It executes t_3 by setting the $modelCounter_{a:A}$ to t_3, emitting the runtime event OnTransition[t_3.name] ($Step_{d1}$), and obtaining the target of t_3, which is mo_2 ($Step_{d2}$).

(8) $Step_b$ It executes mo_2 by setting the $modelCounter_{a:A}$ to mo_2 and emitting the runtime event Before[mo_2.name] ($Step_{b1}$). Since mo_2 is a complex model operation ($Step_{b4}$), the interpreter uses the LD to identify the megamodel module b:B to which mo_2 is bound. It further identifies the initial operation en_{B1} of the FLD instance encapsulated in b:B based on the taken entry en_{B1} of mo_2. Having identified both, the interpreter synchronously invokes Execute(b:B, en_{B1}):

(8.1) $Step_a$ The interpreter obtains the FLD instance encapsulated in b:B and the initial operation en_{B1} to be executed.

(8.2) $Step_b$ It executes en_{B1} by setting the $modelCounter_{b:B}$ to en_{B1} as well as emitting the runtime event Before[en_{B1}.name] ($Step_{b1}$), obtaining the single outgoing transition t_8 to be executed next ($Step_{b2}$), and emitting the runtime event After[en_{B1}.name] ($Step_{b7}$).

(8.3) $Step_c$ It continues execution since en_{B1} is not a final operation.

(8.4) $Step_d$ It executes t_8 by setting the $modelCounter_{b:B}$ to t_8, emitting the runtime event OnTransition[t_8.name] ($Step_{d1}$), and obtaining mo_4 as the target of t_8 ($Step_{d2}$).

(8.5) $Step_b$ It executes mo_4 by setting the $modelCounter_{b:B}$ to mo_4 and emitting the runtime event Before[mo_4.name] ($Step_{b1}$). Since mo_4 is a model operation ($Step_{b3}$), the interpreter retrieves the input runtime models {rm_1, rm_4} of mo_4 and uses the LD to identify the software module imo_4 implementing mo_4. It invokes imo_4 with the parameters {rm_1, rm_4}, which eventually returns a status, either "ex_1" or "ex_2", that is mapped to an exit of mo_4. We assume that the returned status is "ex_1". Thus, the interpreter activates the exit ex_1 and obtains the corresponding transition t_9 to be executed next. Having executed mo_4, the interpreter emits the runtime event After[mo_4.name] ($Step_{b7}$).

(8.6) $Step_c$ It continues execution since mo_4 is not a final operation.

(8.7) $Step_d$ It executes t_9 by setting the $modelCounter_{b:B}$ to t_9, emitting the runtime event OnTransition[t_9.name] ($Step_{d1}$), and obtaining ex_{B1} as the target of t_9 ($Step_{d2}$).

(8.8) $Step_b$ It executes ex_{B1} by setting the $modelCounter_{b:B}$ to ex_{B1} and emitting the runtime event Before[ex_{B1}.name] ($Step_{b1}$). Since ex_{B1} is a regular final operation ($Step_{b6}$), the interpreter has nothing to do and just emits the runtime event After[ex_{B1}.name] ($Step_{b7}$).

(8.9) $Step_c$ It sets the $modelCounter_{b:B}$ to null and returns the final operation ex_{B1}.

Having executed b:B, the interpreter identifies the corresponding exit ex_{B1} of mo_2 based on the returned final operation ex_{B1}. Thus, it activates the exit ex_{B1} and obtains the corresponding transition t_5 to be executed next. Having executed mo_2, the interpreter emits the runtime event After[mo_2.name] ($Step_{b7}$).

(9) $Step_c$ It continues execution since mo_2 is not a final operation.

(10) $Step_d$ It executes t_5 by setting the $modelCounter_{a:A}$ to t_5, emitting the runtime event OnTransition[t_5.name] ($Step_{d1}$), and obtaining the target of t_5, which is do_1 ($Step_{d2}$).

(11) $Step_b$ It executes do_1 by setting the $modelCounter_{a:A}$ to do_1 and emitting the runtime event Before[do_1.name] ($Step_{b1}$). Since do_1 is a decision operation ($Step_{b5}$), the interpreter evaluates all expressions, each of which is annotated to an outgoing transition of do_1. We assume that $expr_1$ evaluates to true and $expr_2$ to false. Thus, the interpreter determines t_6 to be executed next. Having executed do_1, the interpreter emits the runtime event After[do_1.name] ($Step_{b7}$).

(12) $Step_c$ It continues execution since do_1 is not a final operation.

(13) $Step_d$ It executes t_6 by setting the $modelCounter_{a:A}$ to t_6, emitting the runtime event OnTransition[t_6.name] ($Step_{d1}$), and obtaining the target of t_6, which is fo_3 ($Step_{d2}$).

(14) $Step_b$ It executes fo_3 by setting the $modelCounter_{a:A}$ to fo_3 and emitting the runtime event Before[fo_3.name] ($Step_{b1}$). Since fo_3 is a regular final operation ($Step_{b6}$), the interpreter has nothing to do and emits the runtime event After[fo_3.name] ($Step_{b7}$).

(15) $Step_c$ It sets the $modelCounter_{a:A}$ to null and returns the final operation fo_3.

This scenario illustrates that the operations of an FLD instance are executed sequentially (see sequence diagram in Figure 58). Consequently, operations of the same FLD instance do not run concurrently. The same holds for operations of the invoking and invoked FLD instances. The behavior of an invoked instance is synchronously integrated into the invoking instance such that operations of the different instances do not run concurrently.

In general, the EUREMA interpreter alternately executes an operation and a transition until it reaches a final operation. Considering just these two general, executable elements, the execution semantics of FLDs is manageable, for instance, with respect to implementation. The semantics is only refined for the different kinds of operations. Moreover, the specification in terms of an FLD and therefore as well the execution of an instance of the FLD are well-defined and *deterministic*. Executing an operation determines exactly one outgoing transition to be executed next while a transition points to exactly one operation to be executed next. Consequently, there are no open choices with respect to execution, which would have to be decided otherwise by the interpreter. The specification in terms of the FLDs and LD is complete with respect to a deterministic execution.

Finally, to guarantee termination of the execution of an FLD instance, we require that the final operations are reachable from the initial operations of the instance. Moreover, we require that the software modules implementing (basic) model operations terminate. While an FLD specifies the input and output runtime models as well as the return states of model operations, engineers have to provide the concrete implementations of such operations (*i. e.*, the software modules). When executing such an operation, the interpreter invokes the corresponding software module. Thus, the termination of the execution of the operation depends on the termination of the execution of the software module (*cf.* Section 6.7). The other kinds of operations (i.e., initial, final, decision, and complex model operations) do not require any implementation since they are completely specified in the FLD and LD.

6.4.4 *Maintaining Runtime Information of Megamodel Modules*

While executing an FLD instance encapsulated in a megamodel module, the EUREMA interpreter maintains runtime information about the execution. On the one hand, for each megamodel module the interpreter counts how often the module has been executed as well as the point in time when the last execution has finished. Hence, having finished the execution of a megamodel module, the interpreter increments the counter of this module by one and updates the time of the last execution to the current time stamp.

On the other hand, the interpreter maintains similar information for each operation and transition. It counts how often and when an operation or transition has been executed. Thus, having finished the execution of an individual operation or transition, the interpreter increments the corresponding counter by one and updates the time of the last execution to the current time stamp. Moreover, it maintains an additional and special counter for each transition, which reflects the number of how often the transition's source operation has been consecutively executed without taking this transition but another outgoing transition of the same operation. Hence, when executing a transition, the interpreter resets the counter for this transition to zero while it increments each counter of the alternative transitions by one. Thus, the interpreter maintains information about how often and when an operation or transition has been executed as well as how often a transition has not been taken consecutively but another outgoing transition of the same operation. For all operations and transitions, this information can be used and queried at runtime, particularly, by conditions for decision operations that exclusively branch the control flow in FLDs. The query and use of this information in conditions is discussed in detail in Section A.3.

6.4.5 *Maintaining Runtime Models*

As previously discussed, executing a model operation as part of an FLD instance requires that the interpreter provides runtime models as input to the operation (and to the software module implementing the operation) and that it obtains the output runtime models from the operation. Thereby, multiple model operation may use the same runtime models, for instance, the output model of one operation is the input model of another operation. Thus, the interpreter has to maintain the runtime models used within a megamodel module.

Each runtime model as part of an FLD is a representation of a resource that materializes the runtime model. Therefore, each runtime model specified within an FLD is bound to a resource by a Uniform Resource Identifier (URI) that identifies the corresponding resource. In this context, multiple models can be bound to the same resource. To manage the resources and the bindings, the interpreter has a basic repository from which it loads resources, retrieves already loaded resources, and adds as well as removes resources.

When executing a model operation, the interpreter loads the resources, that is, the materialized runtime models that are the inputs of the model operation, and passes them as parameters of the invocation to the software module implementing the operation. Thus, the resources are loaded on-demand when they are actually needed for execution instead of loading all resources when starting interpreter or the execution of the whole FLD instance. The invoked software module may modify the input resources as well as create and destroy resources (*cf.* different kinds of model usages in the FLD such as reading, writing, annotating, creating, and destroying models as discussed in Section 5.1.2). After the invocation, the software module returns the output resources and the interpreter updates the repository if resources have been created or destroyed. Consequently, it is the responsibil-

ity of the model operation and its implementation to create and destroy the materialized runtime models. The interpreter does not instantiate any resources or perform any garbage collection of resources if the resources are created or not used any more by any operation.

For all kinds of modifications of a resource by a model operation, the runtime model element in the FLD representing the resource as well as the model usage link between the runtime model and operation in the FLD are not modified when executing the operation. The reason for this is that they specify how runtime models are modified in *each* run of the feedback loop. For instance, when an operation destroys a model/resource, the runtime model element and the model usage link annotated with `d` (*i. e.*, destroy) remain in the FLD as they specify that the model/resource is destroyed in each run of the feedback loop. Another example is the creation of a runtime model, which means that the resource is created anew in each run of the feedback loop. Practically, this means that the resource is overwritten with each run if it is not destroyed after its creation within the same run.

Consequently, the kind of model usages (*i. e.*, reading, writing, annotating, creating, and destroying models) does not impact the execution semantics of FLD instances. The interpreter is only concerned with loading and providing resources materializing runtime models to the software modules implementing model operations. For this purpose, the interpreter leverages the *unification power of models* (*cf.* Section 2.1.1) that allows it to maintain each materialized runtime model in a unified manner regardless of the model's metamodel, content, and purpose (*cf.* different kinds of runtime models in Section 4.1). Consequently, the interpreter is able to manage any user-defined runtime model that is part of an FLD.

Finally, the unification power of model is further leveraged by the interpreter to manage the EUREMA models as visualized by FLDs and LDs at runtime. Consequently, the EUREMA models are treated similarly to the runtime models, that is, they are materialized by resources with a URI and also maintained in the interpreter's repository. This aspect becomes relevant when a megamodel module uses an FLD instance as a normal runtime model in order to reflect on the feedback loop specified and executed by this instance (*cf.* procedural reflection for layering feedback loops in Section 5.3.1). In this case, an FLD instance specifying and executing the reflected feedback loop is the input and output of a model operation similarly to any other runtime model. Therefore, the interpreter has to obtain the specific FLD instance to which the FLD model (*i. e.*, the runtime model for reflecting on a feedback loop; see Section 5.1.2) is bound when executing the model operations that have the FLD model as an input. The binding of an FLD model to a specific FLD instance is defined in the LD (*cf.* Section 5.3.1). Thus, when executing a model operation that has the FLD model as an input, the interpreter retrieves the corresponding binding and thus the appropriate FLD instance from the LD. Using the URI of the retrieved FLD instance, the interpreter loads the materializing resource, which is similar to loading any other runtime model from the repository.

Summing up, all runtime models including the EUREMA models are managed in a unified manner by the interpreter that is able to generally load any model that is used as an input of a model operation. To execute such an operation, the interpreter loads the runtime models that are required as inputs and passes them to the software module implementing the operation. The software module may then operate on these models at will without having to load and unload them. Thus, the interpreter is responsible of the runtime management of the runtime models, which unburdens the user-defined implementations of the model operations. The implementations can therefore focus on performing the domain-specific task such as the monitoring, analyzing, planning, or executing activities.

6.5 INTERACTIONS BETWEEN MODULES

Having discussed the semantics of executing a single megamodel module (*cf.* Section 6.4), we now discuss the interactions between multiple modules. We already clarified in Section 6.3 that megamodel modules that are not interrelated in the LD—except of sensing and effecting the same adaptable software—are executed concurrently and without any interactions. In contrast, if megamodel modules are directly interrelated in the LD, the interactions between them are explicitly specified with the EUREMA language and executed by the EUREMA interpreter. The potential interactions in terms of *sensing*, *effecting*, and, *using* modules have already been discussed in Section 6.4 from the point of view of an individual megamodel module. In the following, we take a global view to discuss the impact of the interactions on the whole self-adaptive software.

The possible interactions among modules are illustrated in Figure 59 showing the three lowest layers of an exemplary self-adaptive software system. The lowest layer contains the adaptable software and the higher layers comprise the adaptation engine with the feedback loops. More specifically, the megamodel module al:AdaptableLoop at Layer-1 realizes a feedback loop and *senses* as well as *effects* the underlying :AdaptableSoftware. This feedback loop itself is *sensed* and *effected* by the megamodel module fl:FeedbackLoop that also realizes a feedback loop and that is located at the higher layer called Layer-2. In general, the system could have more layers each of which would behave similarly to Layer-2 with respect to the interactions. A megamodel module realizing a feedback loop at any layer of the adaptation engine (*e. g.*, al:AdaptableLoop and fl:FeedbackLoop) *uses* software modules that provide implementation of its model operations and it may *use* other megamodel modules that realize parts of the feedback loop and that are invoked by its complex model operations.

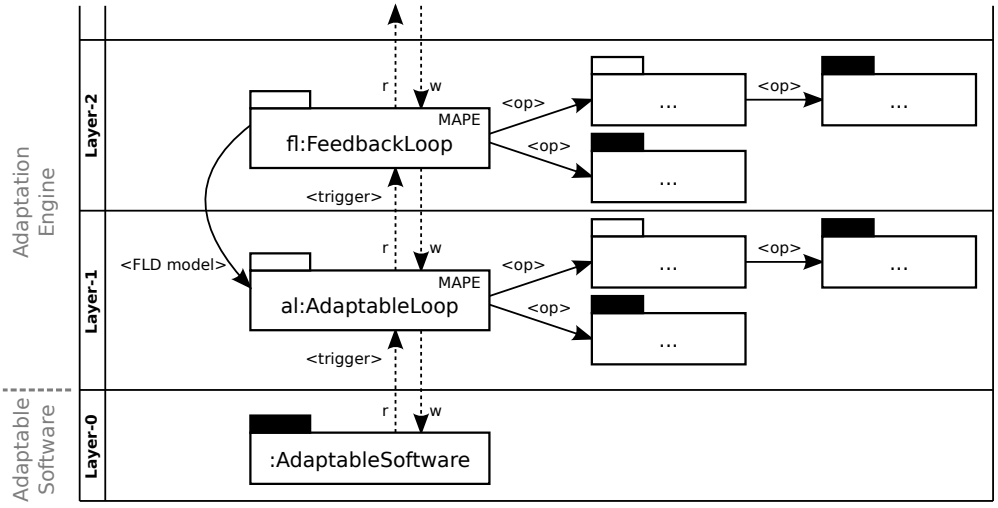


Figure 59: Interactions between Megamodel and Software Modules.

Concerning the execution of the system and interactions, the adaptable software comprising the lowest layer of the system is continuously running. The EUREMA interpreter does not control the execution of the adaptable software. It only uses events emitted by the software to *asynchronously* trigger and execute the megamodel modules at the lowest layer of the adaptation engine (*cf.* Sections 6.4.2 and 6.4.3). Considering Figure 59, the interpreter asynchronously triggers and executes the al:AdaptableLoop while the :AdaptableSoftware is running. Thus, the adaptable software is not blocked while the feedback loops are running.

Executing a megamodel module such as al:AdaptableLoop and particularly the encapsulated FLD instance involves the *synchronous* invocation of a software module when executing a model operation and of another megamodel module when executing a complex model operation. An invoked megamodel module itself may synchronously invoke software and megamodel modules for the same reasons. Thus, the megamodel module realizing a feedback loop and the modules it uses/invokes transitively are executed by the same and single thread of control such that all of these modules are not running concurrently.

For instance, Figure 59 illustrates that the megamodel module al:AdaptableLoop at Layer-1 uses a software module as well as a megamodel module that in turn uses a software module. In this example, all of these four modules at Layer-1 are executed by one single thread of control and they do not run concurrently.

The execution of a megamodel module at a higher layer of the adaptation engine is triggered by intercepting the sensed and effected megamodel module at the directly underlying layer (*cf.* Section 6.4.2). Since this lower-layer module and its used modules are executed by a single thread of control, the execution of all of these modules is intercepted and blocked to *synchronously* execute the higher-layer module. Thus, a higher-layer megamodel module is not running concurrently with the sensed and effected megamodel module as well as the modules used by the sensed and effected module. For instance, the higher-layer megamodel module fl:FeedbackLoop is triggered by intercepting the module al:AdaptableLoop and therefore all of the modules at Layer-1 (*cf.* Figure 59). Thus, all modules at Layer-1 are blocked to execute the module fl:FeedbackLoop. Similar to executing the al:AdaptableLoop, executing the fl:FeedbackLoop involves *synchronous* invocations of software and megamodel modules when executing its basic respectively complex model operations such that fl:FeedbackLoop and its invoked modules do not run concurrently.

Consequently, while executing the higher-layer modules (*e. g.*, fl:FeedbackLoop and its used modules at Layer-2), the execution of the sensed and effected lower-layer modules (*e. g.*, al:AdaptableLoop and its used modules at Layer-1) is blocked. Since the lower-layer feedback loop is blocked, it can be safely adapted by the higher-layer feedback loop (a detailed discussion of safe adaptation is given in Section 6.6). To reflect on the lower-layer feedback loop, the higher-layer loop may directly use the FLD instance that specifies the lower-layer loop and whose execution is currently blocked (*cf.* procedural reflection in Section 5.3.1). As discussed in Section 5.3.1, procedural reflection is visible in the LD because of the binding of the runtime model used by the higher-layer loop to a specific FLD instance (see the use relationship, that has the placeholder <FLD model> for the name of the runtime model to be bound, pointing from fl:FeedbackLoop to al:AdaptableLoop in Figure 59).

The idea of intercepting lower-layer modules to synchronously execute higher-layer modules while potentially using procedural reflection applies recursively when having more than two layers in the adaptation engine. For instance, executing modules at Layer-3 (not shown in Figure 59) is done by intercepting and blocking modules at Layer-2 that were running because of intercepting and blocking the running modules at Layer-1. For the whole adaptation engine, this means that none of the modules are running concurrently and that one thread conceptually controls the execution of all modules. In contrast, concurrency of modules only exists between the adaptation engine and the adaptable software.

Summing up, modules of the adaptation engine that are interrelated in the LD in terms of sensing, effecting, and using relationships do not run concurrently, which avoids any need of synchronization mechanisms to ensure consistency among the modules. In other terms, feedback loops that are stacked on each other and therefore operate at different layers of the engine do not run concurrently. We think that this does not need to be a drawback—as we

save efforts for specifying and executing the synchronization mechanisms—if the computations of the individual feedback loops are not costly and if the individual feedback loops operate on similar time scales. Hence, coupling the stacked feedback loops with respect to execution and potentially procedural reflection is acceptable. Otherwise, a stronger decoupling of the stacked feedback loops by supporting their concurrent execution could be beneficial while requiring synchronization mechanisms. Nevertheless, EUREMA executes the feedback loops concurrently to the execution of the adaptable software such that the software can continue providing services to users or other systems.

Finally, coming back to feedback loops that are independent of each other and therefore not interrelated in the LD (*cf.* Section 6.3), these feedback loops are concurrently executed in EUREMA. The engineer must assure the independence of the feedback loops by design and construction, otherwise interferences between the feedback loops may occur and remain unhandled. Figure 60 illustrates with the dashed, gray boxes the level of concurrency that is supported by EUREMA when having two independent (stacks of) feedback loops. Thus, these two (stacks of) feedback loops run concurrently if they are appropriately triggered such that the two executions overlap in time. Thereby, the adaptable software is continuously running and hence concurrently running to the two (stacks of) feedback loops. In general, more than two independent (stacks of) feedback loops may run concurrently.

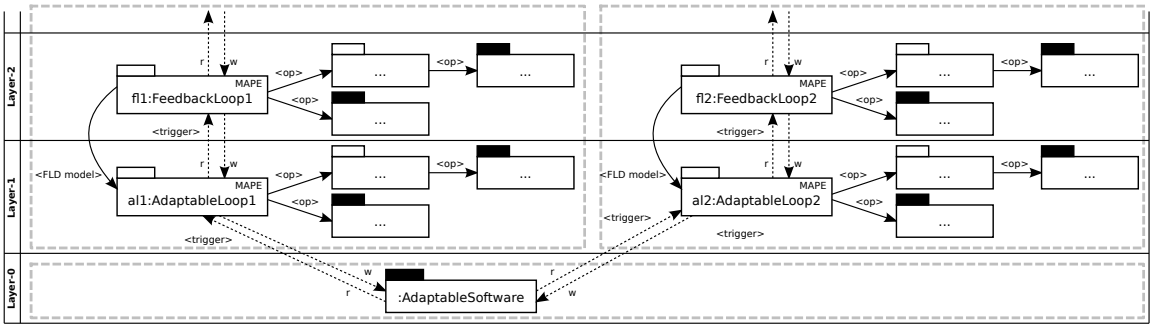


Figure 60: Supported Concurrency between Megamodel and Software Modules.

Thus, EUREMA supports the concurrent execution of the adaptation engine and the adaptable software as well as of independent feedback loops. Within a feedback loop, EUREMA does not support concurrency since the operations in an FLD instance specifying the feedback loop are executed sequentially and synchronously, which involves the synchronous invocation of software modules and other megamodel modules.

6.6 SAFE ADAPTATION OF EUREMA MODELS

Having discussed the execution semantics of EUREMA models, particularly of how the behavior-dominant FLDs are executed with the help of the structure-dominant LD, we focus in the following on the dynamic adaptation of these models. This discussion will cover the scope of adaptation (*i. e.*, what can be changed at runtime) as well as the mechanisms to achieve *safe* adaptation. In general, an adaptation is safe if the system under adaptation is in a consistent state before and after being changed [259, 284, 417]. In the context of EUREMA, safe adaptation refers to dynamically changing the EUREMA models, which leaves the feedback loops in a consistent state before and after the adaptation, and which does not corrupt the execution of the models.

Dynamic adaptation is inherent to self-adaptive software and typically targets the adaptable software. To achieve safe adaptations of the adaptable software, mechanisms such as quiescence [259], tranquility [417], or version consistency [56, 284] have been developed. They determine points in time when (parts of) the adaptable software can be safely adapted such that the software is in a consistent state before and after the adaptation. Such safe adaptation points are achieved by partially disrupting the execution of the software. Since EUREMA is a generic approach and therefore independent of any specific adaptable software and of the software's underlying technology, we require that the adaptable software provides appropriate sensors and effectors that support its safe adaptation (*cf.* Chapter 3).[4] Consequently, the safe adaptation of an adaptable software is out of the scope of EUREMA.

Nevertheless, the issue of safe adaptations becomes relevant for EUREMA when an adaptation targets the adaptation engine and the feedback loops developed with EUREMA. The execution semantics of EUREMA addresses the safe adaptation of EUREMA models such that the feedback loops can be dynamically changed without corrupting their consistency and execution. For this purpose, the execution semantics provides mechanisms that are inspired by quiescence and that are implemented in the interpreter. In the following, we discuss these mechanisms for the two cases of adapting either FLD or LD models.

6.6.1 *Safe Adaptation of FLD Models*

A higher-layer feedback loop can dynamically adapt an FLD model/instance that specifies and executes a lower-layer feedback loop. Hence, the safe adaptation of FLD models becomes relevant when stacking feedback loops in layered architectures, in which higher-layer feedback loops adapt lower-layer ones (*cf.* Section 5.3).

Safe adaptations of feedback loops are mainly enabled by EUREMA's approach of triggering higher-layer feedback loops based on lower-layer loops. Particularly, the megamodel module with its encapsulated FLD model that realizes the higher-layer feedback loop is triggered by intercepting the megamodel module realizing the lower-layer feedback loop (*cf.* Section 6.4.2). Having a stack of arbitrary many megamodel modules, this triggering approach always results in a situation where at most one of these module of the stack is running while the other modules are either intercepted or they have not been triggered yet (*cf.* Section 6.5). More specifically, if a megamodel module in a stack of such modules is running, all of its underlying modules have already been triggered and partially executed but they are now blocked while all of its overlying modules have not been triggered yet.

Thus, the adapted feedback loop in terms of the underlying megamodel module is intercepted and therefore not running while being changed by the currently running overlying module. Moreover, it continues execution only after the execution of the overlying module has terminated and hence after its adaptation has happened. In this context, the adapted feedback loop is in a consistent state when it is intercepted and adapted as well as when it continues execution after the adaptation and interception because of the following aspects:

- Inconsistencies due to concurrently executed feedback loops are generally avoided by intercepting and blocking the lower-layer, adapted feedback loop to run the higher-layer, adapting feedback loop. Thus, a feedback loop is not adapted while it is running.

4 For our implementation and experiments with EUREMA (*cf.* Chapters 8 and 9), we use *mKernel* that provides sensors and effectors for EJB-based applications and that supports quiescence and therefore safe adaptations of such applications [3, 91]. We use the EJB-based mRUBiS application (*cf.* Section 4.5) as the adaptable software, which is observable and adaptable through mKernel. Consequently, the feedback loops developed with EUREMA use the mKernel sensors and effectors to observe and safely adapt mRUBiS at runtime.

- The execution of a feedback loop can be intercepted before or after executing one of its operations or while executing one of its transitions (*cf.* the corresponding Before, After, and OnTransition event types in Section 6.4.2). At these interception points the feedback loop is in a consistent state since no behavior of the feedback loop is currently executed. In contrast, behavior is only executed *while* an operation is executed. Especially a model operation works on runtime models and it may modify these models. Therefore, when intercepting a model operation during its execution, the runtime models might be in an inconsistent state since the operation has not finished its work yet. This impedes adaptation of the feedback loop in such a situation, for instance, by changing its runtime models. Consequently, we consider an operation as an atomic unit of execution that cannot be intercepted during its execution to safely adapt the feedback loop.

  The state of execution between executing two operations (*i. e.*, after executing one operation and before executing the next one) is the execution of a transition connecting these two operations. Since each transition just connects two operations and does not have any associated behavior, we may also intercept the execution of a transition for safe adaptations of the feedback loop.
- The execution of a feedback loop in terms of an FLD instance is the sequential, non-concurrent execution of its operations (*cf.* Section 6.4.3). Thus, there is only one single thread of control, which is intercepted either before or after executing an operation or while executing a transition. Consequently, having intercepted the single thread of control, there does not exist any other thread of control that executes any operation of the same FLD instance. This ensures that there is no active behavior and the feedback loop is completely blocked with one interception.
- The adaptation of an intercepted feedback loop is restricted concerning the interception point such that the execution of this feedback loop can properly continue after the adaptation and interception. The restriction is that the adaptation must not remove the interception point. Otherwise, the EUREMA interpreter looses the position of the model counter in the FLD instance, at which it should continue the execution of the instance after the interception. As discussed in Section 6.4.3, the interpreter maintains for each FLD instance a model counter that points to the currently executed operation or transition of the instance. Removing the operation or transition, to which the model counter points, in the context of an adaptation impedes that the interpreter can continue the execution after the adaptation.

  In general, the interception point of a (lower-layer) feedback loop is determined by the triggering condition of the higher-layer feedback loop using Before, After, or OnTransition events that refer to specific operations or transitions of the intercepted loop. Thus, a feedback loop is intercepted either before executing a specific operation, after execution a specific operation, or when executing a transition (*cf.* Section 6.4.2). When intercepting a feedback loop, the model counter points to the corresponding operation or transition of this feedback loop, that is, the operation or transition to which the event of the triggering condition refers. Hence, the operation or transition used in the triggering condition for the higher-layer feedback loop must not be removed by the higher-layer loop.

  Nevertheless, other kinds of adaptations effecting the corresponding operation or transition of the interception point are possible. For an operation, its wiring to other operations (*i. e.*, the control flow), its usage of runtime models, and the used runtime models themselves can be adapted. For a transition, its source and target operation can be changed, for instance, such that a transition points to a different operation.

- Each operation and particularly each model operation should be *stateless*. Its state such as its working data should be externalized and kept in the runtime models that are used by the operation (*cf.* Sections 4.1 and 4.2). Consequently, the software module implementing a model operation should not keep any working data across multiple executions of the operation and rather store it in the runtime models. This enables a consistent state of the feedback loop, operations, and runtime models after an operation has been executed and before the next operation starts its execution.

  The initial, final, decision, and complex model operations (*i. e.*, all kinds of operations except of model operations) are stateless by design and they do not require any implementation by engineers to execute them. In contrast, engineers have to provide implementations (*i. e.*, software modules) for model operations that work on runtime models. Here we require that engineers implement stateless software modules to ease safe adaptations by capturing the relevant state such as working data in the runtime models.

Based on these aspects, the feedback loop to be adapted is moved to a state similar to quiescence (*cf.* [259]) that enables its safe adaptation by a higher-layer feedback loop. An adaptation of a feedback loop (*i. e.*, of an FLD instance) may change the elements of an FLD. Operations, runtime models, control flow links (*i. e.*, transitions between operations), and model usage links (*i. e.*, the input and output relationship between operation and runtime models) can be added or removed—except of the operation or transition that is actually used to intercept the feedback loop (*i. e.*, the current target of the model counter).

To perform adaptation, the higher-layer feedback loop can either employ procedural or declarative reflection (*cf.* Section 5.3). In both cases, the aspects discussed previously hold such that the adaptation is safe. For procedural reflection, the adaptation is directly analyzed, planned, and executed on the intercepted FLD instance. For declarative reflection, a representation of the intercepted FLD instance is created and used for analyzing and planning the adaptation while the FLD instance is only effected when eventually executing the adaptation. Hence, the two kinds of reflection only differ in how the higher-layer feedback loop uses the FLD instance but they do not impact the aspects discussed previously.

Despite the aspects and restrictions that are required for safe adaptations of FLD models, engineers do not have to implement any mechanism to achieve such safe adaptations. In contrast, EUREMA takes over the responsibility by appropriately intercepting and resuming the execution of the feedback loop to be adapted. Engineers only have to define the concrete interception points (*i. e.*, the specific operation before or after whose execution or the specific transition during whose execution the feedback loop should be intercepted) by specifying appropriate triggering conditions for the higher-layer feedback loop.

User-defined interceptions points allow engineers to take application-specific constraints into account, which are not in the scope of EUREMA. For instance, two model operations of a feedback loop use the same runtime model. When adapting this runtime model, a constraining dependency between the two model operations exists if both of them should use either the original or the adapted version of the runtime model within one run of the feedback loop (*cf.* version consistency [56, 284]). Hence, the adaptation should take place either before or after the execution of both operations but not in between the execution of both. Engineers may respect such constraints by specifying appropriate triggering conditions (*e. g.*, to intercept the feedback loop before or after the execution of the two operations).

Summing up, safe adaptations of FLD models are mainly achieved by EUREMA while engineers contribute by providing stateless implementations for model operations, which externalize the state to runtime models, and by specifying triggering conditions. The latter

aspect gives the engineers the possibility to address application-specific constraints influencing when a feedback loop can be intercepted and safely adapted.

6.6.2 *Safe Adaptation of LD Models*

An LD model describes the layered architecture of the self-adaptive software and especially the structuring of the megamodel modules (*i. e.*, feedback loops) in the adaptation engine together with their triggering conditions and their interrelations in terms of sense, effect, and use relationships. This LD model is kept alive at runtime and supports the execution of the behavioral FLD models by providing the necessary structural information to the interpreter (*cf.* Section 6.4). In this context, the LD model serves as a procedural reflection model of the layered architecture, that is, it specifies the architecture, supports the execution, *and* it can be dynamically changed to directly adapt the architecture.

Technically, an LD model can be changed by adding or removing layers, megamodel modules, sense and effect relationships between modules, bindings (*i. e.*, use relationships) of basic or complex model operations to software respectively megamodel modules, as well as triggering conditions of megamodel modules. For example, such dynamic changes are performed by the EUREMA interpreter in the context of off-line adaptation that supports adding or removing a feedback loop as well as adapting an already deployed feedback loop and the adaptable software (*cf.* Section 5.4). Moreover, it is conceivable that another adaptation engine is employed on top of the self-adaptive software. This engine would use the LD model to reflect upon and adapt the underlying self-adaptive software.

Such adaptations of the layered architecture have to be safe such that the self-adaptive software is in a consistent state before and after the runtime changes. For instance, a megamodel module realizing a feedback loop cannot be removed while it is executing an adaptation to the adaptable software, otherwise it may leave the adaptable software in a partially adapted and thus inconsistent state. In general, adapting the layered architecture of a self-adaptive software is similar to adapting the architecture of the adaptable software as elements of the architecture are added, removed, or changed. Such architectural changes require some form of quiescence to keep the system in a consistent state (*cf.* [259]).

Therefore, we apply the idea of quiescence and map it to EUREMA such that the LD model and thus the layered architecture can be safely adapted. The EUREMA execution semantics supports quiescence and hence safe changes of the LD model as follows:

- Since the EUREMA interpreter is in charge of executing the feedback loops, it can move the whole adaptation engine to a quiescent state. Quiescence of the adaptation engine means that all feedback loops are inactive—since they have finished their execution—and that none of them will be triggered again. The whole adaptation engine is quiescent if *all* megamodel modules are quiescent. A megamodel module is quiescent if

  1. it is currently inactive, that is, its encapsulated FLD instance is not currently executed as well as not currently intercepted to execute a higher-layer megamodel module,
  2. it will not be triggered to start execution, and
  3. it will not be invoked by another megamodel module.

  The first criterion implies that the megamodel module has not been triggered or invoked by another megamodel module yet or if it has, then the execution of the encapsulated FLD instance has terminated properly with a final operation. Hence, the FLD instance is not currently active and will therefore not invoke any other megamodel or software

module, which otherwise happens when executing a complex or basic model operation. In general, an FLD instance can become active when it is triggered or invoked by another megamodel module, which is, however, excluded by the second and third criterion.

These three criteria determine when a megamodel module is quiescent. If they hold for all megamodel modules, then the adaptation engine is quiescent and therefore ready for safe adaptations. Consequently, changes of the layered architecture may only target the adaptation engine but not the adaptable software. The adaptable software is not affected by the quiescence and continues execution. Nevertheless, changing the adaptable software is possible but requires the enactment of the change through the adaptation engine. For instance, to update the adaptable software, an update process needs to be added to the engine where it is executed to perform the update (*cf.* Section 5.4.4).

To achieve quiescence of the adaptation engine, we can use a simple approach at the technical level because of the execution semantics for triggering and executing megamodel modules (*cf.* Sections 6.4 and 6.5). Megamodel modules that are stacked on each other do not run concurrently as well as such modules located at higher layers of the adaptation engine are synchronously triggered and executed by intercepting the execution of the modules at the corresponding adjacently lower layers. Moreover, megamodel modules can be synchronously invoked by other megamodel modules of the same layer in the adaptation engine. Consequently, the execution of a megamodel module can only terminate when the executions of the module's higher-layer modules terminate recursively and when the executions of its invoked megamodel modules terminate.

Accordingly, to achieve the first criterion for all megamodel modules (*i. e.*, all megamodel modules are currently inactive), the EUREMA interpreter continues the executions of all currently running modules until all of them terminate. To achieve termination of a megamodel module, the interpreter allows intercepting the module to trigger and execute higher-layer modules and it allows the module to invoke other modules.

Thereby, the interpreter blocks any triggering of megamodel modules that have a triggering condition and that are located at the lowest layer of the adaptation engine. Such a module is generally triggered asynchronously and only when its previous execution has terminated (*cf.* Section 6.4.2). The blocking prevents any newly initiated executions of megamodel modules at the lowest layer of the adaptation engine, which would otherwise require further executions of other modules at the same layer (*i. e.*, based on invocations) or at higher layers (*i. e.*, based in interceptions) to eventually terminate. Thus, already running modules can terminate their executions while new executions starting from the lowest layer of the adaptation engine are blocked.

Since the stacked megamodel modules are triggered bottom-up the layered architecture based on interceptions, these modules terminate top-down. As soon as all of the megamodel modules at the lowest layer of the adaptation engine terminate, all megamodel modules are inactive (*i. e.*, the first criterion). Consequently, no megamodel module will be triggered by interceptions of any other megamodel module (*i. e.*, the second criterion) and invoked by any other megamodel module (*i. e.*, the third criterion). Therefore and since the interpreter blocks any triggering of the megamodel modules at the lowest layer of the adaptation engine (*i. e.*, the second criterion), the inactive modules do not become active such that the whole adaptation engine is quiescent. This notion of quiescence results in a consistent state of the adaptation engine since all feedback loops have properly terminated and no new executions are initiated.

For instance, considering the example in Figure 59 on Page 127 and assuming that the megamodel module al:AdaptableLoop is already running, the interpreter achieves quiescence of the adaptation engine as follows. It first blocks any further triggering and thus any newly initiated execution of the module al:AdaptableLoop, which is the only megamodel module with a trigger at the lowest layer of the engine. Since a megamodel modules is not reentrant, an execution can only be initiated when the module is not currently active. Meanwhile, the interpreter continues executing the module al:AdaptableLoop until termination, which might require invocations of other modules of the same layer. It might further require triggering and executing the higher-layer module fl:FeedbackLoop if the corresponding interception point of the module al:AdaptableLoop is reached. If this is the case, the interpreter will execute the module fl:FeedbackLoop until termination, which might require as well invoking other modules at the same layer or even triggering modules at higher layers (only betokened in Figure 59 on Page 127). In the other cases, the interception point either has been reached before such that the module fl:FeedbackLoop has already been triggered and executed, or it will not be reached in this specific run of the module al:AdaptableLoop such that the module fl:FeedbackLoop will not be triggered and executed at all. Thus, when triggering the module fl:FeedbackLoop by intercepting the module al:AdaptableLoop, terminating the execution of the former module is a prerequisite for terminating the execution of the latter module. Then, having finished the execution of the module fl:FeedbackLoop, the interpreter continues and eventually finishes the execution of the module al:AdaptableLoop. In this situation, no module of the adaptation engine is currently active and will be triggered or invoked. This results in a quiescent adaptation engine.

Considering multiple, concurrent stacks of megamodel modules as illustrated in Figure 60 on Page 129, the interpreter performs the same behavior for each stack to achieve quiescence. That is, it continues the execution of each stack of megamodel modules until termination and for each stack, it blocks newly initiated executions of the megamodel modules at the lowest layer of the adaptation engine. Quiescence of the adaptation engine is reached when all megamodel modules of each stack are quiescent.

- Having reached quiescence, the LD model and thus the adaptation engine can be safely adapted, for instance, by the EUREMA interpreter that executes an off-line adaptation (*cf.* Section 5.4) or by a higher-layer adaptation engine directly changing the LD model.
- After the adaptation, the interpreter disables quiescence of the adaptation engine by releasing the blocking of the megamodel modules at the lowest layer of the engine. Consequently, these modules can be triggered and executed again (*i. e.*, executions can be newly initiated), which includes invocations of other modules as well as interceptions to trigger higher-layer megamodel modules. In the context of these executions, the changes caused by the adaptation are taken into account, for instance, an added module can be triggered or invoked. In general, the execution of the adapted LD model and of the corresponding megamodel modules follows the semantics discussed in Sections 6.2–6.5.
- The interpreter continues executing the current LD model and the corresponding megamodel modules until the next adaptation cycle happens, which requires quiescence.

These steps constitute a safe adaptation of an LD model. Since the EUREMA interpreter controls the execution of the adaptation engine, it can easily support quiescence and therefore safe adaptations of the engine, for instance, to enact an off-line adaptation (*cf.* Sec-

tion 5.4). Similar to safe adaptations of FLD models, engineers do not have to provide any implementation to achieve quiescence for safely adapting an LD model.

The notion of quiescence to safely adapt LD models is more coarse-grained and disruptive than the notion of quiescence to safely adapt FLD models. The reason is that changes of an LD model target the abstraction level of modules (*e. g.*, adding or removing a megamodel module), which has a larger impact on the whole adaptation engine than changes of an FLD model occurring at the lower abstraction level of elements within an individual module (*e. g.*, adding or removing a runtime model used within a megamodel module).

The changes of the FLD or LD models themselves are not specified by EUREMA and hence not covered by the execution semantics of EUREMA. In contrast, the changes of these models can be done by any mechanism that supports the modification of models and that has its own execution semantics. For instance, we use a generic graph transformation formalism with clearly defined semantics [8] to specify and execute the changes of the LD model in the context of off-line adaptation (*cf.* Section 5.4). For this purpose, EUREMA integrates this formalism as well as an interpreter for this formalism to execute the adaptations of the LD model. In general, EUREMA is not restricted to a specific formalism to express changes of FLD or LD models such that other formalism can be integrated. EUREMA only requires that the changes of an FLD or LD model results in a valid model, that is, the changed model must adhere to the metamodel and well-formedness constraints of EUREMA (*cf.* Appendix A). Validity of FLD and LD models is a prerequisite for the execution semantics such that these models are executable by the EUREMA interpreter.

Summing up, EUREMA automatically supports safe adaptations of feedback loops (*i. e.*, FLD models) and adaptation engines (*i. e.*, LD models) without requiring from engineers to implement any functionality to achieve quiescence. This functionality is generically realized by EUREMA that controls the execution of the FLD and LD models. Thus, engineers are not concerned with the realization and management of quiescence but can focus exclusively on developing the adaptation logic and feedback loops. Since an adaptation of the FLD or LD models only happens in a quiescent state and since it does not change the execution status (*e. g.*, the model counter that is similar to a program counter for an execution stack), the EUREMA interpreter has always control over the execution of the FLD and LD models. This enables the interpreter to automatically achieve quiescence for safe adaptations and finally to properly continue the execution of the adapted models.

6.7 REQUIREMENTS FOR MODEL OPERATIONS IN EUREMA

Having discussed the execution semantics of EUREMA in the previous sections, we have defined how the EUREMA models developed by engineers are executed. Besides the EUREMA models engineers have to provide implementations of the model operations that they specified in the EUREMA models. These implementations are represented as software modules in the EUREMA models while each model operation is bound to such a module (*cf.* Section 5.1.4). When executing a model operation, the corresponding software module is invoked by the EUREMA interpreter (*cf.* Section 6.4.3). However, the actual implementations are only triggered but not executed by the interpreter since they can be realized with any formalism and technology. Consequently, the interpreter does not directly control the execution of the implementations. Executing a feedback loop in terms of the EUREMA models still includes the execution of the implementations for the model operations. Therefore, the implementations have to satisfy certain requirements such that they do not violate the execution semantics of EUREMA. We discuss these requirements in the following.

Satisfying the specification. Engineers define model operations when specifying a feedback loop with an FLD. For an individual model operation, they define the input and output runtime models of the operation, how these models are processed by the operation (*cf.* different kinds of model usages such as reading, writing, annotating, creating, or destroying models), and the possible return states of the operation in terms of exit compartments. This definition serves as an abstract specification of the model operation, which has to be satisfied by the implementation.

When executing a model operation as defined in the FLD, the EUREMA interpreter invokes the implementation and passes the specified input models to the implementation. The implementation executes, processes these input models, eventually terminates, and finally returns the output models as well as the specific return status, in which the execution has terminated. Consequently, the EUREMA execution semantics expect from an implementation of a model operation that the implementation

1. is able to take the materialized runtime models passed to it by the interpreter as an input to its behavior. That is, the implementation must be able to actually have those runtime models as input, which are defined as the input of the operation in the FLD.
2. is actually processing the input runtime models as specified by the model usage links in the FLD. For instance, if the FLD defines that a model operation creates a runtime model, the implementation of the operation must actually create the corresponding materialized runtime model. Otherwise, the runtime model cannot be used by any subsequent model operation of the feedback loop.
3. returns the materialized runtime models that are specified as output models of the corresponding model operation in the FLD. This allows the interpreter to pass the runtime models from one operation to another (*cf.* Section 6.4.5). For instance, if the FLD defines that a model operation creates and returns a runtime model, the implementation must return the created materialized model to the interpreter after the invocation. Otherwise, the interpreter is not aware of the created model and therefore cannot pass this model to another operation.
4. returns a return status of its execution, which can be mapped to an exit compartment of the corresponding model operation. The return status is used by the interpreter to identify the corresponding exit compartment of the model operation to continue the control flow in the FLD instance (*cf.* Section 6.4.3). Consequently, if the return status cannot be mapped to an exit compartment of the model operation, the interpreter does not know how to continue the control flow such that it raises a runtime exception.

An implementation of a model operation must behave according to these four aspects. These aspects are related to the abstract specification of a model operation in the FLD, that is, to the definition of the operation's input and output runtime models, the operation's processing of these models, and the operation's possible return states. This specification with the four aspects are similar to a contract that the implementation of a model operation has to fulfill such that the EUREMA execution semantics is not violated. Technically, the interpreter relies on this contract such that it is able to properly execute an FLD instance.

Stateless implementation. The implementation of a model operation should be *stateless*, which is a prerequisite for safe adaptation as discussed in Section 6.6. This does not mean that the implementation may not maintain any state at all. In contrast, it means that the state should be externalized from the implementation—particularly from the code or from any other formalism used for realizing the implementation—and stored in the runtime

models that are processed by the operation/implementation. Consequently, the requirement of stateless implementations is not that restrictive since it does not forbid any state but only prescribes how the state should be maintained. In this context, an advantage is that capturing the state in runtime models makes the state explicit such that it can be addressed during the design and at runtime, for instance, to safely adapt a feedback loop.

Moreover, engineers are not restricted by this requirement since EUREMA is open for any runtime model that is expressed with any language (*i. e.*, metamodel). Depending on the specific state or generally the knowledge of the feedback loop that should be captured, engineers may develop new or reuse existing languages to express the runtime models. This allows engineers to use and capture application-specific and user-defined knowledge in the feedback loops.

Termination and concurrency. The execution of an FLD instance is the sequential execution of operations, that is, the operations are executed one after the other and not concurrently (*cf.* Section 6.4.3). When executing a model operation, the corresponding implementation is synchronously invoked by the interpreter and therefore executed.

To ensure progress and finally termination of executing an FLD instance, the EUREMA execution semantics requires that the implementation of each model operation eventually terminates its execution. The notion of termination we require from the implementations of model operations means that no behavior should be active in the implementation, or technically that all threads used in the implementation have terminated. This notion of termination is required for avoiding any concurrency between operations (*cf.* Section 6.4.3) and for achieving quiescence to support safe adaptations of EUREMA models (*cf.* Section 6.6). It is the responsibility of the engineers providing the implementations to ensure the termination of executions of these implementations.

Since the operation of an FLD instance are executed one after the other, they are not executed concurrently. However, this does not exclude any concurrency within the implementation of a model operation. After being invoked by the EUREMA interpreter, the implementation may create and use multiple threads to concurrently execute its behavior. For instance, the implementation of an analyze operation may parallelize the analysis to concurrently check multiple constraints on a runtime model representing the adaptable software. However, the implementation and thus the engineer must assure that all of the threads terminate properly such that the whole implementation terminates as required (see discussion in the previous paragraph).

Summary. Summing up, implementations of model operations must satisfy three requirements such that they do not break the execution semantics of EUREMA. These requirements concern the specification of model operations in the FLDs, the stateless implementation of model operations, and the termination as well as concurrency of the implementations. At the technical level, EUREMA expects from implementations of model operations that they satisfy these requirements such that the EUREMA interpreter can execute the FLD and LD models according to the execution semantics discussed in this chapter.

To substantiate the discussion that we started with the requirements of model operations in this section, we focus in the following chapter on the model operations and their implementations in the context of EUREMA-based feedback loops. Particularly, we will discuss the solution space for realizing model operations and how the solution space supports satisfying the requirements of model operations.

7 MODEL-DRIVEN RUNTIME MODEL OPERATIONS

Having discussed the modeling and execution of EUREMA in the previous Chapters 5 and 6, we focus in this chapter on the individual adaptation activities of a feedback loop. Besides the EUREMA models specifying the feedback loops and the layered architecture of the self-adaptive software, engineers have to provide implementations for such activities. According to the idea of a runtime megamodel (*cf.* Section 4.2), we consider the adaptation activities of a feedback loop as *runtime model operations* as they operate on the runtime models employed in the feedback loop. Moreover, we consider operations that are *model-driven*, that is, MDE techniques are used for their development. Particularly, models are used to specify the behavior of the operations and they are kept alive at runtime to execute the specified behavior. Examples are evaluation or change models that specify and execute the behavior of the analyze respectively planning activities (*cf.* Section 4.1).

In this chapter, we elaborate the solution space of model-driven runtime model operations and discuss its influence on the changeability and execution of a feedback loop. We divided the solution space along two major dimensions. The first dimension distinguishes whether the implementation of an operation is based on code or driven by models. The second dimension considers operations that either work in a state-based or in an event-based manner. In the following sections, we discuss both dimensions and their impact on the requirements for model operations introduced in Section 6.7. Finally, we conclude the chapter with a summary.

7.1 CODE-BASED VS. MODEL-DRIVEN OPERATIONS

In the context of EUREMA, we assume that a model operation works on a reflection model that is causally connected to the adaptable software (*cf.* Section 4.1). The behavior of such an operation can be implemented and executed by code or by runtime models. In the former case, the behavior is hard-coded into an operation. This is illustrated in Figure 61a showing the operation Check for failures that analyzes the reflection model called Architectural Model to identify whether failures have occurred or not in the adaptable software. On the other hand, the behavior of an operation can be specified by models that are kept alive at runtime. The behavior is then performed by interpreting and executing these models. For instance, Figure 61b shows the same operation, however, whose behavior is explicitly specified by Failure analysis rules that are interpreted at runtime. That is, the implementation of the Check for failures operation obtains the rules as an input and it triggers an interpreter to run and apply those rules on the reflection model. For example, the rules can be expressed with the Object Constraint Language (OCL) [327], for which an execution engine is available and can be reused by the implementation of the operation[1].

The distinction between a code-based and model-driven operation is based on the kind of artifact, code or model, that is used at runtime to execute the operation. This distinction does not exclude any model-based development of code-based operations. For instance, the behavior of an operation can be specified by models, from which code will be generated and eventually used for the execution while the models remain a design artifact.

1 Eclipse OCL project: http://www.eclipse.org/modeling/mdt/ocl.

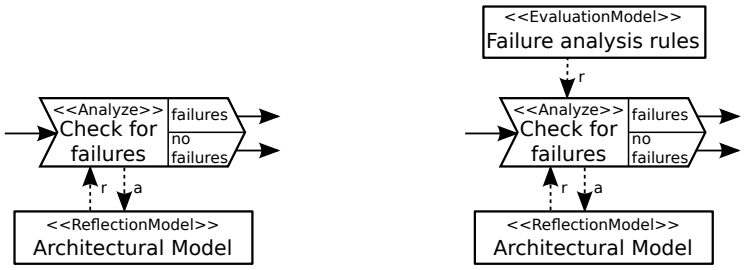


Figure 61: Operations that are (a) Code-Based and (b) Model-Driven.

The decision for a code-based or a model-driven operation is typically a trade-off. Among others, it might depend on the availability of programming or modeling languages that are appropriate for solving the problem at hand. For instance, to implement an analysis technique of a feedback loop, an imperative programming language can be an appropriate formalism if the analysis is an algorithmic problem. In contrast, if the analysis should identify a structural pattern in the reflection model, a modeling language such as Story Patterns (SPs) [155] is more appropriate as it allows to specify patterns declaratively and at the same level of abstraction as the reflection model. Hence, the availability of appropriate languages for the problem at hand as well as the characteristics of these languages such as the abstraction level and the programming/modeling style (imperative or declarative) are concerns determining the trade-off.

Despite these concerns, there is one major advantage of model-driven over code-based operations, which is particularly relevant for self-adaptive software where dynamic changes are inherent. Using explicit runtime models for specifying and executing operations leverages changeability of the adaptation activities of a feedback loop. The flexible nature of runtime models supports adapting the specification and hence the behavior of the operations. In contrast, hard-coded and compiled operations are not directly and as easily adaptable as models. Moreover, models are usually at a higher level of abstraction than code such that the adaptation of operations may happen at an appropriate level of detail, which further supports changeability. We exploit this changeability in EUREMA to adapt running feedback loops on-line by higher-layer feedback loops or off-line by maintenance (*cf.* Chapters 5.3 and 5.4). For instance, adapting the analysis strategy of the self-repair feedback loop requires changing or patching the (compiled) code of the code-based operation shown in Figure 61a while for the model-driven operation shown in Figure 61b, it only requires to replace the runtime model, that is, the Failure analysis rules.

In this context, model-driven operations may seem to be costly to develop since they require appropriate modeling languages and execution engines in order to express and execute the runtime models specifying their behavior. However, there exists plenty of languages and engines to express and process models in MDE, which can be reused in implementations of operations, and EUREMA is open for any of those.[2] Examples of such languages with execution support are OCL [327], EMF-IncQuery [414], SPs/SDs [155], Henshin [47], TGGs [220], and VIATRA [67, 418], which all can be used to query, check, and transform (runtime) models. Specifically, in our work we reused an existing model transformation engine for the monitor and execute activities to synchronize reflection models at

2 Technically, the only restriction is that EUREMA and the languages used for expressing runtime models share the same meta-metamodel such that they are unified in a common MDE infrastructure (see Section 2.1.1).

different levels of abstraction as well as an OCL engine for the analyze activity to check constraints on an abstract reflection model [22, 28]. Such execution engines are usually generic as they completely externalize the user-defined inputs in models that become runtime models in EUREMA—such as the Failure analysis rules in Figure 61b—and therefore amenable for adaptation. Consequently, such execution engines can be considered as reusable implementations for operations, which can reduce the development efforts.

Reuse and EUREMA's openness for integrating existing languages and execution engines potentially simplifies the development of model-driven operations. Such operations promotes changeability and hence the development of adaptable feedback loops. Therefore, we prefer model-driven to code-based operations in EUREMA although EUREMA is open for code-based operations (*e. g.*, the planning activity of the self-optimization feedback loop discussed in Section 5.2 is realized by a code-based operation).

7.2 STATE-BASED VS. EVENT-BASED OPERATIONS

The second dimension for the solution space of model operations distinguishes between event-based and state-based operations, which results in different ways of capturing and processing knowledge in a feedback loop. Depending on whether the operations of a feedback loop work in an event-based or state-based manner, we characterize the feedback loop either as event-based or state-based.

In general, event-based operations use only events to exchange and process knowledge about the adaptable software. Initially, the monitor operation of a feedback loop creates events capturing symptoms observed in the adaptable software. These events are processed along the analyze, plan, and execute operations to perform self-adaptation. Since the events capture just the symptoms of the adaptable software, they do not capture the complete state such as the current runtime architecture of the software but rather changes of the state. Focusing on changes of the state instead of the complete state enables incremental, change-driven processing, that is, individual changes are processed in contrast to the complete state of the adaptable software. However, such an approach requires a certain degree of locality, in which the symptoms can be handled. In this case, an event contains fractions of the state that correspond to the local context of a symptom such that the event is self-contained and operations can process it in a meaningful way. If the context grows in size or even requires comprehensive architectural knowledge of the adaptable software, the events as well as their construction and processing become more complex and more data needs to be transferred from one operation to another throughout the feedback loop.

For instance, if a symptom is isolated and related to a single component of the adaptable software that should be restarted by self-adaptation, the scope of the local context is small and events are a light-weight means to capture the context. This example is depicted in Figure 62a showing a fragment of an FLD. The queue and events in this FLD only illustrate the processing of events and they are not part of the EUREMA language. The Update operation monitors changes of the adaptable software and creates corresponding symptoms that are added to a queue. The subsequent Check for failures operation consumes and analyzes the symptoms to identify failures and the affected components in the adaptable software. It will then use events to notify the subsequent planning operation to project an adaptation for restarting these components (not shown in Figure 62a). This FLD fragment shows that event-based operations perform their tasks by processing and exchanging only events.

In contrast to event-based operations, state-based operations work on a runtime model, particularly a reflection model that represents the state of the adaptable software that is

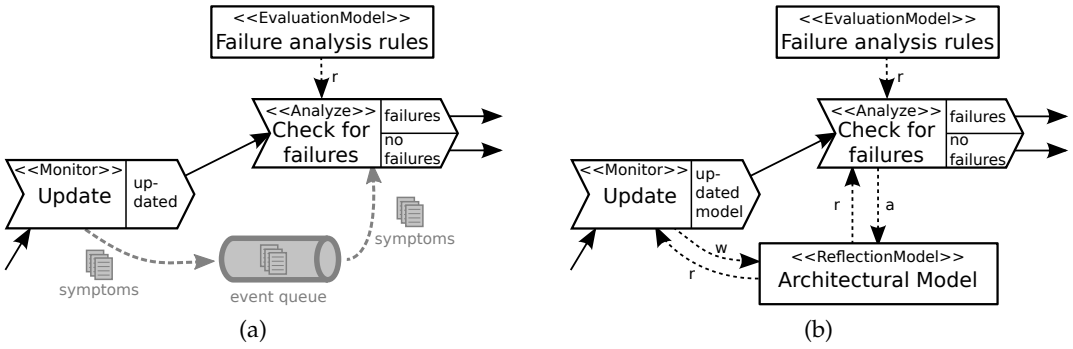


Figure 62: Operations that are (a) Event-Based and (b) State-Based.

relevant for self-adaptation. Typically, a reflection model provides an architectural view of the software (*cf.* Section 4.1). Therefore, the monitor operation observes the adaptable software and updates the reflection model according to these observations. The subsequent analyze, plan, and execute operations work on the updated reflection model to perform self-adaptation. Having a global view of the adaptable software in terms of the reflection model, there are no limitations concerning the locality of a change since the whole architecture can be analyzed and reconfigured. However, such an approach requires that each operation processes the complete model, hence it prevents any incremental processing.

For instance, the analyze operation has to check the complete model in order to identify any need for adaptation. This example is illustrated in Figure 62b showing a fragment of an FLD. The Update operation monitors the adaptable software and updates accordingly the reflection model called Architectural Model. The subsequent Check for failures operation analyzes the Architectural Model to identify any failures that might have occurred in the adaptable software. Therefore, it has to analyze the complete model since it has no hints on the recent model updates performed by the Update operation. As one solution, the analyze operation could produce a diff between the latest reflection model and a snapshot of the reflection model created during the previous run of the feedback loop to obtain the model updates, which, however, also requires processing the complete model. If the analyze operation identifies failures, it annotates and thus updates the model that will be processed afterwards by the planning operation (not shown in Figure 62b). This example shows that state-based operations perform their tasks by processing and sharing runtime models capturing knowledge about the adaptable software.

Summing up, both variants of event-based and state-based operations have advantages and disadvantages. While event-based operations basically support the incremental, change-driven processing in a feedback loop, they require identifying and capturing the knowledge (*i. e.*, the changes and their contexts) that is relevant for self-adaptation in events that might get complex in terms of scope and size of the captured knowledge. On the other hand, state-based operations do not support an incremental processing but they ease the handling of the knowledge in a feedback loop since the runtime models provide comprehensive knowledge about the adaptable software (*e. g.*, the software's runtime architecture), which is shared by all operations of a feedback loop.

So far we discussed state-based feedback loops in this thesis because a basic assumption of EUREMA is that the feedback loops heavily use runtime models, particularly reflection models that are causally connected to the adaptable software (*cf.* Section 4.1). However, to obtain the benefits of event-based and state-based operations, EUREMA combines both

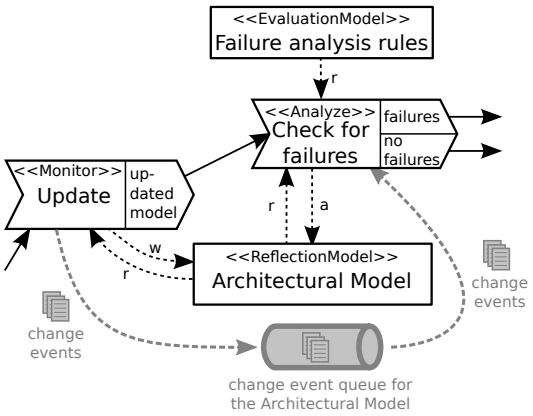


Figure 63: Operations that are Event- and State-Based.

variants.[3] The goal is to support an incremental processing of model operations—a key need for self-adaptation that "comes into play because often changes are local to restricted parts" of the adapatable software [180, p. 369]—that work on runtime models, hence to improve the runtime performance of executing feedback loops operating on runtime models.

More specifically, EUREMA fully supports state-based operations to benefit from the rich semantic base of runtime models, which avoids the construction and processing of complex events carrying the context of a state change (*e. g.*, of a symptom). Instead, the context of a change is represented in the runtime models. To enable an incremental, change-driven processing of state-based operations, EUREMA supports *change events* that are related to runtime models and that notify about changes of these models. Such events are provided and maintained by EUREMA and they can be used by operations to identify locations in a runtime model, from where the processing should start. This may avoid the need to process the complete model, particularly if only a small fraction of the model has changed.

Moreover, we conceive a better changeability of self-adaptive software when using runtime models to capture the state and thus the contexts of changes. For instance, if the context of a change type should be extended, it is only required to adjust the runtime model to cover the extended context. However, if the model is expressive enough to cover the extended context, no adjustments are required. In contrast, if we rely only on events, we have to adjust the individual event types of all operations, thus resulting in a higher number of adjustments. Additionally, a runtime model centrally captures the context of changes instead of scattering the context in multiple events throughout the feedback loop.

An example of such event- and state-based operations is illustrated by the FLD fragment in Figure 63. The Update operation monitors changes of the adaptable software and updates accordingly the Architectural Model. When changing the Architectural Model, change events are automatically generated and added to a queue. For each model change, an event is created that points to the changed element of the model. The subsequent Check for failures operations consumes these change events and uses them to jump directly to the locations in the Architectural Model where the changes have occurred. Starting from these locations, the operation analyzes the model for any failures. For instance, if a component of the adaptable software has stopped operating for any reason, the element in the Architectural Model representing the corresponding component is changed to reflect this situation. Using the change event, the analyze operation can directly jump to this element in the model and

3 In MDE terms, EUREMA combines principles of change- and state-based model operations (*cf.* Section 2.1.1)

start the analysis there. Otherwise, the analyze operation has to check all components in the model to find the ones that are not properly operating.

As shown in Figure 63, the Check for failures operation annotates the Architectural Model with the analysis results, for instance, by marking components that are affected by failures. Annotating the model automatically causes change events that are also added to the queue. The subsequent planning operation can consume these events to start the planning process right away with the affected components in the model instead of searching the complete model for these components. This example illustrates that combining runtime models and change events can potentially improve the runtime performance of operations by enabling an incremental, change-driven processing of the models. This aspect will be further discussed in the context of the evaluating EUREMA in Section 9.4.

Based on this idea, EUREMA provides generic support for change events of any runtime model. That is, the EUREMA interpreter maintains an individual queue of change events for each runtime model used within a feedback loop. This also applies to the EUREMA models (*e. g.*, the FLD instances to execute the feedback loops) that can be used as reflection models when layering feedback loops (see Section 5.3). Moreover, the interpreter instruments all runtime models to observe their changes. For each model change, an event is created that points to the changed element of the model and that contains information about the change such as the changed feature (*i. e.*, which attribute or reference is affected by the change), the feature values before and after the change, and the type of the change (*e. g.*, whether an element or a reference has been added or removed from the model and whether an attribute value has been set or unset). The interpreter puts the created event into the change event queue of the corresponding model. The actual changes of the runtime models are performed by operations. Each operation that has a runtime model as an output may change the model, which is specified in the FLDs by labeling the output relationship with w or a for writing or annotating the model. Considering the FLD fragment in Figure 63 on the previous page, the Update operation has the Architectural Model as an output and the output relationships is labeled with w, that is, the operation writes or more specifically updates the model. The EUREMA interpreter observes these updates, creates the change events, and add these events to the queue maintained for the Architectural Model. Likewise, the Check for failures operation may annotate and therefore also change the model, which causes corresponding change events that are added to the same queue.

In this example, the interpreter also maintains a queue for the other runtime model, that is, for the Failure analysis rules. However, this model is only read but not modified by any operation such that no changes of this model and thus no change events will occur.

While the EUREMA interpreter automatically produces the change events while operations change the runtime models, the operations themselves may consume these events from the queues. Typically, an operation consumes those events which are caused by model changes performed by previously executed operations. In general, any operation that has a runtime model as an input has also access to the change event queue of the runtime model to consume the corresponding change events. The inputs of operations in terms of the runtime models are specified in FLDs while each input relationship is labeled with an r meaning that the operation reads the model. For instance, the Check for failures operation has the Architectural Model as an input and it therefore may consume the change events for this model (see Figure 63 on the previous page).

Consuming a change event does not imply that the event is removed from the queue. This would otherwise restrict the number of readers of an event to one such that, for instance, an event caused by the monitoring operation can only be read by one of the analyze and plan-

ning operations but not by both of them. To allow flexible solutions of consuming events, the EUREMA interpreter does not automatically remove change events from queues but shifts this responsibility to the implementations of the operations. Consequently, the operations of a feedback loop decide if and when a change event is removed from its queue. Different scenarios are conceivable here. For instance, if each operation causes change events and an operation only reads the events caused by the previous operation, an operation can immediately remove the consumed change events from the queue. On the other hand, if the change events should be preserved in the queue such that multiple operations of a feedback loop can read them, they are oftentimes removed from the queue by the last operation of the feedback loop to conclude the current run and to enable a fresh, next run of the feedback loop. Nevertheless, it is also conceivable that change events remain in the queue across multiple runs of the feedback loop. Essentially, it is the responsibility of the implementations of the operations to selectively read the change events that are of interest from the queues, to process these events, and if needed to remove these events from the queues. In this context, developers must assure that an operation does not endlessly cause and process its own change events. Otherwise, the operation might not terminate, which is, however, required by the execution semantics of EUREMA (*cf.* Section 6.7).

Finally, it has to be noted that the use of change events is optional in EUREMA. If they are not used, the feedback loop is state-based since the operations process the runtime models in a non-incremental manner. In this case, the tracking of model changes by the EUREMA interpreter can be deactivated such that no change events are produced and no change event queues are maintained. If the operations should process the runtime models incrementally to improve the runtime performance of the overall feedback loop, EUREMA's support for tracking model changes can be used. The mechanism to track model changes, to produce change events, and to maintain the queues of change events is *generic*.[4] On the one hand, this means that the mechanism works for any user-defined runtime model expressed with any metamodel. On the other hand, developers do not have to adjust the runtime models or the metamodels to enable the change tracking by EUREMA.

Within this thesis we do not specifically evaluate the runtime performance of operations that are only state-based or that are state- and event-based.[5] A decision to choose one variant depends on the size of the model and on the algorithms used by the operations, for instance, on the analysis conducted by the analyze operation. The size of the model as well as the algorithms are out of EUREMA's scope since they concern the implementations of the operations and models. Consequently, EUREMA does not prescribe one variant of operations but leaves the decision to the developers. However, examples are conceivable where an incremental, change-driven processing of a runtime model does not pay off, for instance, when in all situations the complete model is processed. In this case, there needs to be no benefit of directly jumping to a specific element in the model that has changed. On the other hand, if all operations of a feedback loop can handle a symptom locally, an incremental, change-driven processing of the model may pay off and improve the runtime performance of the operation and of the overall feedback loop.

From EUREMA's point of view, it is not critical to leave the decision of choosing between state-based and state-/event-based operations to the developers since the execution semantics of EUREMA (*cf.* Chapter 6) is not affected by the decision. In contrast, the decision

4 Technically, the change tracking mechanism of EUREMA is based on the notification feature of the Eclipse Modeling Framework (EMF), which provides notifications about individual changes of any EMF-based model [402].

5 However, we will evaluate the runtime performance of the whole feedback loop in the context of comparing alternative solutions to the self-healing and self-optimization of mRUBiS in Section 9.4.

only influences how an implementation of an operation processes the runtime models but not how the operation as a whole is executed as part of an FLD instance.

Nevertheless, to preserve the execution semantics of EUREMA, implementations of model operations must satisfy several requirements that we introduced in Section 6.7. In the following section, we discuss the two dimensions of the solution space for model operations (*i. e.*, code-based vs. model-based and state-based vs. event-based operations) with respect to these requirements.

7.3 IMPACT ON SATISFYING THE REQUIREMENTS FOR MODEL OPERATIONS

In the context of the EUREMA execution semantics, we discussed requirements for model operations (see Section 6.7), which are about satisfying the specification, stateless implementation, as well as termination and concurrency. Implementations of such operations must satisfy these requirements such that the EUREMA execution semantics is not violated. In the following, we discuss the impact of the previously discussed dimensions of the solution space for model operation on satisfying these requirements. That is, certain dimensions make satisfying these requirements either more easy or more difficult.

Satisfying the specification. The first requirement concerns the specification of a model operation in an FLD, which comprises the input and output models, the processing of the models, and the return status. Such a specification of an operation must be satisfied by the implementation of the operation. The solution space for implementations of operations does not impact the difficulty of satisfying this requirements. Regardless whether an operation is implemented in a code-based or model-driven as well as in a state-based or event-/state-based manner, the implementation must satisfy the specification. In all cases, the implementation must be able to have the specified runtime models as input, to process them as specified (*e. g.*, a model can be created, read, annotated, written), to have the specified runtime models as output, and to return a status. Consequently, satisfying this requirement is not eased or hindered by the decision for one implementation variant of the solution space.

Stateless implementation. The second requirements demands stateless implementations of model operations, which is a prerequisite for safe adaptations of feedback loops. With EUREMA, we generally assume that operations work on a runtime model, particularly a reflection model that is causally connected to the adaptable software and that can be enriched with working data of the operations. Consequently, any implementation of an operation can externalize and store its state in such a reflection model, which makes the implementation stateless.

Compared to code-based operations, changeability can be improved by model-driven operations that further externalize the behavioral specification of the operations into runtime models (*e. g.*, see the Failure analysis rules in Figure 61b on Page 140 that have been externalized from the code-based operation shown in Figure 61a on Page 140). To adapt the behavior of an operation, the corresponding runtime models have to changed or replaced, which is usually easier than adapting the (compiled) code of code-based operation.

Concerning the use of runtime models with or without change events, this requirement is not affected. The state of an operation such as the working data is captured only in runtime models (*i. e.*, typically the reflection models) while the change events are just notifications about changes of the models, which references the models. Consequently, the use of events does not contribute to achieving a stateless implementation of an operation.

Termination and concurrency. The third requirement states that the execution of a model operation must eventually terminate. Thus, developers must assure that the implementations of operation eventually terminate, which might get more complicated if multiple threads are used for the execution.

While assuring termination does not differ for the dimension distinguishing code-based and model-driven operations, it can become relevant for the other dimension. If an operation is event- and state-based, the implementation—once triggered by the EUREMA interpreter—may use the change events to trigger its internal behavior. However, executing its behavior, the implementation may cause new change events that may be used again to trigger its internal behavior. This requires that the implementation carefully handles and consumes the change events (*e. g.*, by consuming only those change events that have been caused by other operations) to assure termination of its execution. In contrast, an implementation of an operation that is only state-based typically triggers once its internal behavior to process the whole state after it has been triggered by the EUREMA interpreter.

Consequently, assuring termination of the execution can be more difficult for operations that use change events in addition to the runtime models.

7.4 SUMMARY

To conclude the discussion of the solution space for implementations of model operations in EUREMA, model-driven operations promise a better changeability of their behavior than code-based operations since models are typically easier to change than (compiled) code. However, code-based operations can be the better choice if there is no modeling language available to intuitively create runtime models that specify and execute the operations behavior for a specific problem.

Using runtime models, especially reflection models, to capture the state of operations eases the stateless implementation of the operations and therefore achieving safe adaptations of feedback loops. We consider by default such state-based operations in EUREMA. A disadvantage of such operations is their non-incremental processing of the runtime models, that is, they have to process with every execution the whole state captured in the models. However, to improve the runtime performance of operations, an incremental, change-driven processing can be achieved by exploiting change events. Such events notify about changes of models, which can be used as starting points for efficiently processing the models. While the EUREMA interpreter provides generic support for change events (*i. e.*, it automatically observes the runtime models for changes, creates change events, and maintains these events in queues for each runtime model) without requiring any engineering support from developers, developers have the responsibility of assuring termination of executions of the individual operations. This responsibility requires from developers to carefully handle the change events in the implementations and to avoid endless triggering of implementation-internal behavior by events.

Having discussed major dimensions of the solution space for implementing model operations in EUREMA, we discuss examples of such implementation in the following chapter that covers as well the implementation of EUREMA.

Part III

VALIDATION AND CONCLUSION

In this part of the thesis, we discuss the implementation of EUREMA and of an experimentation environment, which we use for the evaluation. Afterwards, we comprehensively evaluate EUREMA by assessing the design and expressiveness of the language, conducting experiments with a running system, comparing EUREMA-based solutions with alternative solutions, and discussing how well EUREMA covers quality attributes and fulfills the requirements for engineering self-adaptive software. Moreover, we discuss related work to contrast EUREMA with the state of the art in engineering self-adaptive software. Finally, we conclude the thesis and provide an outlook on future work.

8 IMPLEMENTATION

In this chapter, we discuss the implementation of EUREMA and of the technical framework that we use to experiment with EUREMA. The EUREMA implementation is loosely integrated in the framework and can therefore be easily used in other technical contexts[1]. The implementation architecture of the framework follows the external approach to self-adaptive software (*cf.* Section 2.2.3) and consists of a stack of elements shown in Figure 64.

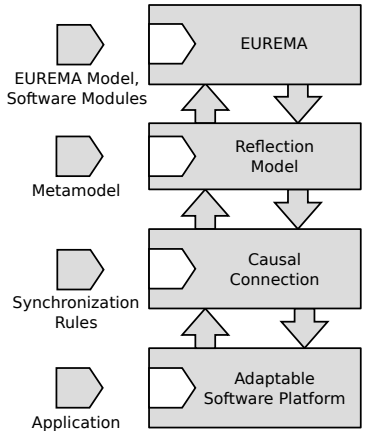


Figure 64: Implementation Architecture of the Framework.

The framework is *generic*, that is, it is not limited to one specific adaptable software and one specific self-adaptation capability such as self-optimization or self-healing. Consequently, developers using the framework have to customize each element of the framework.

- The *Adaptable Software Platform* is the runtime environment of the adaptable software. Developers have to provide the *Application* that is going to be managed by the feedback loop(s). The platform automatically instruments the application with sensors and effectors such that the application becomes adaptable. This disburdens the application developers who can focus on the business instead of the adaptation logic.
- The *Causal Connection* provides means to synchronize the adaptable software and the *Reflection Model*, that is, any change of the software is reflected in the model, and vice versa (*cf.* Section 2.1.5). A particular aspect is that the causal connection works incrementally in both directions such that only the changes are propagated. To specify the causal connection, developers have to provide *Synchronization Rules*.
- The *Reflection Model* is a causally connected runtime model of the adaptable software. It is maintained and synchronized with the software by the *Causal Connection* and the basis for the feedback loop(s) to perform self-adaptation. The language to express this model is user-defined such that developers have to provide a *Metamodel*. This metamodel and the *Synchronization Rules* determine the content and construction of the reflection model such that the model is completely user-defined.

1 As an example, EUREMA has been integrated into the *SimuLizar* framework [61], which addresses the performance analysis of self-adaptive systems, to model and execute feedback loops [349].

- The topmost element of the framework is EUREMA with its language and interpreter to model and execute feedback loops. Development with EUREMA results in an *EUREMA Model* that specifies the feedback loops and the layered architecture of the self-adaptive software (see FLDs and LD in Chapter 5) and in a set of *Software Modules* that implement the model operations (*cf.* Chapter 7). If the operations are model-driven, the software modules are complemented by runtime models (*cf.* Section 7.1). The EUREMA model and the software modules are both provided by the developer and eventually executed by the EUREMA interpreter (*cf.* Chapter 6).

The whole framework is based on the Eclipse Modeling Framework (EMF)[2] that provides the required MDE infrastructure for the model-driven engineering of self-adaptive software with runtime models and runtime megamodels. Like EMF, we implemented the framework in *Java* [194, 278]. As the adaptable software platform, we support an Enterprise Java Beans (EJB) [132] server that hosts the application provided by the developer. Hence, the adaptable software in the framework can be *any* EJB-based software.

Based on this overview of the framework, we discuss in the following sections bottom-up the technical details of the individual elements of the framework. Finally, we briefly present a simulator for the reflection model that supports early testing of EUREMA feedback loops without having to run the underlying causal connection and adaptable software platform.

8.1 ADAPTABLE SOFTWARE PLATFORM

As previously mentioned, the adaptable software platform is an EJB environment. Its implementation architecture is depicted in Figure 65. It consists of *GlassFish*[3], which is an application server for EJB-based applications, and *mKernel* [90, 91], which comprises *Preprocessor* and a *Plugin* for GlassFish. Developers provide an *Application* that is implemented with EJB technology targeting the GlassFish server. An example of such an application is mRUBiS [18] that we introduced in Section 4.5. In general, EJB is a specification [132] for developing component-based software [409] targeting enterprise applications. Hence, such applications consist of multiple components and interconnections among those components. In the context of the framework, the application realizes the business logic of the self-adaptive software. It does not need to be adaptable such that developers can focus on the business logic and do not have to take any adaptation concerns into account.

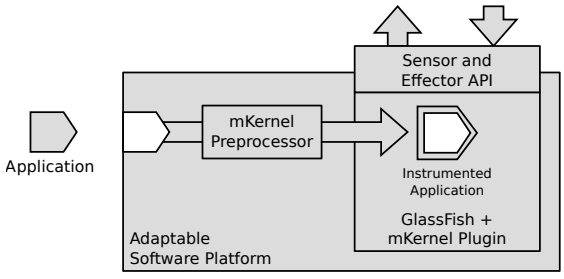


Figure 65: Implementation Architecture of the Adaptable Software Platform.

2 Eclipse Modeling Framework (EMF): See http://www.eclipse.org/modeling/emf/ and [402].

3 *GlassFish* is the reference implementation of the Java Enterprise Edition (Java EE) specification and therefore of the included EJB specification: https://glassfish.java.net/. We use version 2.1.1 of GlassFish that is the reference implementation of Java EE version 5 [385] and therefore of EJB version 3.0 [132].

Before the provided application is deployed on the GlassFish server, the mKernel preprocessor automatically instruments the application with sensors and effectors. This instrumentation makes the application observable and adaptable. For instance, the sensors and effectors support observing and adapting the application's runtime architecture, particularly, the individual components with their parameters and connections to other components. Consequently, mKernel enables parameter and structural adaptation, which are the two general means to adaptation [292]. Moreover, it supports safe adaptations of the application by realizing the quiescence mechanism proposed by Kramer and Magee [259].

The instrumented application is afterwards deployed on the GlassFish server that is extended with the mKernel plugin. This plugin is the runtime environment for the sensors and effectors of the application and it provides them as an Application Programming Interface (API) [3] (see *Sensor and Effector API* in Figure 65). This API supports observing and adapting the runtime architecture of the application. Moreover, it includes functionality for performance monitoring that is realized by GlassFish according to the Java EE performance data framework [226]. Thus, GlassFish and mKernel jointly provide the sensor and effector API that can be used by programs, external to the server, to observe and adapt the application. For instance, we used the API in previous work to realize programmatically different reconfiguration strategies for exchanging individual components of an application [20], among others, by keeping the current and updated versions of components running during reconfiguration (*cf.* [122]).

However, the sensor and effector API is platform-specific and at a rather low level of abstraction as it refers to the technical concepts of the EJB platform. Therefore, we aim for raising the level of abstraction with the framework such that feedback loops may operate on a higher-level, platform-independent reflection model to perform self-adaptation. It is the task of the *Causal Connection* to bridge the abstraction gap between the low-level, platform-specific API and the higher-level, platform-independent reflection model. Technically, the causal connection uses the sensor and effector API to maintain and synchronize the reflection model with the application, which will be discussed in the following section.

From the framework's point of view, the adaptable software platform can be exchanged and alternative solutions to the EJB platform such as *OSGi*[4] or *FRASCATI* [380] are conceivable. The framework only requires that the platform provides a sensor and effector API for monitoring and adapting the application, ideally at the architectural level, such that the causal connection can be built upon this API. Finally, the platform should not be restricted to one specific application. In our case, the platform can host any EJB-based application.

8.2 CAUSAL CONNECTION AND REFLECTION MODEL

In this section, we describe the implementation of the causal connection that synchronizes the reflection model and the adaptable software (see Figure 66). In general, the causal connection refers to synchronizing a runtime model and a running software such that changes in of them are reflected in the other (*cf.* Section 2.1.5). The causal connection of the framework does the same but in a special way because of three reasons:(1) to cope with the abstraction gap between the adaptable software and the reflection model, (2) to be open for user-defined reflection models and metamodels instead of prescribing the metamodel or how the reflection model is constructed and synchronized with the software, (3) to provide an efficient solution for the causal connection by leveraging incremental techniques. To address these issues, we adopt an MDE approach and use model synchronization techniques.

4 OSGi Alliance: The Dynamic Module System for Java: https://www.osgi.org/.

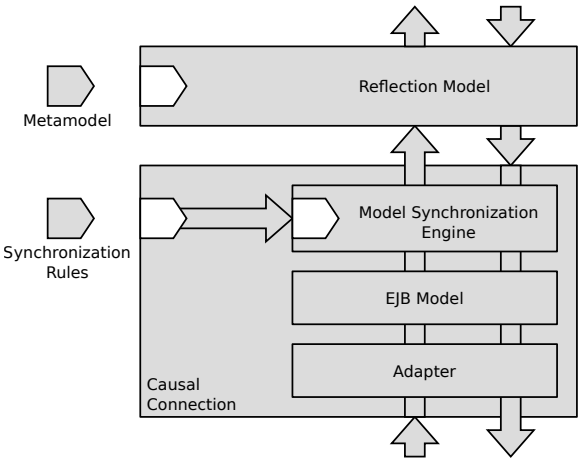


Figure 66: Implementation Architecture of the Causal Connection and Reflection Model.

The *Adapter* uses the sensor and effector API provided by the adaptable software platform to maintain the *EJB Model* that is causally connected to the software (see Figure 66). The corresponding EJB metamodel is shown in a simplified[5] version in Figure 67. It is at the same abstraction level as the API such that both of them are platform-specific since they are concerned with EJB concepts to monitor and adapt the underlying software.

In this context, the metamodel is used to describe the application at three levels: type, deployment, and instance level. The type level captures the results of the application development such as the developed types of enterprise beans (see classes drawn white in Figure 67). Such types can be instantiated and deployed to a server, which includes configuring the application such as connecting required and provided interfaces (see EjbConnector wiring an EjbReference to an EjbInterface). The deployment level comprises the classes drawn light-gray in Figure 67. Finally, the instance level (see classes drawn dark-gray in Figure 67) captures concrete interactions within the running application such as calls among bean instances. A detailed description of these EJB concepts can be found in [90, 3].

The EJB model is an instance of this metamodel and it is used to monitor the adaptable software. The model can also be changed to correspondingly adapt the software. For instance, to adapt the interconnection among beans, connector elements can be removed and added. The adapter synchronizes the EJB model with the adaptable software by using the sensor and effector API. Particularly, it uses the sensors to observe the running software and to reflect changes of the software in the EJB model. Similarly, it uses the effectors to enact adaptation to the running software according to the changes of the EJB model.

The synchronization in both directions (*i. e.*, from the adaptable software to the model, and vice versa) is eased since the EJB model and the API are at the same level of abstraction such that there is a one-to-one mapping between the model and API elements. Thus, the model basically mirrors the capabilities of the API and instead of the API, the model can be used as the interface to monitor and adapt the running software.

To make the synchronization runtime-efficient, the adapter exploits the push-oriented sensor API that emits events notifying about structural changes of the adaptable software to incrementally update the EJB model. However, the sensors for behavior monitoring that

5 To illustrate the extent of the simplification, the complete EJB metamodel consists of 67 classes while the simplified version shown here consists of only 29 classes. Among others, the classes addressing the security, transaction, timer, interceptor, quiescence, and performance concerns are hidden. Moreover, the simplified version hides several associations and does not show any attributes and operations.

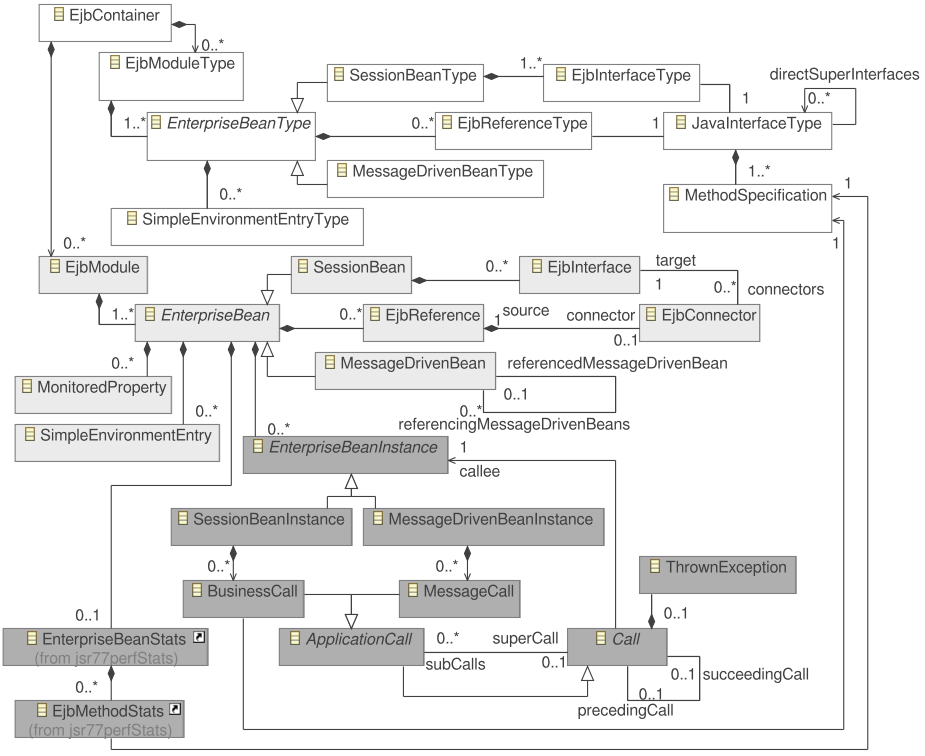


Figure 67: Simplified EJB Metamodel.

establishes the instance level (*i. e.*, bean instances and calls among them) in the EJB model are provided in a pull-oriented fashion—to avoid the proliferation of events—and therefore require more costly batch updates of the model. Nevertheless, if the instance level is not of interest, the corresponding sensors can be deactivated to improve efficiency of monitoring and updating the model. Similarly to the incremental updates of the model, the adapter observes adaptations of the EJB model and maps them incrementally to invocations of the effector API to eventually enact an adaptation to the software. Thus, as far as possible the adapter synchronizes the EJB model and the adaptable software incrementally.

We designed the EJB model to be *generic* such that it can be used to (re)configure any EJB-based software as well as to be *comprehensive* to provide a rich semantic base for different concerns such as performance (see EnterpriseBeanStats and EjbMethodStats in the EJB metamodel), failures (see ThrownException in the EJB metamodel), and security (not shown in the simplified version of the EJB metamodel) which support different self-adaptation capabilities such as self-configuration, self-optimization, self-healing, and self-protection.

The advantage of having such a model in contrast to a code-based sensor and effector API is that the model is amenable for MDE techniques to easily create and maintain further abstractions.[6] On the one hand, we may abstract from the adaptable software platform and thus from the solution space of the software to obtain runtime models that are platform-independent and that are targeting problem spaces.[7] On the other hand, we may abstract from concerns that are not of interest for the addressed self-adaptation capabilities. For

6 We envision that ideally the adaptable software platform directly provides its sensor and effector capabilities as a runtime model instead of a code-based API. This would avoid the need for the adapter in our framework.

7 A similar abstraction is made in MDA that distinguishes a Platform Independent Model (PIM) and a Platform Specific Model (PSM) in software development while the PIM is refined to the PSM (*cf.* Section 2.1.2). Thus, we may consider the reflection model as a PIM and the EJB model as a PSM, which are, however, used at runtime to provide views of the adaptable software at different levels of abstraction.

instance, focusing on self-healing does not require to capture performance concerns in the model. Such an abstraction results in a concern-specific view of the application.

The *Reflection Model* in our framework shown in Figure 66 on Page 154 is such an abstract runtime model. The type of the reflection model is not prescribed by the framework such that developers can plug-in their own *Metamodel*. Thus, developers decide about the abstraction level and content (*e. g.*, which concerns should be captured) of the model.

In this context, we developed the *CompArch* metamodel to conduct experiments with EUREMA. A simplified[8] version of it is shown in Figure 68. It supports describing the architecture of the adaptable software in a platform-independent way. Therefore, it considers the types of the software, that is, the ComponentTypes that require and provide InterfaceTypes and that may have configuration options in terms of ParamterTypes. An interface type consists of methods represented by MethodSpecifications. A deployed software is considered as instances of the types and captured as a Tenant architecture comprising Components with RequiredInterfaces, ProvidedInterfaces, and configuration Parameters. Connectors wire required and provided interfaces and a MonitoredProperty is an observable application-specific property of a component. Finally, the metamodel captures two concerns: Failures and PerformanceStats. Both refer to a provided interface and method since they capture occurred failures or the runtime performance of using the method of the interface.

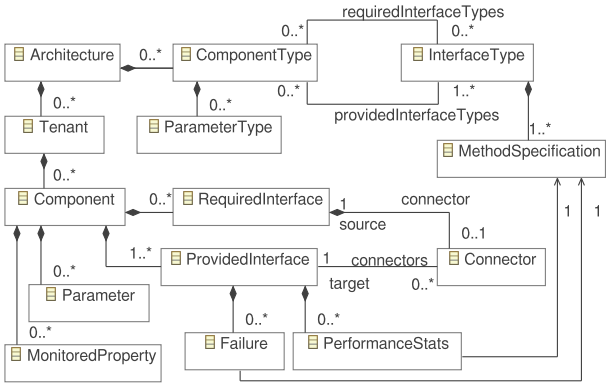


Figure 68: Simplified *CompArch* Metamodel of the Reflection Model.

The motivation for the CompArch metamodel is to obtain a runtime model reflecting the adaptable software, which is less complex than the EJB model and therefore easier to use for feedback loops. For instance, the CompArch metamodel eases the mapping of failures to the affected components (by navigating from the Failure to the ProvidedInterface to the Component) in contrast to the EJB metamodel that requires a more complicated mapping and navigation from a ThrownException to the affected EjbModule (*cf.* metamodels in Figures 67 and 68). Moreover, we aim for an abstract runtime model that is independent of the EJB platform to apply feedback loops that generally address component-based software, which paves the way for reusing (parts of) the feedback loops across different platforms.

To leverage feedback loops that operate on the reflection model rather than on the EJB model, both models must be causally connected. That is, changes have to be propagated from the EJB model to the reflection model for monitoring, and in the opposite direction for adapting the software, which calls for a bidirectional model synchronization.

8 The complete CompArch metamodel consists of 17 classes and the simplified version shown here of 14 classes. The simplified version further hides several associations and does not show any attributes and operations.

In our framework, the *Model Synchronization Engine* realizes the propagation of changes among both models (see Figure 66 on Page 154). It is parameterized with *Synchronization Rules* that define how the changes are propagated between both models in both directions. Since the metamodel of the reflection model is user-defined, the framework cannot prescribe the synchronization but requires corresponding rules from the developer. Technically, we use a model transformation engine [186, 188] that is based on Triple Graph Grammars (TGGs). It supports the incremental propagation of changes among two models in both directions (*i. e.*, it supports the bidirectional synchronization of models) while the two models are expressed with different metamodels (*cf.* Section 2.1.1 giving a brief introduction to model transformation). Thus, the provided synchronization rules are TGG rules that specify declaratively at the level of metamodels how two models as instances of these metamodels can be transformed and synchronized with each other. More specifically, they define how concepts of one model are reflected in the other model, and vice versa.

For the CompArch metamodel, an example TGG rule is shown in Figure 69. Such a rule combines three graph grammars: one grammar describes a source model (in this case the EJB model), a second one a target model (in this case the CompArch model), and a third one a correspondence model that stores relationships between corresponding source and target model elements. Concerning the shown example rule, elements on the left refer to the source model, elements on the right to the target model, and elements in the middle constitute the correspondence model. The elements drawn black form the application context of the rule, that is, these elements must already exist in the models before the rule can be applied. Applying the rule then creates the elements annotated with ≪create≫.

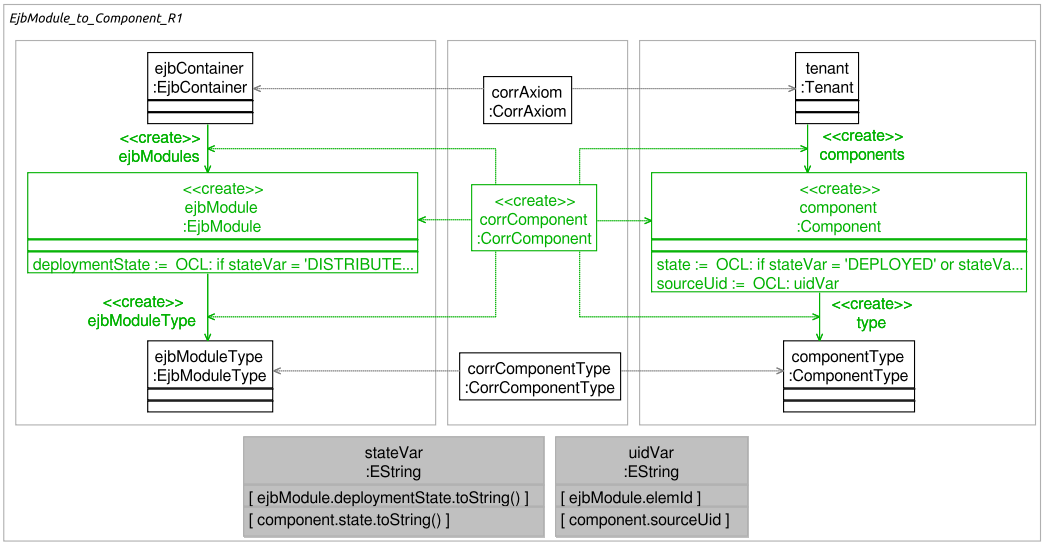


Figure 69: TGG Rule Defining the Synchronization of EjbModules and Components.

For the specific example in Figure 69, the rule defines that for an EjbModule that is created in the EJB model and associated to the EjbContainer and its EjbModuleType a Component is created in the CompArch model and associated to its Tenant and ComponentType. Thereby, a correspondence element (CorrComponent) is created that maintains the mapping of the corresponding EjbModule and Component elements. The other correspondence elements of the rule are used to identify the corresponding elements to which the newly created elements are associated. For instance, the corresponding and newly created EjbModule and

Component elements should be associated to the EjbModuleType and ComponentType that are also corresponding based on the CorrComponentType element. Moreover, the rule exemplifies the synchronization of attributes (*cf.* [267]), which is done by defining variables and OCL expressions that reference these variables. For instance, the rule synchronizes the deploymentState of the EjbModule with the life cycle state of the Component. The same rule also specifies the synchronization in the opposite direction, that is, how the EJB model should be modified to reflect the creation of a Component in the CompArch model or modified attribute values of a Component. Finally, besides the synchronization of created elements, the same rule also specifies the synchronization of elements that are removed from the model. That is, if an EjbModule is removed from the EJB model, the corresponding Component is removed from the CompArch model, and vice versa.

For the CompArch metamodel, we specified in total 20 TGG rules, each one similar to the shown example, to specify the synchronization between any CompArch model and any EJB model. Table 6 lists the correspondence relationships between the simplified versions of the CompArch and EJB metamodels (*cf.* Figure 67 on Page 155 and Figure 68 on Page 156). Based on these relationships, the EJB metamodel supports white-box views of the application as it considers the EjbModules with their constituting EnterpriseBeans. In contrast, the CompArch metamodel supports black-box views as it abstracts from the EnterpriseBeans and only represents the EjbModules as black-box Components. The step from a white- to a black-box view determines the different structural abstraction levels of both metamodels.

In general, the correspondence relationships between the metamodel elements does not have to be one-to-one mappings (*e. g.*, the EjbContainer element is mapped to two elements in the CompArch metamodel) and it does not have to be complete, that is, some elements of one metamodel may not have any corresponding element in the other metamodel (*e. g.*, the EnterpriseBean is not mapped and represented in the CompArch metamodel). Thus, developers can specify the synchronization entirely according to their needs.

Based on the declarative TGG rules, the transformation engine automatically generates operational rules to execute the bidirectional synchronization. For developers, there is usually no need to write any imperative code in addition to the TGG rules, which eases the development of the causal connection. Moreover, the transformation engine efficiently synchronizes both models since it incrementally processes the models to identify and synchronize changes among them. The engine is driven by events notifying about changes of both models to leverage the incremental and therefore runtime-efficient synchronization [185].

Finally, the execution of the causal connection (*i. e.*, the adapter and the model synchronization engine) can be triggered on-demand in each direction, either bottom-up from the adaptable software to the EJB model and eventually to the reflection model to support the monitoring activity as well as top-down in the opposite direction to support the execution activity of a feedback loop. Thus, the feedback loop and typically its monitoring and execution activities control when the causal connection is enacted such that it can operate on a consistent view of the adaptable software during one of its runs. Moreover, the on-demand execution of the causal connection is based on a loose coupling between the models and the software such that each individual change does not have to be immediately propagated when it occurs. In contrast, the causal connection is able to propagate a set of changes on-demand, which supports its flexible use by feedback loops.

In this thesis, the discussion of our model-driven approach to the causal connection is kept rather short as its contributions are not directly related to EUREMA. Therefore, we refer to [22, 28–30] for more details on the platform-specific EJB model, adapter, model synchronization engine, and abstract reflection models. However, we briefly summarize

Table 6: Correspondence Relationships of the Simplified CompArch and EJB Metamodels.

EJB Metamodel Elements	CompArch Metamodel Elements
EjbContainer	Architecture, Tenant
JavaInterfaceType	InterfaceType
MethodSpecification	MethodSpecification
EjbModuleType	ComponentType
EnterpriseBeanType	–
SessionBeanType	–
MessageDrivenBeanType	–
EjbInterfaceType	ComponentType.providedInterfaceTypes
EjbReferenceType	ComponentType.requiredInterfaceTypes
SimpleEnvironmentEntryType	ParamterType
EjbModule	Component
EnterpriseBean	–
SessionBean	–
MessageDrivenBean	–
EjbInterface	ProvidedInterface
EjbReference	RequiredInterface
EjbConnector	Connector
SimpleEnvironmentEntry	Parameter
MonitoredProperty	MonitoredProperty
EnterpriseBeanInstance	–
SessionBeanInstance	–
MessageDrivenBeanInstance	–
Call	–
ApplicationCall	–
BusinessCall	–
MessageCall	–
ThrownException	Failure
EnterpriseBeanStats	–
EjbMethodStats	PerformanceStats

the benefits of our approach in the following. We proposed the general idea in [30] and focused either on the bottom-up [28, 29] or top-down [22] synchronization of changes. Particularly, we evaluated our approach by developing three different types of reflection models that we synchronized with the EJB model and the adaptable software using either our model-driven solution or an alternative, code-based solution [28, 29]. On the one hand, the evaluation based on the three different types of reflection models focusing either on architectural constraint, performance, or failures indicates that the EJB model is sufficiently generic and comprehensive to address different concerns. On the other hand, the evaluation shows that our model-driven approach is efficient concerning development costs and runtime performance compared to alternative solutions. Moreover, our approach supports architectural adaptation, which required solving the challenges of refining adaptations when propagating changes from an abstract reflection model to the detailed EJB model, defining adaptation operators for the abstract reflection model (*i. e.*, what and how such a model can be adapted), and properly ordering individual steps of an adaptation when propagating the adaptation down to the adaptable software (*cf.* [22]). Thus, our framework

provides the following benefits for the causal connection with respect to the state of the art:

Development. The framework eases the development of the causal connection for user-defined, problem space-oriented reflection models supporting architectural monitoring and adaptation. Developers only have to provide declarative TGG rules specifying how these models are synchronized with the EJB model and thus how they are constructed. It usually does not require any coding from developers to realize the causal connection.

Typically, approaches using runtime models do not support developing the causal connection but already provide an implementation of it that is specific to a particular model and software platform. Hence, they prescribe the causal connection as well as the construction and content of the model (*e. g.*, [101, 307, 334, 335]). However, some approaches support such a development. *Rainbow* allows developers to implement mappings between the platform and the model to represent concerns of interest in the model [170]. Such mappings are supported for performance monitoring [173]. The support for adaptation and other concerns is not clear. Work by Song et al. supports the development by semi-automatically inferring metamodels for runtime models from sensor and effector APIs [396], providing a code generation for connecting a runtime model and an API [397], and using a bidirectional QVT-based transformation engine to synchronize such a model with another model providing a different view [395]. However, the abstraction level of all those models is close or even similar to the APIs. The models do not structurally abstract from the APIs such that they are rather related to the solution than to the problem space of the adaptable software. Thus, these models are similar to the EJB model provided by our framework. Finally, Colson et al. [119] use bidirectional programming to synchronize a platform-specific and a platform-independent runtime model to achieve reusability of analysis and planning algorithms. However, the platform-specific model is a configuration file of a web server, which is not causally connected to the running server as it captures configuration parameters that cannot be dynamically changed. Capturing only parameters, the causal connection does not support the synchronization of structural information such as the architecture of the adaptable software, which prevents architectural monitoring and adaptation.

Runtime Efficiency. The framework achieves a runtime-efficient causal connection by incrementally synchronizing the adaptable software, the EJB model, and the reflection model. The adapter and the model synchronization engine are driven by changes and propagate these changes incrementally to support monitoring and adaptation.

In contrast, the causal connections of other approaches work either partially or only for specific cases completely incrementally. While Morin et al. [306–308] update the runtime model incrementally by monitoring the software, the adaptation is only partially incremental. Since the adaptation is planned in a copy of the model, they use state-based MDE techniques, especially *EMF Compare*[9], to compare the original and the copy of the model to obtain a diff. A reconfiguration script is generated from the diff and executed to the software. The comparison is usually not incremental as it might require processing the whole models instead of individual changes of the models. Finally, the synchronization in [119, 395] seems to work incrementally for monitoring and adaptation. However, it considers either only parameters but not any structural information such as the architecture [119], or only models close or even at the same structural abstraction level as the software [395].

Abstract Reflection Models. The framework enables the use of reflection models that target problem spaces rather than solution spaces, which Blair et al. [79] consider as a

9 EMF Compare: https://www.eclipse.org/emf/compare/.

key characteristic of runtime models. A reflection model in our framework abstracts from the adaptable software platform and focuses on a specific concern from a problem space perspective. The abstraction gap between the higher-level reflection model and the low-level EJB model is mainly addressed by the TGG rules and the model synchronization engine. However, developers sometimes have to implement factories that extend the TGG rules and that provide missing knowledge to successfully bridge the abstraction gap.[10]

Related approaches usually employ structural runtime models. However, these models are often at the same abstraction level as the adaptable software. The model of Oreizy et al. [334, 335] is as complex as the software since it is a one-to-one mapping between model elements and implementation classes. Hierarchies of models such as the EJB and CompArch models in our framework are only considered theoretically [336]. Similarly, the models used by Song et al. [395] are at the same abstraction level as the sensor and effector APIs. In contrast, Colson et al. [119] consider abstract models in the sense that such models contain less data than lower-level models. However, all of these models capture only parameters and do not provide architectural abstractions. Abstract runtime models are also considered in [101, 170, 306], however, their abstraction levels are predefined to a specific architectural view, which cannot be customized by developers. The concerns attached to the architecture in the model can be customized [173] although they are typically predefined [101, 306].

Openness. The framework is *open* for user-defined reflection models. Instead of prescribing a metamodel and thus the abstraction level and content of the model, developers have the choice of reusing an existing (*e. g.*, CompArch) or developing a new metamodel based on their needs. This supports reuse of metamodels and eventually of software such as analysis and planning algorithms that are based on these metamodels.

Other approaches are generally not open as they prescribe the metamodels [101, 140, 213, 306, 335] or support only annotations to the model within the envelope of the metamodel [173]. The work by Colson et al. [119] and Song et al. [395, 397] is partially open since developers can use their own metamodel. However, the kind of supported metamodels seems to be restricted as the (meta)models used in both approaches provide no structural abstractions such as moving from a white- to a black-box view of the architecture.

Flexible Use. The framework supports a flexible use of the causal connection by loosely coupling the adaptable software, the EJB model, and the reflection model. Thus, feedback loops developed on top of the reflection model control the triggering of the causal connection on-demand when a set of changes should be synchronized. Such a flexible use is the state of that art and also applied by others (*e. g.*, [170, 306, 395, 397]).

Summing up, our framework supports maintaining abstract runtime models that reflect the adaptable software while the causal connection works incrementally and therefore runtime-efficient. Developers may customize the causal connection by providing a metamodel for the reflection model and by specifying the synchronization rules. However, they may also use the framework with the CompArch model and the corresponding TGG rules to conduct experiments for feedback loops that operate on top of the reflection model.

10 For the case of causally connecting the CompArch and the EJB models, one factory was required when synchronizing the creation of a Component in the CompArch model to the creation of a corresponding EjbModule in the EJB model. The factory determines the internal structure (*i. e.*, the constituting EnterpriseBeans) of the corresponding EjbModule based on the module's type specification, that is, the EjbModuleType in the EJB model. This internal structure is not captured in the CompArch model. Therefore, it cannot be synchronized to the EJB model and has to be obtained from a factory. We refer to [22] for a detailed discussion of such issues.

8.3 EUREMA

In this section, we discuss the implementation of EUREMA that comprises the topmost layer of the framework. We integrated EUREMA in the framework such that the feedback loops developed with EUREMA can use the causal connection between the reflection model and the adaptable software (see previous section). The EUREMA implementation itself does not depend on the framework such that EUREMA can be used without the framework in other contexts.[11] The implementation consists of the *EUREMA Language* and the *EUREMA Interpreter* (see Figure 70), each of which we discuss in the following.

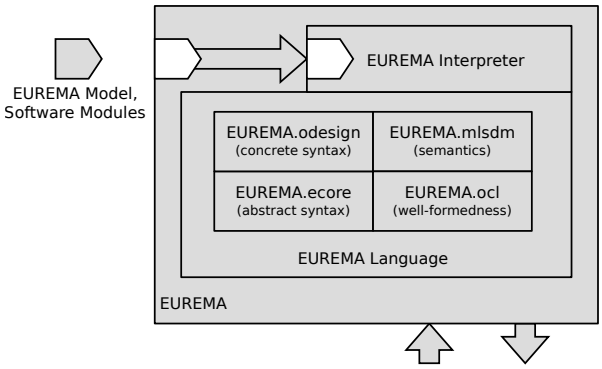


Figure 70: Implementation Architecture of EUREMA.

According to Mernik et al. [302], developing a DSL such as the EUREMA language is difficult because it requires domain knowledge and language engineering skills. However, the implementation can be eased by a *language development system* that generates tools from language specifications (*cf.* [212, 302]). Such a system is EMF that supports implementing modeling languages. Using EMF, we have developed the EUREMA language by implementing the four aspects of every modeling language: the abstract syntax, the well-formedness, a concrete syntax, and the semantics (*cf.* Section 2.1.1). These aspects are shown in Figure 70 by listing the corresponding artifacts and they are discussed in the following.

The abstract syntax of EUREMA is defined by the metamodel that we discuss in detail in Section A.1. The metamodel is expressed in *Ecore*, which is a partial implementation of the Meta Object Facility (MOF) [328]—the de-facto standard meta-metamodel in MDE—and provided by EMF. Therefore, the artifact implementing the metamodel is *EUREMA.ecore*. A diagram representing the concepts of this metamodel is shown in Section A.1.

The well-formedness of EUREMA is also discussed in Section A.1. It has been implemented by a set of constraints expressed in the Object Constraint Language (OCL). Therefore, we used *Eclipse OCL*[12] that implements the OCL specification [327] for EMF to express all of the constraints discussed in Section A.1 in the artifact *EUREMA.ocl*.

The concrete syntax of EUREMA refers to the notation of the language, that is, the Feedback Loop Diagram (FLD) and Layer Diagram (LD). Conceptually, we defined the concrete, graphical syntax by defining notational elements and by mapping the concepts of the EUREMA metamodel to these elements (*cf.* Section A.2). Technically, we developed a modeling editor for FLDs and LDs. The editor is based on *Sirius*[13] such that we implemented

11 As previously mentioned, EUREMA has been integrated into the *SimuLizar* framework [349].
12 Eclipse OCL: https://projects.eclipse.org/projects/modeling.mdt.ocl.
13 Sirius is a project that supports developing graphical modeling tools: http://www.eclipse.org/sirius/.

the concrete syntax in the artifact *EUREMA.odesign*. The resulting editor is integrated into the Eclipse workbench. Figure 71 shows a screenshot of the EUREMA editor.

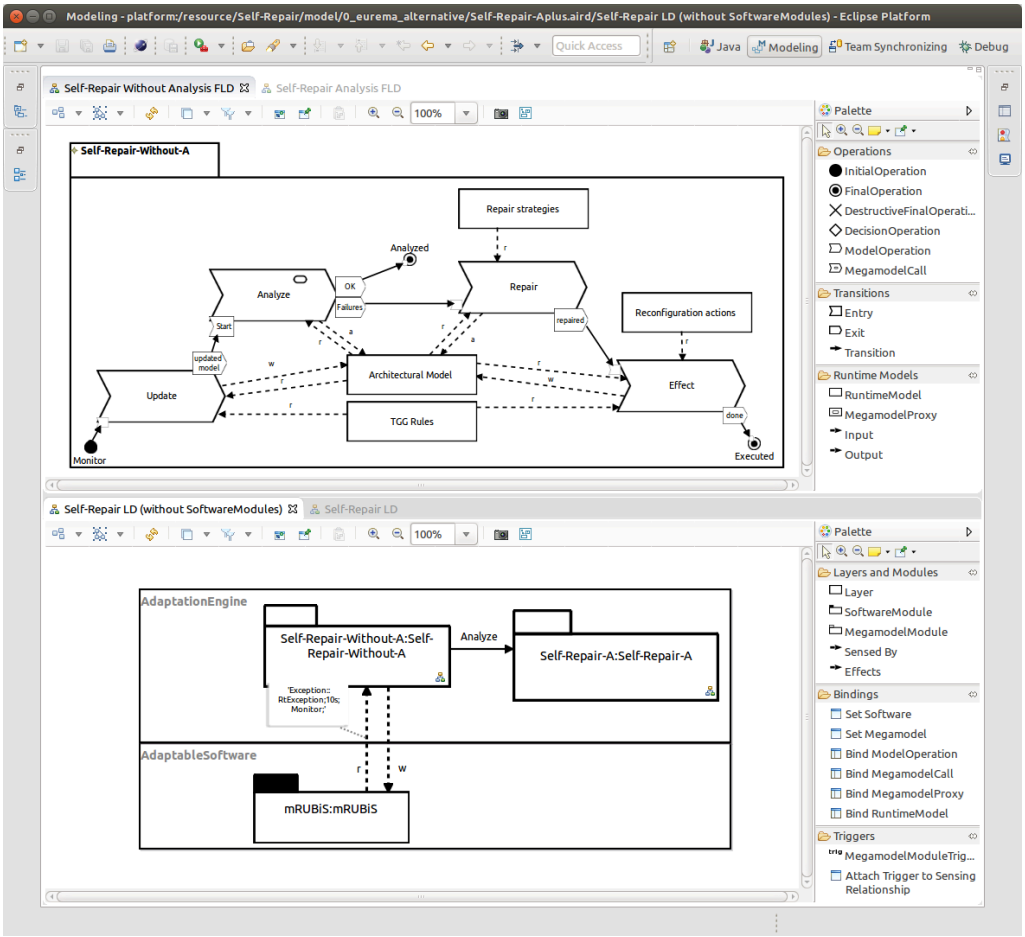


Figure 71: Screenshot of the EUREMA Editor.

The editor allows developers to create, edit, and comment FLDs and LDs as discussed in Chapter 5. The screenshot depicted in Figure 71 shows on top the FLD editor supporting the modeling of operations, transitions among operations, and the runtime models. It further shows at the bottom the LD editor supporting the layers and modules, the bindings of operations and models to implementations, as well as the triggers for modules. Moreover, the editor supports navigating between the diagrams, for instance, from the megamodel module in an LD to the FLD that describes the megamodel encapsulated in the module.

Moreover, the concrete syntax of the textual fragments of the EUREMA notation in terms of the triggering conditions for modules, which are modeled in the LD, and the conditions for branching the control flow in FLDs, are defined by grammars (*cf.* Sections A.4 and A.3). We implemented these grammars with the *Java Compiler Compiler (JavaCC)*[14] and embedded the resulting textual languages in EUREMA.

The EUREMA model created with the editor is executable. The execution of the model is determined by the semantics that we defined for the EUREMA language in an operational way. While we discussed the semantics informally in Chapter 6, we have additionally specified the semantics precisely and more formally such that a computer can consume the

14 Java Compiler Compiler (JavaCC): `https://javacc.java.net/`.

resulting specification. Particularly, we used Story Diagrams (SDs) to define the computations that are performed when executing the individual concepts of the EUREMA language. Therefore, we used the *Story-Driven Modeling (SDM) Tools*[15] that comprise an editor, interpreter, and debugger for SDs. We developed one SD, that is, the *EUREMA.mlsdm* artifact[16], that invokes other SDs while all of the SDs jointly specify the execution semantics. The execution semantics are discussed in detail Appendix B that shows the corresponding SDs.

Based on the EUREMA language with its four aspects (*i. e.*, the abstract syntax, well-formedness, concrete syntax, and semantics), we developed the Java-based *EUREMA Interpreter* (*cf.* Figure 70 on Page 162). The interpreter realizes the execution semantics of the EUREMA language such that it is able to execute an EUREMA model describing the self-adaptive software and especially the layered architecture modeled with the LD and the feedback loops modeled with the FLDs. Moreover, it integrates the parsers for triggering and branching conditions, which have been partially generated from the corresponding grammars by JavaCC. These parsers assists the interpreter in evaluating the triggering and branching conditions during the execution of the EUREMA model.

In short, the interpreter parses the EUREMA model to identify the megamodel modules (*i. e.*, the feedback loops) and their triggering conditions. This information is described in the LD. The interpreter evaluates these conditions based on time or incoming events (*cf.* time- and event-based triggers in Section 6.4.2). When such a condition of a megamodel module is satisfied, the interpreter executes the FLD instance encapsulated in the module to run the feedback loop. Therefore, it executes the operations and transitions of the FLD instance in a step-wise manner (*cf.* Section 6.4.3). Throughout the execution, the interpreter maintains runtime information about the execution (*cf.* Section 6.4.4) as well as the runtime models used within the feedback loops such that the models can be shared among different operations (*cf.* Section 6.4.5). In this context, the interpreter observes the runtime models to track their changes with events that can be accessed by event-based model operations (*cf.* Chapter 7) and it supports exporting snapshots of the EUREMA model for off-line inspection. Moreover, it addresses the interactions among modules (*cf.* Section 6.5), for instance, concerning concurrently executing feedback loops. Finally, it realizes a quiescence mechanism to enable safe adaptations of feedback loops (*cf.* Section 6.6).

To use the interpreter, developers use the interface shown in Listing 1. The interface declares two asynchronous `execute` methods for executing an EUREMA model depending on whether the model has already been loaded or not. The first method assumes that the model has been loaded and requires the root element of the model as a parameter. In contrast, the second method requires a Uniform Resource Identifier (URI) that the interpreter uses to load the model. Both variants return a queue. Sensors of the adaptable software add events to this queue, which are consumed by the interpreter and matched against event-based triggers of megamodel modules.[17] If a trigger is activated, the interpreter executes the corresponding module. An event may simultaneously activate more than one trigger such that the corresponding modules are executed concurrently (*cf.* Section 6.3).

Listing 1: Interface of the EUREMA Interpreter.

```
1 package de.mdelab.eurema.interpreter;
2
3 /** Interface to use the EUREMA interpreter. */
```

15 Story-Driven Modeling (SDM) Tools: http://www.mdelab.de.

16 The *EUREMA.mlsdm* story diagram is shown in Figure 101 on Page 331.

17 For instance, the adaptable software platform in our framework (*cf.* Section 8.1) emits events that notify about changes in the software. Such events are the basis for the events to be added to the queue.

```java
public interface EuremaInterpreter {

  /**
   * Executes the EUREMA model.
   * @param architecture
   *         The root element of the EUREMA model.
   * @return The queue to which sensor events from the adaptable software are added. */
  public EventQueue execute(eurema.Architecture architecture);

  /**
   * Executes the EUREMA model.
   * @param euremaModelUri
   *         the URI of the EUREMA model whose root element is an eurema.Architecture.
   * @return The queue to which sensor events from the adaptable software are added. */
  public EventQueue execute(String euremaModelUri);
}
```

The EUREMA model as the parameter of the `execute` methods is one input to the interpreter. The other input is a set of *Software Modules* (see Figure 70 on Page 162) that are the implementations of the model operations (*cf.* Chapter 7 discussing the solution space of such implementations). These software modules in terms of Java classes must be part of the interpreter's classpath such that the interpreter can load them and trigger their execution on demand when executing the model operations as part of the feedback loops. Such a module has to realize the interface shown in Listing 2 such that the interpreter is able to trigger the module. This interface declares the `run` method with two parameters. First, a list of models that are defined as the input of the model operation in the FLD. Each of these `IModels` provide access to the name, the content, and the change events of the corresponding runtime model. Second, the entry name refers to the entry point of the model operation that has been taken for this execution of the operation. The method returns a result that comprises a list of models that are defined as the output of the model operation in the FLD as well as a status that refers by name to the exit point of the model operation. The interpreter uses this status to identify the exit point that should be used for continuing the execution of the FLD instance. Thus, a software module must implement the `run` method, which is invoked by the interpreter when executing the corresponding model operation.

Listing 2: Interface of a Software Module Implementing a Model Operation.

```java
package de.mdelab.eurema.operation;
import java.util.List;
import de.mdelab.eurema.model.IModel;

/**
 * Interface to be realized by implementations of model operations specified in Feedback
 * Loop Diagrams (FLDs). The implementation must provide a public default constructor.
 */
public interface IModelOperation {

  /**
   * Method that implements a model operation declared in an FLD.
   *
   * @param models
   *         The input models of the model operation as specified in the FLD.
   * @param entryName
   *         The name of the entry taken for executing this model operation.
   * @return The result of executing the implementation of this ModelOperation. */
```

```
19    public IResult run(List<IModel> models, String entryName);
20  }
```

In this context, the runtime models that are the input or output of a model operation and of the corresponding software module are defined and developed completely by the user of EUREMA. Such runtime models are reflection models that are causally connected to the adaptable software. Moreover, if the operations are model-driven, further runtime models exist that specify and execute the operations (*cf.* Section 7.1). The interpreter only relies on EMF such that the models must be defined and developed with EMF. Thus, the interpreter can handle any EMF-based model that is expressed with any metamodel, for instance, to dynamically load, maintain, and pass a model from one operation to another. Although the interpreter relies on EMF, it can run standalone and decoupled from the Eclipse workbench to embed it into the runtime environment of the self-adaptive software.

The EUREMA implementation based on the framework serves as a basis for the evaluation of EUREMA, which will be discussed in the following chapter. Beforehand, we briefly discuss a simulator we developed additionally to the framework to ease the evaluation of EUREMA and particularly the development and testing of EUREMA feedback loops.

8.4 SIMULATOR

In addition to the framework discussed in the previous sections, we developed a simulator that emulates parts of the framework. The simulator therefore makes the framework more light-weight, which supports the early development and testing of EUREMA feedback loops as well as experiments with EUREMA, for instance, to explore alternative designs of the feedback loops. Figure 72 contrasts the implementation architectures of the framework without and with using the simulator.

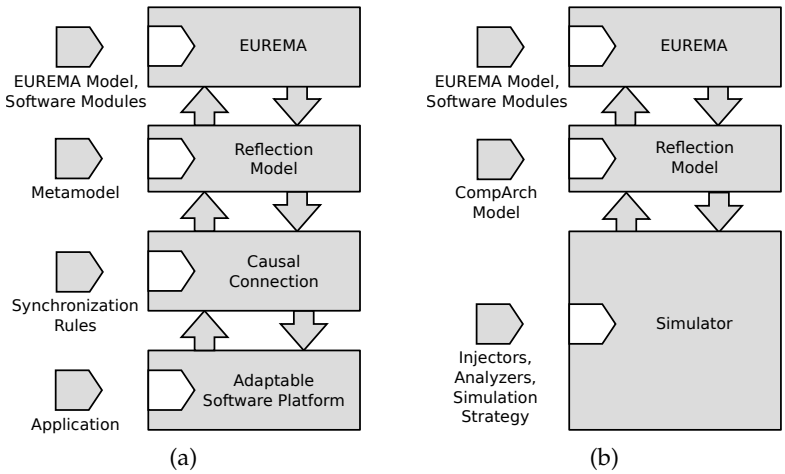


Figure 72: Implementation Architecture of the Framework (a) without and (b) with the Simulator.

As illustrated in Figure 72, the *Simulator* emulates the adaptable software platform (*cf.* Section 8.1) and the causal connection (*cf.* Section 8.2) such that it maintains the reflection model. Consequently, developers can use the simulator to run their EUREMA feedback loops on top of the reflection model without having to operate the adaptable software

platform, including the application, and the causal connection. On the one hand, this eases the early development of EUREMA feedback loops as it allows developers to focus on the adaptation logic instead of being concerned with the setup, configuration, and operation of the adaptable software platform and causal connection. On the other hand, relying only on the reflection model enables early testing of feedback loops while the adaptable software with its sensors and effectors does not have to be available (*cf.* [14]). Additionally, the simulator might reduce the turnaround times of testing since in our case the initialization of the simulator is less time-consuming than of the GlassFish server (including the required database server and mKernel plugin), adapter, and model synchronization engine. The simulator is a standalone Java application that can be used as a basic Java library, which makes it a light-weight solution for testing opposed to the EJB server infrastructure.

To use the simulator, developers have to provide *Injectors*, *Analyzers*, and a *Simulation Strategy* (see Figure 72b). An injector implements the injection of an issue into the reflection model. For instance, an injector can introduce a failure into the model for a self-healing scenario. Consequently, an injector simulates the occurrence of an issue such as a failure in the adaptable software and the synchronization of this issue from the software to the model. An analyzer implements an analysis of the reflection model, for instance, to check whether the injected issues have been handled, the model is valid, or the model corresponds to a given reference model. Besides the injectors and analyzers, developers specify a *Simulation Strategy* that defines for each simulation run (*i. e.*, for each run of a feedback loop) the number of injections as well as for each injection of the run the issue to be injected (*i. e.*, the injector to be used) and the location in the model as the target of the injection. Thus, such a strategy determines the self-adaptation scenario for testing the feedback loop.

Based on this information, one simulation run consists of three steps:(1) The monitoring activity of a feedback loop triggers the simulator to inject issues to the reflection model. For the given simulation run, the simulator consults the simulation strategy provided by the developer to obtain the number of injections as well as the issue and target for each injection. Given these issues and targets for all injections, it executes the corresponding injectors. (2) The feedback loop performs its adaptation activities to identify and handle the injected issues. In this context, issues are handled by adapting the reflection model. In a Models@run.time approach, an adaptation is prescribed and previewed in the reflection model before it is executed to the adaptable software through a causal connection. We follow the same approach with the simulator that emulates the adaptable software and causal connection such that the feedback loops operate only on the reflection model regardless whether the actual adaptable software with the causal connection or the simulator is used. Therefore, the same feedback loops that eventually should manage the adaptable software can be tested early with the simulator (*cf.* [14]). (3) Having adapted the model, the execute activity of the feedback loop triggers the simulator to analyze the model as its last step. The simulator then runs the analyzers and provides feedback to the developers by logging the results of the analyses. Consequently, one simulation run corresponds to testing a single execution of the feedback loop while the simulator further supports multiple runs—as determined by the user-defined simulation strategy—to test consecutive executions.

While the framework without the simulator (see Figure 72a) is open for any metamodel to express the reflection model, the simulator is tailored to reflection models that are instances of the CompArch metamodel (see Section 8.2). The tailoring is the prerequisite that the simulator can orchestrate the injectors and analyzers and in particular provide default analyzers (*e. g.*, to check whether the architectural reflection model contains any components whose required interfaces are not connected to other components). As the CompArch

metamodel supports describing the runtime architecture of the adaptable software, the simulator addresses the testing of feedback loops performing architectural adaptation.

Since there is no adaptable software running, the simulator cannot construct the reflection model, in contrast to the causal connection discussed in Section 8.2 that produces the reflection model based on observations of the adaptable software. Therefore, developers have to provide a *CompArch Model* (see Figure 72b) describing the initial architecture of the adaptable software and in particular of the application that will eventually be provided to the adaptable software platform when using the original framework. Consequently, the simulator is not restricted to a specific application but addresses any application that can be described with the CompArch metamodel. For our experiments based on the mRUBiS application, we created and provide a corresponding CompArch model that describes the runtime architecture of mRUBiS.

Summing up, the simulator supports the early development and testing of EUREMA feedback loops by disburdening developers from operating the adaptable software platform and developing and operating the causal connection. However, to use the simulator, developers have to provide injectors, analyzers, at least one simulation strategy, and a CompArch model describing the application under adaptation.

8.5 SUMMARY

In this chapter, we discussed the implementation of EUREMA and in particular of its language (*i. e.*, the abstract and concrete syntax, well-formedness, and semantics), editor, and interpreter. Moreover, we discussed the technical framework that we use to experiment with EUREMA. The EUREMA implementation is loosely coupled to the framework and can therefore be easily used in other technical contexts. For instance, parts of the framework can be replaced by a simulator to support the early development and testing of EUREMA feedback loops. The overall implementation is generic and therefore open for any EJB-based application to be managed and for any architectural self-adaptation capability to be realized. Consequently, developers may customize the framework to their needs while we provide a default customization comprising the mRUBiS application, the CompArch metamodel, and the synchronization rules for this metamodel.

The EUREMA and framework implementation is the basis for evaluating EUREMA. We will present the evaluation of EUREMA in the following chapter. To evaluate specific aspects, we individually use the EUREMA language and interpreter with or without the framework. Depending on the aspect, we use the framework either with the running adaptable software (*i. e.*, the EJB server extended with the mKernel plugin and hosting mRUBiS) and causal connection (*i. e.*, the adapter, the EJB model, and the model synchronization engine) or with the simulator that emulates the adaptable software and causal connection.

9 EVALUATION

In this chapter, we evaluate EUREMA qualitatively and quantitatively. First, we assess the language design from a language engineering point of view in Section 9.1. Then, we investigate the expressiveness of the language by applying it to third-party approaches in Section 9.2. Afterwards, we detail our mRUBiS application example and report on experiments that show the effective application of EUREMA to self-adaptation problems (see Section 9.3). Using the same example, we conduct a comparative study to assess the development costs and runtime performance of different solutions that are either developed with or without EUREMA (see Section 9.4). Moreover, we assess the EUREMA prototype by discussing quality attributes that target development frameworks for self-adaptive software (see Section 9.5). Based on the evaluation, we discuss in Section 9.6 EUREMA's coverage of the requirements for engineering self-adaptive software that we introduced in Chapter 3. Finally, we summarize the whole evaluation of EUREMA in Section 9.7.

9.1 LANGUAGE DESIGN

In this section, we take a language engineering perspective and assess the design of the EUREMA language by contrasting it with 26 design guidelines proposed by Karsai et al. [241]. They argue that following these guidelines supports achieving a "better quality of the language design and a better acceptance among its users" [241, p. 7]. Therefore, we first discuss the design rationale of the EUREMA language in Section 9.1.1 and do the assessment in Section 9.1.2. Finally, we summarize the assessment in Section 9.1.3.

9.1.1 *Design Rationale*

With EUREMA, we have created a novel modeling language to address the need of making feedback loops explicit in the architectural design of self-adaptive software [93, 311, 386]. This need calls for a language to express feedback loops at the abstraction level of software architectures and thus, for a modeling language above the abstraction level of programming languages. For this purpose, an existing general purpose modeling language such as the Unified Modeling Language (UML) [329] is not helpful because of three reasons.

First, Shaw [386] argues that typical software engineering paradigms such as object orientation are not the best option for architectures of software systems with feedback loops, which calls for novel means to design such systems. Particularly, she proposes following a control paradigm that clearly separates the controller and the controlled process, that considers the data flow *loop* between these two elements, and that addresses the specialty that the former monitors and effects the latter element. This paradigm contrasts with typical data flow architectures such as pipes and filters that treat the individual elements equally and structure them sequentially. In the control paradigm, however, there is an asymmetry between the controller and the process and both are structured in a loop. Such a control paradigm for software architectures may provide a useful abstraction for self-adaptation. Müller et al. [310] argue similarly and stress the need for feedback control and architecture perspectives. This excludes the use of UML that follows an object-oriented paradigm.

Second, we want to clearly separate the adaptation logic (*i. e.*, the adaptation engine and feedback loops) from the domain logic (*i. e.*, adaptable software). This promotes separation of concerns and promises more maintainable and reusable designs (*cf.* external approach in Section 2.2.3 and [370]). Likewise, Shaw [386] argues that such a separation allows us to distinguish issues about the desired functionality and the adaptation, to explicitly select and design a control strategy, and to make the current and desired model of the controlled subsystem explicit and hence, amenable for analysis and validation. To keep the separation strict, we aim for a different language for the adaptation engine than for the adaptable software. The latter may follow an object-oriented paradigm and be modeled with UML.

Third, using or extending UML is cumbersome due to the size and complexity of the language [71, 163, 164] and ambiguity of the semantics [163, 164, 207, 216]. Though it is possible to define a UML profile (*i. e.*, a customized and extended version of UML) using a subset of UML [383], this approach still requires understanding the whole UML to identify the relevant subset. Moreover, the profile mechanism is restricted to adding properties and relationships to UML while neglecting elements and a way to precisely define the semantics of such extensions [164]. The profile mechanism is further restricted since the extensions have to "fall within the syntactic and semantic envelope defined by standard UML" [383, p. 308]. Therefore, integrating another paradigm such as control to the object-oriented UML will likely reach its limits and require from the user to cope with a highly complex language. Moreover, having the idea of runtime models in mind, we aim for a lightweight solution that contradicts the size and complexity of the UML (metamodel).

Consequently, we do not want to use UML and since there does not exist an established notation and best practices to describe feedback loops for self-adaptive software, we have decided to create with EUREMA a novel Domain-Specific Language (DSL). This DSL tackles the three issues discussed previously.

First, EUREMA follows a control paradigm that explicitly addresses the asymmetry between the adaptation engine and adaptable software as well as the feedback loop between both of them. Adopting such a paradigm makes the feedback loops explicit in the architectural design (see FLDs and LDs in Chapter 5). Moreover, it allows engineers to directly use established concepts of the self-adaptive software domain such as MAPE-K and runtime models to describe feedback loops without having to translate such (mental) concepts to other, less suitable paradigms such as object orientation realized in existing languages.

Second, following a control paradigm clearly separates the adaptation engine and the feedback loops from the adaptable software while considering the data flow among them (*cf.* sense and effect relationships in the LDs) as well as within a feedback loop (*cf.* runtime models shared by operations in the FLD). The feedback loops are specified with EUREMA (*cf.* FLDs) while the adaptable software can be modeled with any other language, which promotes separation of concerns. Thus, engineers can focus on the self-adaptation problem and adopt a control perspective when using EUREMA to design a self-adaptive software.

Third, we propose a *lean* language that covers only the essential concepts to describe feedback loops (*cf.* Chapter 5 and Appendix A) and that additionally eased the execution semantics as realized by a lightweight interpreter (*cf.* Chapter 6 and Appendix B).

This motivation for a DSL is complemented by the idea of a *runtime megamodel* that has become the underlying principle of the EUREMA language. In Section 4.2, we defined a runtime megamodel as a runtime model that specifies the feedback loops in a self-adaptive software system by describing the elements of these feedback loops, which are runtime models and adaptation activities, as well as the interplay between these elements, and that enables direct execution of the model to change an adaptable software at runtime, that is,

to perform self-adaptation. While we generally sketched this idea in Chapter 4, we discuss in the following how it has become manifest in the language design.

Based on this definition, the FLD *is* a runtime megamodel. It describes the runtime models and the adaptation activities in terms of model operations as first class entities while the interplay is substantiated to the control flow among the operations and to the usage of the models by the operations. Such a description specifies a feedback loop. Moreover, an FLD is kept alive at runtime, which makes it a runtime model, to directly execute the specified feedback loop. An FLD as a runtime model is encapsulated in a *megamodel module* that is named after the fact that an FLD is a (runtime) megamodel. Consequently, an FLD as a runtime megamodel leverages the benefits of making the elements of a feedback loop explicit and supporting the execution, which have been conceptually discussed in Section 4.2.

Technically, an FLD makes the runtime models and model operations explicit during design and run time, which makes these elements amenable for analysis and adaptation. When creating FLDs (*cf.* Sections 5.1 and 5.2), engineers design feedback loops by specifying these elements and their relationships. EUREMA allows engineers to freely express the design, which promotes explicitly reasoning, deciding, and adapting the design. For instance, engineers have to make explicit choices of how the adaptable software should be reflected by a runtime model and which adaptation strategy should be used. Consequently, an explicit design requires from engineers to think about the self-adaptation problem and express proper solutions, which should eventually result in a well-engineered self-adaptation mechanism [386]. Keeping the FLD alive at runtime, the design can be further dynamically adapted since the described feedback loop with its runtime models and operations remain explicit. The flexible nature of runtime models in contrast to hard-coded or generated mechanisms for self-adaptation supports the runtime changeability of the feedback loop (designs). This has been exploited by layered feedback loops and off-line adaptation (*cf.* Sections 5.3 and 5.4). Thus, the maintenance of model operations and runtime models continues at runtime and thus beyond the initial development.

The execution support of a runtime megamodel is realized by directly executing instances of FLDs. Besides structuring the elements of a feedback loop, an FLD serves as a behavioral specification of the feedback loop that is sufficiently detailed for execution. Consequently, the same formalism is used for specifying and executing feedback loops, which avoids any translation of FLDs to an executable formalism and supports engineers in retracing the execution. Since the execution happens at the level of the same domain concepts used for the design, FLDs as design artifacts can be directly simulated for early validation (*cf.* Section 8.4) and FLDs as runtime artifacts can be seamlessly transferred from the runtime to the development environment for manual inspection (*cf.* Section 5.4).

These benefits result from having a runtime megamodel as the *underlying* principle of EUREMA. Consequently, this principle is visible in the abstract syntax.[1] However, it is not visible in the concrete syntax such as the FLD notation since the idea of a runtime megamodel is not related to the problem domain of self-adaptive software. While the whole FLD corresponds to a megamodel, engineers only use problem-level abstractions to fill the FLD with model operations and runtime models to specify a feedback loop.

Consequently, the notation we designed for FLDs provides elements that are focused on the problem domain. Since there does not exists an established notation for the domain of self-adaptive software, we created a novel one that is specific for this problem domain. This notation is also limited to the domain such that engineers concentrate on the essence,

1 The EUREMA metamodel defines the language concept of a Megamodel that corresponds to an FLD and that contains the elements of a feedback loops such as model operations or runtime models (*cf.* Appendix A).

that is, specifying feedback loops. Therefore, the FLD notation is different from existing notations such as UML to avoid any intertwining of the adaptation engine and the adaptable software. Engineers may use UML to specify and develop the adaptable software that should be kept separate from the adaptation engine, for which EUREMA is used. Nevertheless, FLDs share some ideas with flowcharts and data flow diagrams such as *UML Activities*. FLDs and UML Activities are similar with respect to modeling flows of actions (in UML) or operations (in EUREMA) such that engineers who are familiar with UML can more easily grasp FLDs to specify feedback loops. This balances the burden for engineers of learning a new language and the need for a novel DSL to specify feedback loops.

The same holds for LDs that share ideas with UML, especially *UML Packages* and *UML Objects*, such as the description of instances and their relationships to each other. However, the LD focuses on the domain of self-adaptive software and it is therefore specialized on describing instances of feedback loops and of the adaptable software as well as their sense, effect, and use relationships to each other. The motivation for this type of diagram is to complement the behavioral FLDs with an architectural instance view that structures the feedback loops in a layered architecture and connects them to the adaptable software.

Considering a runtime megamodel in a broader sense as a runtime model that captures other runtime models and their relationships to each other (*cf.* Section 4.2), the LD *is* a runtime megamodel. FLD models specifying feedback loops are kept alive at runtime and encapsulated into modules for execution. An LD model is also kept alive at runtime and it captures such modules and their sense, effect, and use relationships to each other, which makes an LD a runtime megamodel in the broader sense. Thus, an LD enables similar benefits as runtime megamodels. Keeping the feedback loop specifications in terms of the FLD models alive and explicit in a runtime LD enables the runtime evolution of the feedback loops. Examples of evolution such as adding, removing, or patching a running feedback loop are discussed in the context of off-line adaptation in Section 5.4.

Thus, the idea of a runtime megamodel is the underlying principle of the EUREMA language although it is not made explicit in the FLD and LD notation. Consequently, engineers are not concerned with runtime megamodels but they still get the related benefits.

In this thesis, we introduced the EUREMA language rather informally in Chapters 5 and 6. We used natural language and EUREMA models to describe and illustrate the syntax and semantics of the language. However, we complement this informal description with a formal design of the language (*cf.* [302]) that covers the abstract syntax, well-formedness, and execution semantics. The abstract syntax and well-formedness are defined by a metamodel and a set of constraints (*cf.* Appendix A). The execution semantics is specified in an operational way by graph transformations (*cf.* Appendix B). Concerning the concrete syntax of graphical languages, there does not exist a standard method for its specification (*cf.* Section 2.1.1). Thus, we conceptually defined the concrete syntax by notational elements that are mapped to the concepts of the abstract syntax (*cf.* Section A.2). This concrete syntax is implemented in the prototype editor for EUREMA (*cf.* Section 8.3). Summing up, we address the abstract syntax, well-formedness, semantics, and concrete syntax for the design of EUREMA, which are the four constituent parts of any modeling language (*cf.* Section 2.1.1). With respect to these four parts, we may consider the design of EUREMA as complete.

9.1.2 *Assessing the Language Design*

Based on the design rationale discussed in the previous section, we now assess the design of the EUREMA modeling language. In this context, Selic [383, p. 290] argues that "[t]he

design of modeling languages is still much more of an art than a science" particularly as there does not exists a "systematic consolidated body of knowledge" of designing a modeling language. Consequently, it is also difficult to evaluate such a design. However, initial attempts to define design guidelines or patterns for DSLs have been made [241, 399, 441]. We use the most recent attempt by Karsai et al. [241]. They propose a comprehensive list of 26 design guidelines for domain-specific modeling languages that is based on their experience in this field and on existing guidelines for general purpose and modeling languages. According to the authors, following these guidelines supports achieving a "better quality of the language design and a better acceptance among its users" [241, p. 7].

In the following, we use these guidelines to assess the design of the EUREMA language. Karsai et al. [241] group the guidelines by considering the purpose, realization, and content of a language as well as the concrete and abstract syntax. We briefly describe each group with its guidelines and contrast each guideline with the design of the EUREMA language.

Language Purpose. This group consists of three guidelines that are concerned with the goal of the language, that is, what is the language used for.

1. *Identify language uses early*. The first guideline recommends early determination of the intended use of the language as it influences the language concepts. For instance, using a language for documentation requires less concrete concepts than for code generation.

Right from the start we aimed for a language to specify and execute feedback loops in self-adaptive software [31, 32], which has resulted in the EUREMA language [21, 25] as discussed in this thesis. Consequently, we designed the EUREMA language to provide the required concepts to specify feedback loops while being executable by an interpreter.

2. *Ask questions*. The aim of this guideline is to identify the domain of the language, roles that use the language, and how the language is integrated into the development process.

We analyzed the research field of self-adaptive software to identify and scope the domain (*cf.* requirements discussed in Chapter 3). We primarily consider software engineers who develop and maintain self-adaptive software as the roles for users of EUREMA. Consequently, EUREMA is embedded into a development process by supporting the design/specification (*cf.* Sections 5.1–5.3) and the post-deployment maintenance (*cf.* Section 5.4) of self-adaptive software as well as the related implementations (*cf.* Chapters 7 and 8). However, we do not constrain the development process such that we only consider the major phases of initial development and maintenance during which EUREMA is used. In a concrete setting, the development process needs to be refined for the concrete setting (*cf.* [1]).

3. *Make your language consistent*. This guideline demands that each concept of the language should contribute to the purpose of the language such that there are no unnecessary concepts with respect to the domain and usage of the language.

For EUREMA, each language concept contributes to the purpose we intend for EUREMA. Moreover, each concept is required to completely cover the domain of self-adaptive software that we identified as requirements in Chapter 3. In Section 9.6, we will discuss in detail which language concepts cover which requirements and show that each concepts is actually required. Consequently, there does not exist any unnecessary concept or feature in the language that we could have omitted without reducing the coverage of the domain.

Language Realization. This group of guidelines addresses the implementation of a DSL.

4. *Decide carefully whether to use graphical or textual realization*. Karsai et al. [241, p. 9] suggest considering the advantages and disadvantages of graphical (*e. g.*, "graphical models

provide a better overview and ease the understanding of models") and textual (*e. g.*, faster development, platform and tool independence) approaches to make an informed decision.

We decided for a graphical realization of EUREMA to make it easier for engineers to grasp the feedback loop. We think that a feedback loop as a cycle can be easier identified and understood in a graphical than in a textual representation. A graphical model can directly visualize the cycle, which cannot be done with a linear, textual model.

5. *Compose existing languages where possible*. This guideline advocates the reuse and composition of existing languages to realize the DSL. Its goal is to reduce implementation efforts, which is only feasible if the reused languages and their paradigms match with each other.

As discussed in Section 9.1.1, the reuse of an existing general purpose language such as UML is not an option. Moreover, there does not exist any established language for feedback loops in self-adaptive software that could have been reused. Consequently, reuse was not possible and we developed with EUREMA a completely novel language for this purpose. Moreover, the EUREMA language itself integrates other languages that are used to express the runtime models used within a feedback loop (*e. g.*, OCL, SD/SP, TGG and CompArch). Such an integrating language for self-adaptive software did not exist before.

6. *Reuse existing language definitions*. If existing languages cannot be reused right away (see previous guideline), existing definitions of languages should be reused. Karsai et al. [241, p. 9] argue that reusing "the definition of a language as a starter to develop a new one is better than creating a language from scratch"—assuming that the reused language definition has a "good" design, which enables the reuse of best practices.

As discussed in Section 9.1.1, the EUREMA language shares ideas with UML, particularly, FLDs with *UML Activities* and LDs with *UML Packages* and *UML Objects*. In this context, we investigated the UML metamodel and reused ways of structuring language concepts. For instance, the EUREMA metamodel (see Appendix A) structures the different types of operations, that are used in FLDs, in a similar way as the UML metamodel does it for actions used in *UML Activity* diagrams. Thus, we have not directly reused the definition of UML but only patterns or practices of how to structure language concepts in a metamodel.

7. *Reuse existing type systems*. The last guideline for the DSL implementation encourages the reuse of a type system to avoid the costly and error-prone development of such a system.

On the one hand and at the level of EUREMA models, the EUREMA language adopts a dynamic typing approach in contrast to a static and explicit type system to cope with the required flexibility in self-adaptive software. On the other hand and concerning the implementation of the EUREMA language, we reuse the type system provided by EMF which is the underlying technology of the EUREMA implementation (*cf.* Chapter 8).

Language Content. This group of guidelines focuses on the elements of the language and on deciding which elements should be part of the language and which should not.

8. *Reflect only the necessary domain concepts*. The DSL should generally provide elements only for the necessary part of the domain it is addressing. This ensures the focus of the DSL on the targeted domain and the expressiveness to capture the relevant domain concepts.

We identified and scoped the domain of self-adaptive software with requirements (*cf.* Chapter 3) and developed the EUREMA language to reflect only concepts of this scoped domain (*cf.* discussion in Section 9.6). Other concepts are not reflected in the EUREMA language

to keep the focus on self-adaptive software. The resulting expressiveness has been demonstrated by modeling various examples of self-adaptive software such as single (*cf.* Section 5.1), multiple (*cf.* Section 5.2), and layered (*cf.* Section 5.3) feedback loops as well as third-party approaches to self-adaptive software (*cf.* Section 9.2). As argued by Karsai et al. [241], such examples can be used to validate the language definition against the domain.

9. *Keep it simple*. This guideline defines simplicity as an important aspect of a DSL as it eases the implementation, introduction, learning, understanding, and use of the language. Simplicity can be achieved by following the Guidelines 10, 11, and 12 [241].

We think that we achieved a rather simple language with EUREMA by avoiding unnecessary generality (Guideline 10), limiting the number of language elements (Guideline 11), and avoiding conceptual redundancy (Guideline 12) as discussed in the following.

10. *Avoid unnecessary generality*. This guideline recommends avoiding any generalization of the DSL to cover more than the necessary domain concepts. This ensures the focus of the DSL on the domain and does not complicate the language without contributing to the purpose of the DSL. Thus, this guideline is similar to Guideline 8.

We do not introduce any generalization into EUREMA to cover more than the intended domain (*cf.* Section 9.6). In contrast, we conceive a rather stable language that covers the core concepts for specifying feedback loops: operations, runtime models, model usage, and control flow. Still, these concepts can be refined with stereotypes to emphasize their role in a feedback loop without changing their execution semantics (*cf.* Section 5.1.2). For instance, an operation can be stereotyped with ≪Monitor≫, ≪Analyze≫, ≪Plan≫, and ≪Execute≫ to highlight its task in a MAPE-K cycle. In this context, stereotypes for other blueprints of feedback loops such as sense-plan-act or learn-reason-act can be used (*cf.* Section 2.2.3).

11. *Limit the number of language elements*. The number of language elements should be limited to the target domain. This should avoid unnecessary complexity and therefore ease the understanding and use of the DSL.

In general, the EUREMA language does not have that many elements considering its concrete (*cf.* Figure 16 on Page 58 and Figure 18 on Page 61) and abstract syntax (*cf.* Figure 99 on Page 306). The number of elements is further limited by providing two diagram types. A subset of the language is used either in FLDs or LDs. Finally, we limit the number by providing just the core concepts for feedback loops, which can be specialized with stereotypes (*cf.* previous guideline). Such a specialization is optional and not reflected in the language such that it does not increase the number of elements. For instance, regardless of the feedback loop blueprint such as MAPE-K, sense-plan-act, or learn-reason-act (*cf.* Section 2.2.3), the same elements—but stereotyped—are used to specify the feedback loop.

12. *Avoid conceptual redundancy*. A DSL should avoid redundancy of language concepts, that is, it should not provide multiple concepts to describe the same aspect of a domain.

In EUREMA, we did not add any redundant concepts. To describe certain aspects of the domain (*e. g.*, a feedback loop activity), unique language concepts (*e. g.*, a model operation) are used. Engineers have no choice of concepts to describe the same fact of the domain.

13. *Avoid inefficient language elements*. This guideline aims for the efficient execution of the software developed with the DSL. Any element that causes inefficiency should be avoided in the language design such that the efficiency of the execution solely depends on the model created with the DSL and not on the specific language elements used in the model.

In EUREMA, engineers have no choice among different language elements to describe the same fact. Therefore, the problem of choosing less efficient elements does not appear. Moreover, the EUREMA language itself does not contain any inefficient elements as indicated by the evaluation of the runtime performance and overhead of the EUREMA interpreter (*cf.* Section 9.5.5). Consequently, the efficiency only depends on the concrete model created by the engineer and not on the language—as demanded by the guideline.

Concrete Syntax. This group of guidelines addresses the design and definition of the readable representation, that is, the notation of the DSL.

14. *Adopt existing notations domain experts use*. If a notation is established in a domain, it should be used such that users do not have to learn a novel one. If there is no established notation, the new notation should be "as close as possible to other existing notations within the domain or to other common used languages" [241, p. 10] to reduce learning efforts.

There does not exist an established notation to describe feedback loops in self-adaptive software such that we created a novel one. This novel notation shares characteristics with existing notations of the UML. The flow of operations in FLDs is similar to the flow of actions in *UML Activity* diagrams, and the structuring of modules in LDs is similar to the structuring of objects or packages in *UML Object* or *Package* diagrams. Thus, users who are familiar with UML know about these general characteristics that also apply to EUREMA. However, the EUREMA notation is still sufficiently different from the UML notation to clearly separate the adaptation engine from the adaptable software (*cf.* Section 9.1.1).

15. *Use descriptive notations*. This guidelines demands the use of descriptive notations that ease the learning and understanding of the DSL and the models created with the DSL.

We think that we use descriptive notations in EUREMA. Particularly, the model operations in FLDs are visualized by block arrows that typically indicate activities while the runtime models are depicted by rectangles/boxes that often describe structural elements such as artifacts. The control flow is visualized by solid arrows connecting operations such that the whole flow of activities can be easily understood by following the block and the solid arrows. In LDs, the modules are visualized by packages that indicate an encapsulation, that is, there exists a further specification (*i. e.*, an FLD or code) encapsulated in the module.

16. *Make elements distinguishable*. To ease the reading of a model, the notational elements should be distinguishable. In other words, each language element should have its own graphical representation in the notation.

In EUREMA, each language element has a different graphical representation to avoid any confusion of the user (*cf.* Section A.2). However, similarities within the notation exist. First, an FLD is framed with a package while an instance of such an FLD (*i. e.*, a megamodel module) is represented as a package in the LD. Second, the control flow among model operations in an FLD is represented by a solid arrow that is also used in the LD, although with an additional label, for instance, to bind a complex model operation to an FLD instance to be invoked. Such a binding defines that the control flow is passed from the complex model operation to the invoked FLD instance during execution. Thus, the solid arrow in the LD is also concerned with the control flow. Third, the use or access of runtime models by operations (*e. g.*, to read or write/change a model) is visualized by dotted arrows in the FLDs. Such dotted arrows are also used in LDs for representing sensing and effecting relationships among modules. Such relationships requires access to sensed or effected modules in terms of reading or writing/changing them. Consequently, similarities between the FLDs and LDs are considered in the graphical notation. This additionally addresses Guideline 21.

17. *Use syntactic sugar appropriately*. Syntactic sugar concerns language elements that do not enhance the expressiveness of the language but that can support the readability of the models. However, too much of syntactic sugar could make it difficult for the user to see the essence of a model. Therefore, syntactic sugar should be used wisely.

Currently, we do not use any syntactic sugar in EUREMA since we have not encountered yet any need for it. We think that the individual constructs of the EUREMA language are sufficiently simple and do not require any simplification in terms of syntactic sugar.

18. *Permit comments*. The graphical notation should allow users to make comments for documentation purposes, for instance, to justify modeling decisions the user has made.

EUREMA permits comments either directly in the model (see `description` attribute of several metamodel elements in Appendix A) or in the modeling editor (see Section 8.3).

19. *Provide organizational structures for models*. This guideline matters when the DSL models are complex. In this case, organizing and structuring a model into multiple, interrelated files and in a hierarchy (*e. g.*, directories) become necessary to cope with the complexity.

In EUREMA, each FLD model and the LD model can be saved its own file and stored in arbitrary directories. An LD further references the FLDs when structuring them in layers. Thus, the different models can be stored in individual files. From a technical point of view, even a single FLD or LD can be split among multiple files based on the corresponding flexibility offered by EMF that we use for the EUREMA implementation (*cf.* Section 8.3).

20. *Balance compactness and comprehensibility*. Compact notations ease creating/writing the model while a more verbose notation supports comprehensibility. Consequently, a trade-off between compactness and comprehensibility has to be found for the notation at hand.

With EUREMA, we focus rather on comprehensibility than on compactness. A major goal of EUREMA is to make the feedback loop visible in the architectural design of self-adaptive software, which requires visual expressiveness. Moreover, we think that EUREMA models are more often read than written, particularly, as they are kept alive at runtime, may evolve at runtime, and finally can be inspected at runtime by engineers. Thus, the EUREMA models go with the self-adaptive system throughout the life time of the system. However, we consider compactness of the language by providing a set of core elements that can be optionally stereotyped or labeled (see Guidelines 10 and 11). Omitting the stereotypes and labels improves the compactness but degrades the comprehensibility.

21. *Use the same style everywhere*. If a DSL consists of several sublanguages (*e. g.*, diagrams), the same notational style should be used for all of them and for all elements that appear in multiple diagrams. A common style improves understandability of the DSL and models.

As outlined for Guideline 16, we use the same notational style for elements in the FLDs and LDs in EUREMA if these elements share similarities. In general, the style of EUREMA is inspired by UML while still being sufficiently different with respect to the concrete notation of UML. We keep such a difference to clearly separate adaptation concerns, which are addressed with EUREMA, from business logic concerns, which are addressed by any other language but most likely UML. This promotes separation of concerns (*cf.* Section 9.1.1).

22. *Identify usage conventions*. The last guideline for the concrete syntax advises identifying usage conventions. Such conventions are not defined within the DSL but they are additional regulations that can be enforced on the use of the DSL (*e. g.*, how identifiers of model elements are formatted [241]). They may support comprehensibility of the DSL.

So far, we have not defined any usage conventions for EUREMA since all regulations to obtain models that are valid for execution are defined within the language. Further aspects such as the format of identifiers are ensured by EUREMA that automatically creates them.

Abstract Syntax. The last group of guidelines concerns the abstract syntax and thus the metamodel of a DSL. The abstract syntax of EUREMA is discussed in detail in Appendix A.

23. *Align abstract and concrete syntax*. This guideline calls for aligning the abstract and concrete syntax "to ease automated processing, internal transformations and also presentation (pretty printing) of the model" [241, p. 11]. Three principles are proposed for the alignment:(1) Language elements with a different concrete syntax should also be differentiated in the abstract syntax. (2) Elements with a similar meaning can be jointly structured in the metamodel by using inheritance. (3) An element of the abstract syntax and its semantics should be independent of the context, in which the element is used within a model.

The concrete and abstract syntaxes of EUREMA are aligned as we consider the three principles. Elements with a different concrete syntax are distinguished in the abstract syntax. Each element of the concrete syntax is mapped one-to-one to an element of the abstract syntax (see Section A.2). Moreover, elements with similar semantics (*e. g.*, the different types of operations in FLDs) are structured in an inheritance hierarchy within the metamodel (see Section A.1). Finally, all elements do not depend on the context they are used in, that is, each element has the same semantics regardless of its relationships to other elements.

24. *Prefer layout which does not affect translation from concrete to abstract syntax*. This guideline demands that the layout of a diagram does not influence the representation of the corresponding model in the abstract syntax as well as the semantics of the model.

In EUREMA, the layout of the FLDs and LDs does not modify or introduce any additional semantics. For instance, instead of interpreting the layers in an LD by the layout in terms of modules that are positioned on top of each other, the layers are explicitly modeled. In FLDs, the layout of the model operations and the control flow among the operations almost forms a cycle to highlight the feedback loop. This layout improves understandability while it does not affect the semantics. For instance, the operations and control flow can also be positioned in a row similar to a workflow while still describing a valid feedback loop.

25. *Enable modularity*. Due to the complexity of software, the language should be modular to allow users to decompose, model, and compose individual parts of the software.

EUREMA provides such a modularity. Engineers may use a distinct FLD for each feedback loop and for the coordination of multiple loops (*cf.* Section 5.2). A single feedback loop can even be decomposed and modeled in multiple FLDs (*cf.* Section 5.1.4). Finally, having specified the feedback loops with FLDs, the LD composes and structures individual instances of these FLDs in a layered architecture. Consequently, the modularity of EUREMA distinguishes between behavioral (FLD) and structural (LD) aspects while supporting a further decomposition of the behavioral aspect by using multiple FLDs.

26. *Introduce interfaces*. The last guideline suggests that a DSL provides interfaces similar to programming languages to enable information hiding and support modularity.

Adopting a dynamic typing approach, EUREMA only introduces an implicit notion of interfaces. For example, if we want to invoke an FLD instance, the initial and final states and the runtime models that should be passed as parameters to the instance are the signature used by complex model operation for the invocation (*cf.* Section 5.1.4). Such a signature is

similar to an interface of an FLD. Having multiple FLDs with the same interface, the FLDs can be exchanged and still be invoked by the same complex model operation. Likewise, a basic model operation has return states as well as runtime models as input and output, which all serve as a signature/interface to invoke the implementation of this operation. Consequently, the implementation of a basic operation can be simply exchanged if the replacing and replaced implementations have the same signature/interface.

9.1.3 *Discussion*

In this section, we have outlined and assessed the language design of EUREMA against the 26 design guidelines for DSLs that have been proposed by Karsai et al. [241]. Since these guidelines are qualitative, the assessment tends to be rather subjective. Nevertheless, due to the breadth of these guidelines, they provide a reasonable basis for an initial evaluation of a language design. In this context, Karsai et al. [241] state that some guidelines might be contradicting or irrelevant depending on the concrete language, purpose, and domain such that they should be seen as proposals to be studied when designing a DSL. They further argue that considering explicit guidelines improves the language design.

For the EUREMA language, we have considered and addressed all of the 26 DSL guidelines. Consequently, we have preliminary evidence that we have achieved a "good" quality for the language design of EUREMA. We think that a comprehensive evaluation of the design can be conducted over a longer period of time when more experience has been gained in the use of EUREMA, for instance, when EUREMA has been applied more often, when more tools have been built for EUREMA (*e.g.*, a code generator or a debugger), and when EUREMA itself evolves. Such aspects are all influenced by the language design.

9.2 LANGUAGE EXPRESSIVENESS

In this section, we investigate how expressive EUREMA is with respect to the domain of self-adaptive software. For this purpose, we apply EUREMA to examples from literature to see whether EUREMA is expressive enough to capture a variety of aspects in this domain and therefore to validate the language against the domain (*cf.* [241]). On the one hand, we already have initial evidence about the expressiveness based on modeling our mRUBiS example introduced in Section 4.5. Therefore, we modeled the self-repair and self-optimization feedback loops for mRUBiS, the coordination of both feedback loops, and a higher-layer feedback loop controlling the self-repair feedback loop (*cf.* Chapter 5).

To provide additional evidence about the expressiveness, we use EUREMA to model three further approaches to self-adaptive software from literature, namely, *Rainbow* [170], *DiVA* [306], and *PLASMA* [410]. We selected these approaches for three reasons. First, each of them is described at a sufficient level of detail in the literature, which allows us to create the corresponding EUREMA models. Second, these approaches have different characteristics: They are driven by different techniques that are either ADLs (Rainbow and PLASMA) or MDE (DiVA). They have different goals of either providing a reusable framework (Rainbow), managing dynamic variability with runtime models (DiVA), or supporting plan-based adaptation in a three-layer architecture (PLASMA). They either employ a single (Rainbow and DiVA) or two layered feedback loops (PLASMA). Third, each of them can be considered as the state of the art with respect to their individual goals. Thus, these three approaches are different and cover a broad variety of solutions in the domain of self-adaptive software. In the following, we discuss the EUREMA models for each of them.

9.2.1 *Rainbow*

The *Rainbow* approach supports the architecture-based development of self-adaptive software [113, 170]. Its major goal is the cost-effective development by providing a reusable framework for the adaptation engine. The framework allows engineers to customize the engine to a specific system. However, it prescribes the architecture of the engine by supporting exactly one feedback loop as shown in the LD in Figure 73. It further prescribes the adaptation activities and their structuring in the feedback loop. We modeled this predefined feedback loop in the FLD depicted in Figure 74. In contrast to this FLD, the diagrams in [113, 170] provide views of Rainbow that do not make the used runtime models explicit.

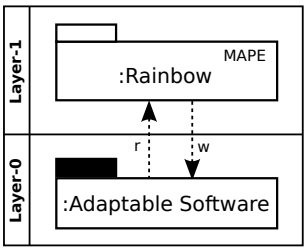


Figure 73: LD for Rainbow.

Concerning the FLD in Figure 74, Gauges realize the monitoring by abstracting from the sensors of the adaptable software and collecting data (*e. g.*, average response times) that is relevant for the concern (*e. g.*, performance) managed by the self-adaptation. Gauges notify the Model Manager about changes in the software by providing Gauge Events that reflect those changes at the abstraction level of the Architecture Model and Environment Model. Thus, the model manager directly uses these events to update the architecture model if the software or the resource utilization have changed, or the environment model if resources have been added or removed from the system. In the FLD, the gauge events are reflected by a gray-shaded element since Rainbow does not explicitly capture them in a model.

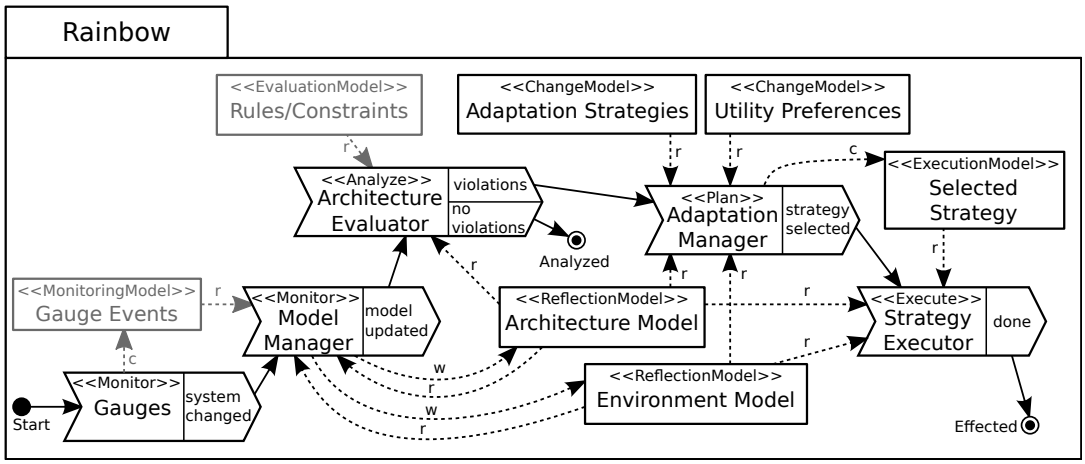


Figure 74: FLD for Rainbow.

Whenever, the model manager updates the architecture model, it triggers the Architecture Evaluator that analyzes this model by applying Rules/Constraints. Rules and constraints are

specified as part of the architecture model and not by a distinct evaluation model. However, to make them visible in the feedback loop, we depict them by a gray-shaded element in the FLD. Rules or constraints check, for instance, whether monitored response times exceeds a given threshold. The feedback loop terminates if no rule or constraint is violated. Otherwise, the Adaptation Manager is executed to plan an adaptation. Therefore, based on given Utility Preferences among multiple concerns and the current reflection models, a Selected Strategy from the repertoire of Adaptation Strategies is chosen. Strategies are similar to ECA rules specifying a reconfiguration. The selected strategy is the most promising one to address the adaptation needs constrained by the utility preferences. Finally, the Strategy Executor enacts the selected strategy by mapping and executing it on the effectors of the adaptable software. To properly execute a strategy, the executor uses the architecture and environment models to identify software elements and resources, which are referenced by the strategy. Changes of the adaptable software caused by executing the strategy are reflected in the architecture model by monitoring in the next run of the feedback loop.

The Rainbow framework is based on the *Acme* ADL to express the architecture and environment models. Rules and constraints as part of the architecture model are specified in a first-order predicate logic provided by Acme. Adaptation strategies and utility preferences are defined in *Stitch* [114]. Though Rainbow does not employ runtime models that follow MDE principles, the EUREMA language is able to capture Rainbow's feedback loop while making the Acme and Stitch models explicit in the feedback loop design.

9.2.2 *DiVA*

The goal of the *DiVA* project is to manage dynamic variability in complex, adaptive systems. Therefore, an MDE approach to specify and execute self-adaptive software has been proposed [306–308], that is, MDE techniques and runtime models drive the feedback loop. As shown by the LD in Figure 75, DiVA employs a single feedback loop that we decomposed in two modules. We modeled this feedback loop in the FLD depicted in Figure 76.

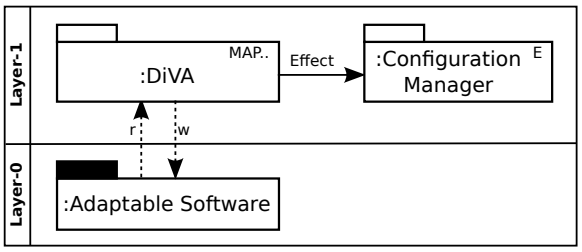


Figure 75: LD for DiVA.

DiVA relies on sensors that monitor the adaptable software and context. In Figure 76, the CEP (*i. e.*, the complex event processor) consumes, analyzes, and filters sensor events to update the Architecture Model and the Context Model that reflect the software and context. As the context drives the adaptation in DiVA, the monitoring activity terminates the feedback loop if there is no relevant context change. Otherwise, the Reasoning Engine is triggered to find a new target configuration suitable for the current context. Therefore, reasoning is performed on the architecture and context models, which is guided by a Reasoning Model and Aspect Models. Aspect models define the variability of the system in terms of features. The reasoning model specifies the mechanism to determine which aspects should be activated

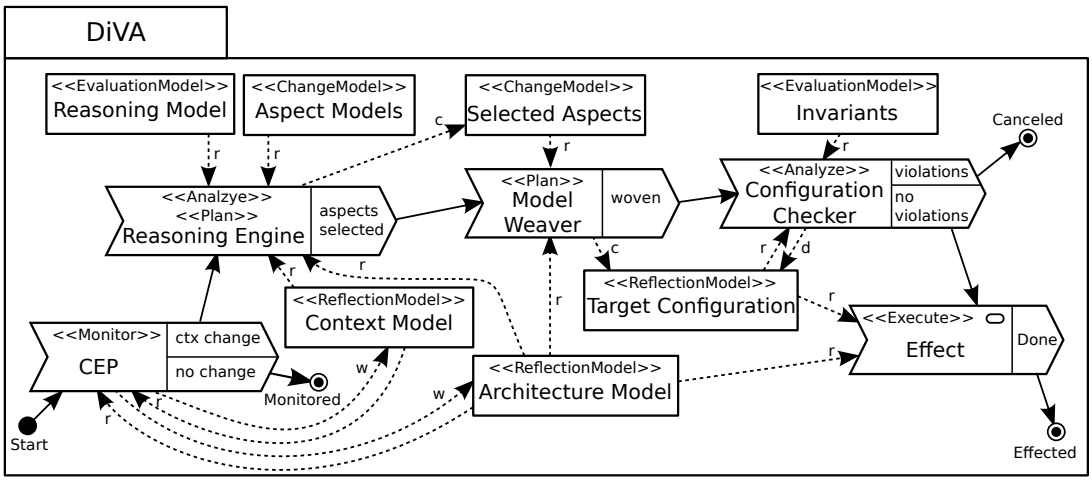


Figure 76: FLD for DiVA.

or de-activated in the current configuration. These Selected Aspects are woven or removed by the Model Weaver from the architecture model to obtain the Target Configuration described in a newly created model. Before enacting the target configuration, the Configuration Checker evaluates the Invariants on it. If any invariant is violated, the configuration checker discards the target configuration and terminates the feedback loop. Otherwise, the complex model operation Effect invokes the Configuration Manager defined in the FLD shown in Figure 77. This invocation executes the adaptation by moving the adaptable software from the current configuration reflected in the architecture model to the target configuration.

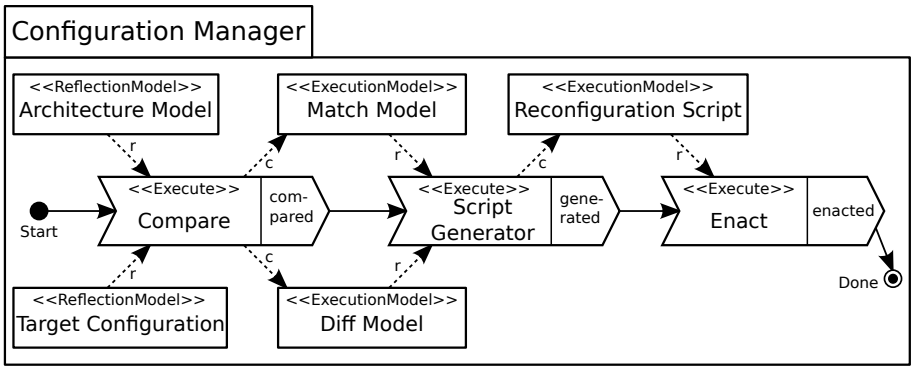


Figure 77: FLD for the Configuration Manager of DiVA.

Therefore, the Compare operation compares the architecture model with the target configuration to obtain a Match Model and a Diff Model (*cf.* Figure 77). These models describe the common respectively the different model elements of the architecture model and the target configuration. Thus, they represent which architectural elements should remain unchanged and which elements should change when moving from the current to the target configuration. This information is used by the Script Generator to create a Reconfiguration Script that is finally executed by the Enact operation using the effectors of the software.

Since DiVA is driven by runtime models and MDE techniques to weave and compare these models, we can easily describe it with EUREMA that targets such MDE approaches.

9.2.3 *PLASMA*

The *PLASMA* approach [410] proposes a three-layer architecture for plan-based adaptation, that is, its goal is the generation of plans for adapting and executing software applications. We modeled the three-layer architecture in the LD shown in Figure 78.

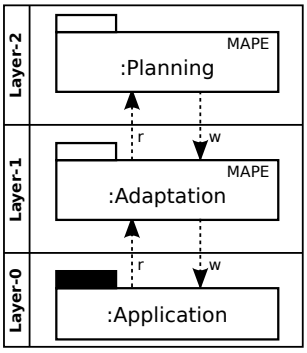


Figure 78: LD for PLASMA.

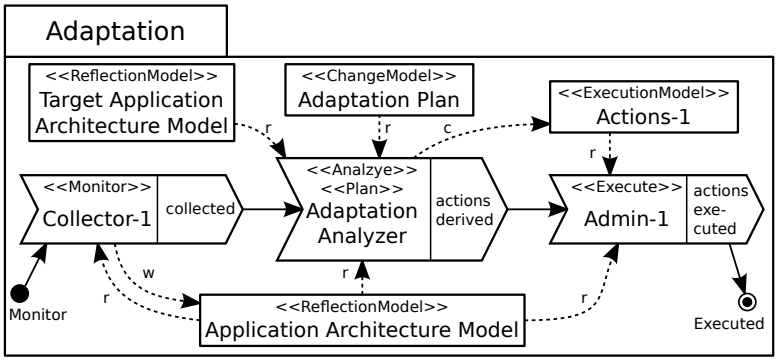


Figure 79: FLD for the Adaptation Layer in PLASMA.

While the adaptable application is located at the lowest layer, the feedback loop of the middle layer adapts the application and the highest-layer feedback loop (re)generates plans to be executed by the two lower layers. We modeled these two feedback loops with FLDs.

The Adaptation feedback loop of the second layer is defined by the FLD in Figure 79. The Collector-1 operation monitors the application and maintains the Application Architecture Model reflecting the application. This model is used by the Adaptation Analyzer to execute the Adaptation Plan provided by the third-layer feedback loop. This plan specifies the adaptation to move the current application architecture to the target architecture defined in the Target Application Architecture Model by the third-layer feedback loop. Additionally, the Adaptation Analyzer analyzes any deviations in the current application architecture and resolves them to align it with the target architecture. Therefore, reconfiguration commands (Actions-1) are created and enacted by the Admin-1 operation to the application.

The third-layer Planning feedback loop is defined by the FLD in Figure 80. It is executed when plans are generated initially or replanning is required. The Application Planner uses a domain model (Application Domain Description) and the initial and goal states (Application Problem Description) of the application, which are all provided by the developer. It devices an Application Plan that will be executed by the application to reach the goal state as well as the Target Application Architecture Model prescribing the application architecture that is able to execute the plan. This results in the adaptation problem of how to move the application from the current to the target architecture. Therefore, the Adaptation Planner devises an Adaptation Plan based on the current (*i. e.*, the Application Architecture Model maintained by the second-layer feedback loop) and target (*i. e.*, the devised Target Application Architecture Model) application architecture. Thereby, it takes the Adaptation Domain Description into account, which is provided by the developer and prescribes the architecture and capabilities of the second-layer feedback loop that will execute the generated adaptation plan.

The following operations adapt the second-layer feedback loop to enable the execution of the adaptation plan. The Collector-2 updates the Adaptation Architecture Model reflecting the second-layer feedback loop by monitoring. The Analyzer enriches this model with the Adaptation Plan and the Target Application Architecture Model to provide them to this feedback loop (see Figure 79). Moreover, it devises commands (Action-2) to adapt this loop tak-

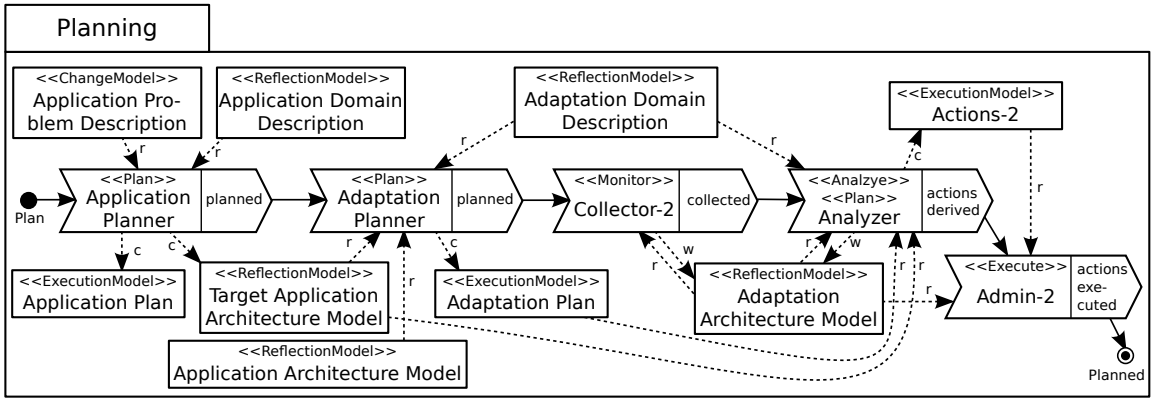


Figure 80: FLD for the Planning Layer in PLASMA.

ing the adaptation plan and the loop's capabilities (Adaptation Domain Description) into account such that the loop is able to execute the plan (*e. g.*, by replacing the Adaptation Analyzer shown in Figure 79). Finally, the Admin-2 operation adapts the second-layer loop by executing the reconfiguration commands and synchronizing the Adaptation Architecture Model. The latter injects the new adaptation plan and target application architecture model into the second-layer loop that is going to execute this plan afterwards.

Overall, the EUREMA language is able to express PLASMA's three-layer architecture and feedback loops. However, it is difficult to asses whether the EUREMA models properly describe the feedback loops as PLASMA encodes the runtime models within the middleware instead of capturing them and defining their usage by the feedback loops explicitly.

9.2.4 *Discussion*

In this section, we showed the results of our modeling effort to express three existing, state-of-the-art approaches to self-adaptive software from the literature with EUREMA. These approaches are Rainbow, DiVA, and PLASMA, which we selected due to their different characteristics: They are driven by different techniques that are either ADLs (Rainbow and PLASMA) or MDE (DiVA). They have different goals of either providing a reusable framework (Rainbow), managing dynamic variability with runtime models (DiVA), or supporting plan-based adaptation in a three-layer architecture (PLASMA). Thereby, they either employ a single (Rainbow and DiVA) or two layered feedback loops (PLASMA). Thus, these approaches are quite different (*e. g.*, considering the design of the feedback loops) and cover a broad range of solutions for self-adaptation. Comparing the individual EUREMA models for these approaches makes such differences visible. The EUREMA models clearly characterize the individual feedback loops, adaptation activities, and runtime models of the approaches, which are typically neglected such as in Rainbow and PLASMA.

Overall, the results of this modeling effort demonstrate that the EUREMA language is expressive enough to capture these three third-party approaches with their different characteristics.[2] This increases the initial evidence of the expressiveness of EUREMA that we obtained from modeling our application example since these three approaches are the works by other researchers and they are quite different in their design. Hence, we think that EUREMA can express a variety of solutions in the domain of self-adaptive software.

2 We completely modeled these three approaches with EUREMA except of the individual triggering conditions in the LDs because the corresponding papers do not discuss the triggers of feedback loops in sufficient detail.

Consequently, it can be useful for and applied by other researchers and developers. For instance, Piskachev [349] successfully applied the EUREMA language to specify a feedback loop for controlling the performance of cloud-based software systems and in particular of the *Znn.com* system[3] developed by Cheng et al. [116]. He further integrated the EUREMA interpreter into the *SimuLizar* framework to support simulation of the feedback loop and prediction of quality of service attributes for the controlled Znn.com system.

However, the obtained evidence about the expressiveness of EUREMA is limited since the assessment of the language considers only our application example including several variants (*cf.* EUREMA models in Chapter 5), three third-party approaches (Rainbow, DiVA, and PLASMA), and the application to Znn.com within SimuLizar. Thus, the findings of the assessment cannot be generalized to any self-adaptive software, that is, we cannot conclude that the EUREMA language is expressive enough to describe *any* feedback loop.

9.3 EXPERIMENTAL APPLICATION EXAMPLE

To illustrate EUREMA throughout this thesis, we have used the Modular Rice University Bidding System (mRUBiS) [18] extended with an adaptation engine realizing self-repairing and self-optimizing capabilities as a running example (*cf.* Section 4.5). Particularly, we modeled the corresponding feedback loops and the layered architecture of the overall system with EUREMA in Chapter 5. In this section, we report in detail about the experiments we conducted with the resulting EUREMA-based feedback loops and the running mRUBiS.

For the experiments, we completed the realization of the feedback loops by implementing the individual model operations that complement the EUREMA models specifying the feedback loops. Thus, we obtained fully implemented feedback loops that can be executed by the EUREMA interpreter. In this context, we emphasize that the goal of the experiments is to demonstrate the capabilities of EUREMA and to show that EUREMA can be effectively applied to specify and execute feedback loops on top of a running system. Therefore, we do not aim for developing novel or advanced mechanisms for the analysis and planning activities of a feedback loop. In contrast, we use an action-/rule-based mechanism (*cf.* Section 2.2.3), which is a basic approach but also the foundation for more sophisticated goal- and utility-based approaches [247]. The experiments are based on our implementation framework (*cf.* Chapter 8) that we customized as shown in Figure 81.

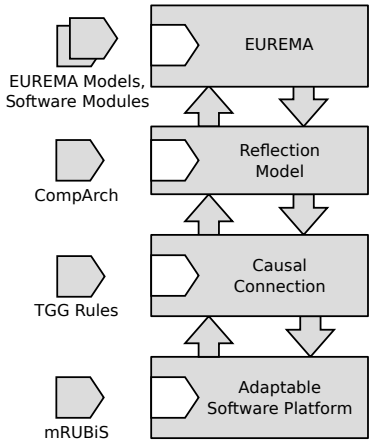


Figure 81: Customized Implementation Framework for the Experiments (*cf.* Chapter 8).

3 Znn.com: http://www.self-adaptive.org/exemplars/model-problem-znn-com/.

For all experiments, the EJB-based *mRUBiS* is the adaptable software, the *TGG Rules* are the synchronization rules that specify the causal connection between mRUBiS and the reflection model, and the *CompArch* language is used to express the reflection model (*cf.* Section 8.2). For the individual experiments, we have different *EUREMA Models* specifying the feedback loops and *Software Modules* implementing the model operations of these loops.

In the following, we first detail mRUBiS and motivate its selection as the application example. Then we report about the individual experiments with variants of the adaptation engine:(1) self-repairing mRUBiS (Section 5.1), (2) self-optimizing mRUBiS (Section 5.2), (3) coordinating the self-repair and self-optimization of mRUBiS (Section 5.2), and finally (4) the three-layer architecture for self-repairing mRUBiS (Section 5.3). All of these experiments were conducted on one machine[4].

9.3.1 *Detailing and Motivating the Use of mRUBiS*

mRUBiS is an internet marketplace on which users sell or auction products. Technically, it is derived from RUBiS, which is a case study and benchmark to evaluate the performance of application design patterns and application servers.[5] Moreover, RUBiS has become a popular case study for self-adaptive systems managing performance [343]. We extended RUBiS with new functionalities, migrated it to version 3 of Enterprise Java Beans (EJB) [132], and modularized it into 18 components. The modularization enables architectural adaptations that are otherwise not possible with the monolithic RUBiS. With the migration to EJB 3, mRUBiS can be executed and dynamically adapted on our platform (*cf.* Section 8.1).

The mRUBiS architecture is shown in Figure 82. It consists of components to manage the products (*Item Management Service*), users (*User Management Service*), auctions and purchases (*Bid and Buy Service*), inventory (*Inventory Service*), and rating of users (*Reputation Service*), and components to authenticate users (*Authentication Service*) and to persist (*Persistence Service*) and retrieve data (*Query Service*) from the database. To improve the results when users search for items (*i. e.*, products), a pipe of filter components is used. This pipe follows the batch sequential pipe-and-filter architectural style. It iteratively filters the list of items obtained by the query service by removing items that are not considered as relevant for the specific user and search request. Thus, the pipe of filters aims for improved search results while performing the filtering increases the response time for the user.

The company running mRUBiS aims for high sales volumes by achieving customer satisfaction and encouraging customers to additional purchases. Therefore, the system should be highly available and the response times should be low or at least acceptable. To reach these properties, self-adaptation should be employed to automatically repair failures (*i. e.*, self-healing or self-repair by detecting, diagnosing, and recovering from disruptions [184, 350]) and improve performance (*i. e.*, self-optimization by reconfiguring the system [443]). For instance, to mitigate faulty components, they can be restarted, redeployed, or replaced with alternatives. Performance can be improved by reconfiguring the pipe of filter components. Changing the number of filter components and reordering them result in different performance characteristics that are exploited for self-optimization.

We selected mRUBiS as our application example because it is based on the well-known RUBiS case study [343] and in particular it allows us to investigate a broad range of self-adaptation problems. For instance, we investigate two different self-adaptation capabili-

4 Quad-core CPU (Intel Core i5-2400, 3.10GHz), 8GB RAM, Ubuntu 14.04 (Kernel 3.13), Java SE Runtime Environment 1.8.0_40, EMF Runtime and Tools 2.6.0, and GlassFish v2.1.1-b31g.

5 Rice University Bidding System (RUBiS): http://rubis.ow2.org/.

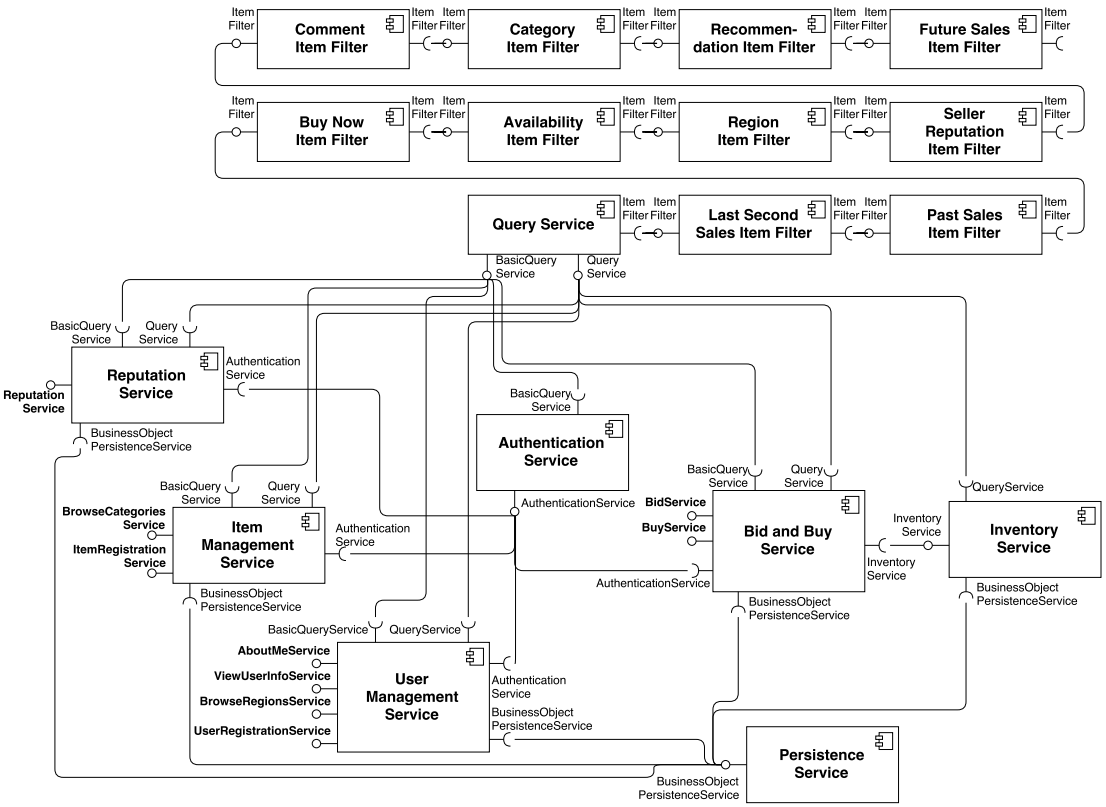


Figure 82: Architecture of mRUBiS.

ties (*i. e.*, the self-repair and self-optimization of mRUBiS), the coordination of these two capabilities, and the stacking of feedback loops for the self-repair case. Thereby, we consider parameter and structural adaptation (*e. g.*, restarting a component is realized with parameter adaptation while replacing a component with an alternative requires structural adaptation that reconfigures the architecture). In contrast, these different aspects are not addressed by the popular *Znn.com* exemplar. Znn.com primarily supports the management of performance (*i. e.*, self-optimization)[6], which is furthermore only realized with parameter adaptation [411]. Hence, Znn.com does not support other capabilities and variants of self-adaptation as well as structural adaptation such that it is too limited for our needs.

9.3.2 *Self-Repairing mRUBiS*

The self-repair feedback loop should automatically identify and handle disruptions in the running mRUBiS. We have modeled this feedback loop with the FLD depicted in Figure 15 on Page 57. For the experiment, we use a slightly simplified feedback loop as specified by the FLD in Figure 83. The simplification only omits the decision operation and combines the two analyzing operations into one. Employing this (simplified) feedback loop on top of mRUBiS results in the two-layer architecture modeled with the LD in Figure 17 on Page 60.

The goal of the self-repair is to improve the availability of mRUBiS. We defined five Critical Failures (CFs) that disrupt the operation of mRUBiS and that should be handled by the self-repair: *(CF1)* A component has crashed such that its life cycle state is out of order but it is still operating to some extent. *(CF2)* The use of functionality through a provided interface of a component causes multiple exceptions exceeding a given threshold. *(CF3)* A

6 Znn.com: see [116] and http://www.self-adaptive.org/exemplars/model-problem-znn-com/.

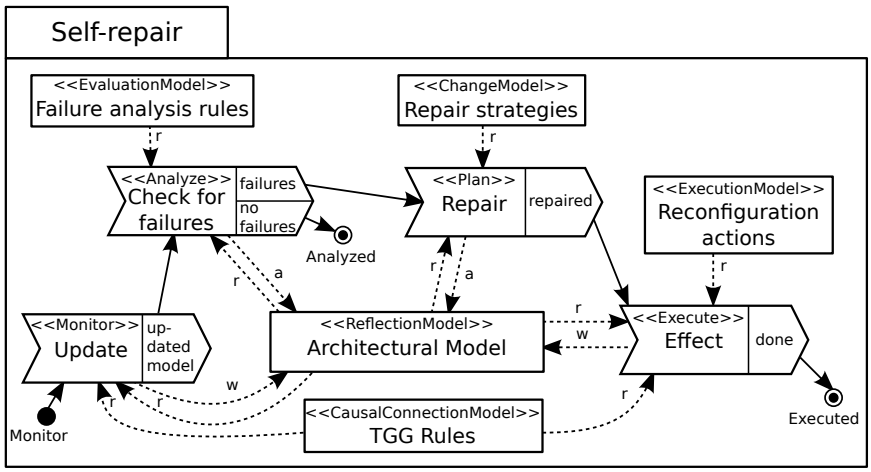


Figure 83: Simplified FLD for the Self-repair Feedback Loop.

component has crashed and disappeared completely from the system. *(CF4)* A component is continuously affected by CF1s or CF2s. Thus, a disruption persists in the system. *(CF5)* A connector has crashed such that the connection between two components is lost.

The system elements related to these CFs are observable in the running mRUBiS through the adaptable software platform we use (*cf.* Section 8.1). Among others, we can observe the component life cycle (CF1/CF4), exceptions (CF2/CF4), components (CF3), and connectors (CF5). These elements are further represented in the CompArch-based reflection model through the causal connection (*cf.* Section 8.2). The feedback loop operates on this reflection model that describes the runtime architecture of mRUBiS including these system elements. This model corresponds to the Architectural Model in the FLD depicted in Figure 83.

In the following, we discuss the realizations of the individual model operations of the self-repair feedback loop, which constitute the software modules provided together with the EUREMA models to the EUREMA interpreter. Then, we evaluate the feedback loop.

Update. The monitoring operation is realized with code that invokes the causal connection with the TGG Rules to incrementally update the Architectural Model based on changes of the running mRUBiS (*cf.* Section 8.2). The model then reflects the current state of mRUBiS.

Check for failures. This operation analyzes the updated Architectural Model for any CF. It is model-driven and event-/state-based (*cf.* Chapter 7). On the one hand, this means that models are used to specify and execute the analysis. For this purpose, we use executable Story Diagrams (SDs)[7]. On the other hand, the analysis is driven by change events resulting from updates of the Architectural Model (*i. e.*, of the state) done by the monitoring operation. These events point to individual elements of the model which have changed due to monitoring. Thus, the analysis can be performed right away on these elements without having to search and check the whole model, which supports an incremental processing.

The implementation of the operation consists of a code snippet that consumes the change events for the Architectural Model. Based on these events, it runs the relevant analyses by invoking the SD interpreter to execute the corresponding SDs. For instance, if exceptions have occurred when using a provided interface of a component in the system, the model is updated by representing these exceptions as failures attached to the provided interface in the model. Such an update causes a specific change event that indicates a potential occurrence of CF2. The corresponding SD for identifying a CF2 is shown in Figure 84.

7 A brief introduction of Story Diagrams (SDs) is given in Section B.1.

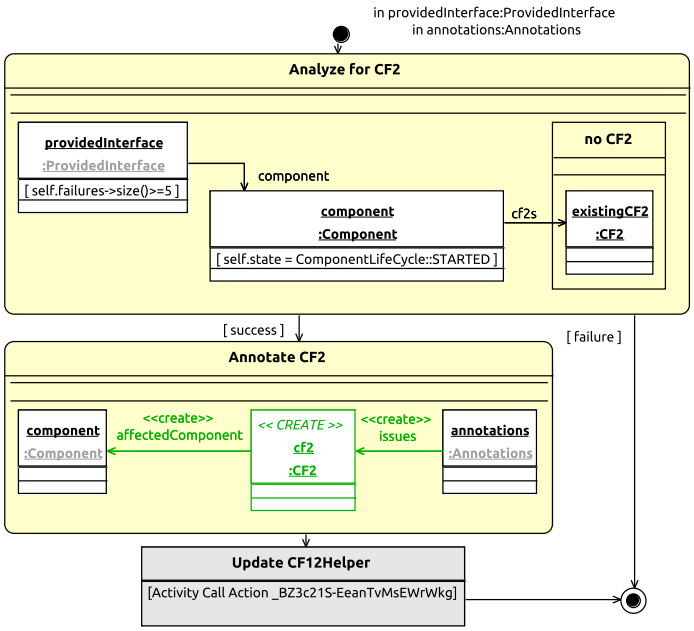


Figure 84: Story Diagram Specifying the Analysis for CF2.

This SD has two parameters: The ProvidedInterface model element obtained from the change event that notifies about the addition of failures to this element. The Annotations element that contains all annotations made to the mRUBiS architecture in the model.[8] The first Story Pattern (SP) called *Analyze for CF2* checks if there are at least five failures attached to the given interface, if the Component providing this interface is started (*i. e.*, running), and if this Component is not already annotated and thus affected by a CF2 identified in an earlier run of the feedback loop. If this condition holds, a CF2 has been identified and the second SP annotates the affected Component with a CF2 marker element. Finally, another SD (not shown here) is invoked to update the history of CF1 and CF2 occurrences , which is required for identifying CF4s. If no CF2 has been identified, the SD terminates immediately.

For each of the five CF types, we specified such an SD that analyzes the model for an occurrence of this CF type and where necessary creates a corresponding annotation. All of the SDs together constitute the evaluation model called Failure analysis rules in the FLD in Figure 83. Thus, the result of the analysis operation is the Architectural Model that is annotated with CF marker elements representing the occurrences of CFs in mRUBiS.

Repair. For each of the identified CFs, the planning operation decides which Adaptation Strategy (AS) should be applied to repair the CF. Five strategies are available: *(AS1)* Redeploying a component with the same configuration. *(AS2)* Restarting a component. *(AS3)* Redeploying a component with a new configuration. *(AS4)* Replacing a component with an alternative component of a different type that provides the same functionality. *(AS5)* Recreating a connector between two components. We use a very basic planning approach that maps each CF to an AS that should repair the CF. We specify such mappings with SDs, for instance, a CF2 should be handled by AS2 as shown in Figure 85.

The *Check CF2* action invokes the SD shown in Figure 86 to check whether the given CF2 marker still exists and is associated to a component with at least one provided interface that has five or more failures. If so, the SD identifies the context for AS2 (*cf.* Figure 85). The

8 Technically, we extended the CompArch metamodel to support annotations in terms of marker elements that are associated to individual architecture elements in a CompArch model.

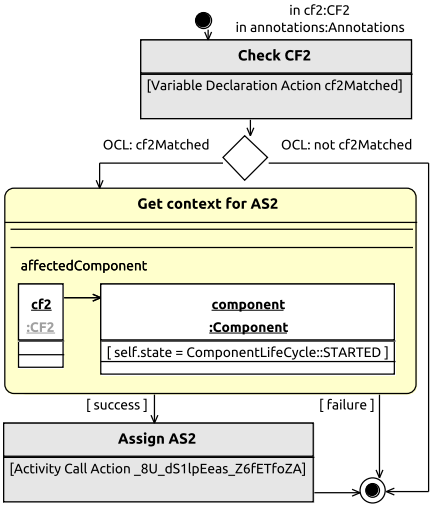


Figure 85: Planning the Repair of CF2.

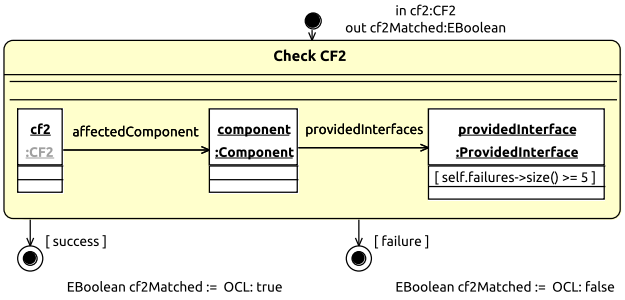


Figure 86: Checking CF2.

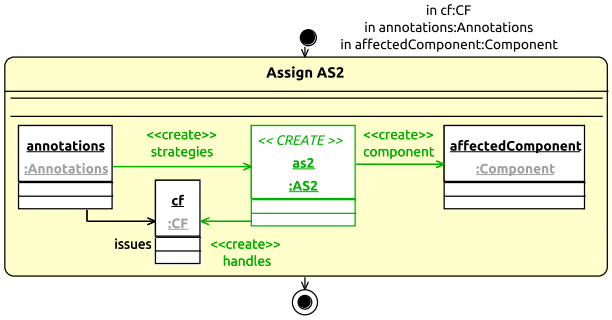


Figure 87: Assigning AS2.

context of an AS is the precondition of applying the strategy. For instance, AS2 that restarts a component requires that the component is started, otherwise a restart is not possible. If the precondition holds, the Assign AS2 action invokes the SD shown in Figure 87 to assign AS2 to repair the given CF. The assignment is realized by creating an AS2 marker element and associating it to the CF to be repaired and to the component to be restarted. Similar to the CF markers, AS markers are annotations to architecture elements in the model.

Similar to the analysis step, the planning operation supports an incremental processing since it does not search through the whole model to assign strategies. Given the CF marker elements added by the previous operation to the model, the implementation of this operation is a code snippet that invokes the SD interpreter to execute the corresponding planning SD for each CF marker. Then, each SD starts planning from such a CF marker (*cf.* input parameters of the SD in Figure 85), from which it can directly obtain the component or connector that is affected by the critical failure and that is the target of the adaptation.

For each of the five CF types, we created similar SDs that together specify the planning. The planning deterministically maps AS*i* to CF*i* for the five types of strategies and critical failures. All of the SDs together constitute the change model called Repair strategies in the FLD in Figure 83 on Page 188. The result of the planning operation is the Architectural Model annotated with AS marker elements. These elements represent the adaptation strategies that are assigned to the occurred CFs and that should be executed by the next operation.

Effect. The last operation of the feedback loop executes the adaptation strategies assigned by the previous planning operation to effect mRUBiS. Following a models@run.time approach, the adaptation is first performed on the Architectural Model and then synchronized to mRUBiS. This approach allows us to specify and execute the strategies with SDs that operate at the model level. For instance, Figure 88 shows the SD for AS2 (component restart).

The SD first checks whether the component is started. If not, restarting the component is not possible and the SD terminates. Otherwise, the component is stopped at the model level (*i. e.*, its life cycle state is set to DEPLOYED). The following action (*Enact stop component*)

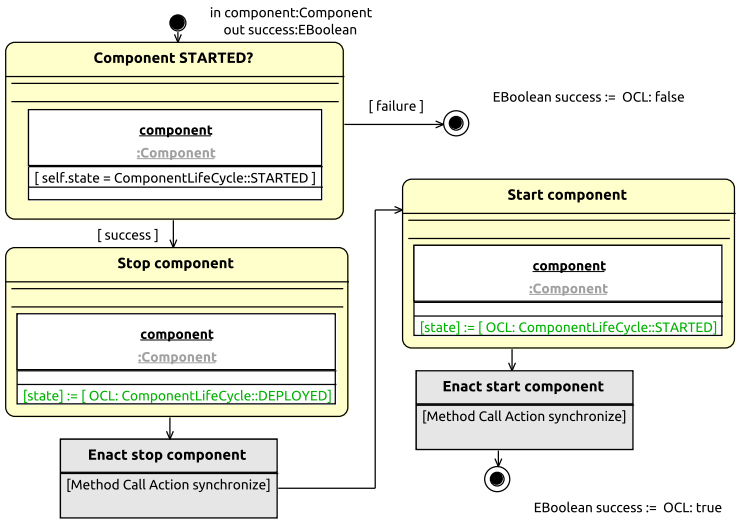


Figure 88: Restarting a Component (AS2).

uses the causal connection to incrementally propagate this model-level adaptation step to mRUBiS (*cf.* Section 8.2). Afterwards, the component is started again at the model level (*i. e.*, its life cycle state is set to STARTED), which is also synchronized to mRUBiS (*Enact start component*) similar to the previous step. This accomplishes a restart of the component.

For each of the five strategies AS1 to AS5, we created a similar SD. All of these SDs constitute the execution model Reconfiguration actions in the FLD in Figure 83 on Page 188. Given the AS marker elements added by the planning operation to the model, this operation retrieves the related CF marker elements as well as the components or connectors that are affected by the critical failures. For each AS marker, it executes the corresponding SD. When synchronizing adaptation steps from the Architectural Model to the running mRUBiS, it invokes the incremental causal connection with the TGG Rules (*cf.* Figure 83 on Page 188).

Thus, this operation is driven by changes of the Architectural Model in terms of the AS markers that have been added to the model by the previous operation. This supports an incremental processing as the execution of a strategy starts with an AS marker that is directly associated to the CF marker and affected component or connector. With this information, the operation invokes the SD interpreter to execute the corresponding SD that realizes the strategy. Thereby, the adaptation performed at the model level is incrementally synchronized to mRUBiS. Having executed a strategy, the operation removes the corresponding AS and CF markers from the model. Afterwards, the feedback loop terminates.

The realization of the self-repair feedback loop shows how adaptation models such as SDs are used to specify and execute the individual operations of a feedback loop. We presented the SDs addressing CF2 and AS2 due to their simplicity since the SDs for the other failures and strategies are more complex. Moreover, the realization shows how EUREMA can be used to develop a feedback loop that addresses multiple issues (*e. g.*, failures) and adaptation strategies. The choice of using SDs for the individual model operations is based on our assessment of SDs with respect to requirements for adaptation models [23, 27]. Finally, the realization shows how a reflection model such as the Architectural Model may capture the working data of the model operations such as the CF and AS markers and how change events relating to the model drive the execution of the individual operations.

Experiment. We conducted an experiment to show the effective application of the self-repair feedback loop and EUREMA for maintaining the availability of the running mRUBiS.

For the experiment, we deployed mRUBiS as shown in Figure 82 on Page 187 on our adaptable software platform. 50 clients are continuously and concurrently requesting services from mRUBiS, particularly from the *User Management Service* component. Each client pauses for 250 ms between receiving a reply for a request and sending the next request. Half of the clients request the *ViewUserInfoService* interface and the other half the *AboutMeService* interface. While the former service requires the *AuthenticationService* component, the latter does not. To introduce a disruption, we inject a CF3 to the *AuthenticationService* component, that is, we remove this component from the running mRUBiS. Consequently, half of the clients will be affected by this failure since their requests cannot be served by mRUBiS. Instead of a proper reply, they will receive an exception from mRUBiS. After the failure has been injected, the EUREMA interpreter is triggered to execute the self-repair feedback loop. The loop will perform AS3, that is, it will identify the type of the missing component and instantiate a new component of this type as well as deploy, configure, and start the instantiated component. We selected CF3 and AS3 for the experiment since they are the most complex adaptation scenario among all types of CF and AS discussed previously.

The result of the experiment is shown in Figure 89. The x-axis depicts the time in seconds (s) and the y-axis the success rate of the client requests (*i. e.*, the number of proper replies from mRUBiS divided by the number of all requests) for a time frame of 1 s as observed by the clients.[9] The success rate is used as a metric for the availability. A rate of 1 (0) denotes that each (no) request is successfully served.

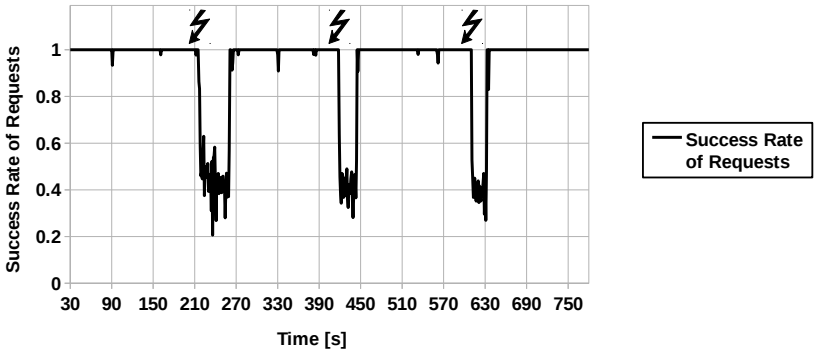


Figure 89: Experiment Results for Self-Repairing mRUBiS.

The experiment consists of the following steps. The first 30 s are the warm-up phase to initialize the clients and mRUBiS (not shown in Figure 89). Then, we inject a CF3 after 180 s and later on two times after 180 s when one run of the self-repair feedback loop has terminated (see lightning symbols in Figure 89). The figure shows that the success rate is initially 1 and that mRUBiS is not 100% reliable considering the small drops of the rate, for instance at seconds 90 and 330. For each injection, the success rate drops immediately to around 0.4 when the failure has been injected since at least half of the client requests cannot be served. The feedback loop is executed right after the injection. While it is running to repair this failure, the success rate stays around 0.4. As soon as it has repaired the failure, the rate quickly increases back to 1 in all three cases.

We repeated this experiment five times and obtained consistent results between the different runs. The results of the four other runs of the experiment are shown in Section C.1 of the appendix. Each run took in total 750 s (excluding the 30 s warm-up phase) during which on average 31406 requests have been made, out of which 28936 were successfully

9 The time when a client receives a reply or exception determines the time frame to which the request is assigned.

served. This results in an overall success rate of 0.92. We think that this rate is acceptable given the three injected failures in the short time period of the experiment and the fact that we did not employ any redundancy of components to improve availability.

Consequently, the experiment gives us some evidence that the EUREMA-based self-repair feedback loop is able to effectively repair occurrences of CF3 and to maintain the availability of the running mRUBiS in this context. Moreover, it demonstrates the successful application of EUREMA to a self-adaptation problem and a running software system.

Finally, we observe that a run of the self-repair feedback loop can take up to 50 s to recover mRUBiS from the CF3 failure (*cf.* Figure 89). In the experiment, the performance of the feedback loop is mainly affected by two aspects. First, we conducted the experiments on *one* machine such that the 50 clients, the feedback loop, the failure injector, and the adaptable software platform (mRUBiS deployed on the GlassFish and database servers) compete for limited resources. Second, mRUBiS and GlassFish fiercely throw and log exceptions while the failure is present causing additional load on the machine. Both aspects should be taken into account when considering here the performance of the feedback loop.

9.3.3 *Self-Optimizing mRUBiS*

The self-optimization feedback loop should automatically address performance issues in the running mRUBiS. We have modeled this feedback loop with the FLD depicted in Figure 29 on Page 72. Employing this feedback loop on top of mRUBiS results in the two-layer architecture (*cf.* LD in Figure 30 on Page 74 excluding the self-repair feedback loop).

The goal of the self-optimization is to keep the response time of the product search on mRUBiS at an acceptable level. Considering the mRUBiS architecture in Figure 82 on Page 187, mRUBiS provides a personalized product search as part of the *BrowseCategoriesService* offered by the *Item Management Service* component. For a search, the user is authenticated with the *Authentication Service* component, the search request is sent to the *Query Service* component that obtains relevant products from the database, and finally these products are sequentially filtered by the pipe of filter components based on user preferences. Thus, the pipe increases the quality but also the response time of the search.

In this context, we defined three Performance Issues (PIs) that indicate potential to improve the response time or quality of the search. *(PI1)* The components in the pipe are not well-ordered. Each of them processes a list of products and removes products from this list such that the size of the list decreases along the pipe. Hence, components that perform well, that is, they remove quickly a lot of products from the list, should be placed toward the beginning of the pipe and those that do not perform well toward the end of the pipe. *(PI2)* The average response time of the personalized search exceeds a given threshold. This calls for means to improve the response time. *(PI3)* The average response time of the personalized search falls below a given threshold. This allows improving the quality of the search while accepting an increase of the response time.

The data related to these PIs are observable in the running mRUBiS through the adaptable software platform we use (*cf.* Section 8.1). Among others, we can observe the average computation time and fraction of filtered products for individual filter components (PI1) as well as the average response time of any type of request (PI2/PI3). This data is further represented in the CompArch-based reflection model through the causal connection (*cf.* Section 8.2). The self-optimization feedback loop operates on this reflection model that describes the runtime architecture of mRUBiS enriched with this data. This model corresponds to the Architectural Model in the FLD depicted in Figure 29 on Page 72.

In the following, we discuss the realizations of the individual model operations of the self-optimization feedback loop, which constitute the software modules provided together with the EUREMA models to the EUREMA interpreter, and we evaluate the feedback loop.

Update. The monitoring operation is realized with code that invokes the causal connection with the TGG Rules to incrementally update the Architectural Model based on changes of the running mRUBiS (*cf.* Section 8.2). The model then reflects the current state of mRUBiS.

Check for p-issues. This operation analyzes the updated Architectural Model for any PI. Similar to the analysis operation of the self-repair feedback loop, this operation is model-driven and event-/state-based (*cf.* Chapter 7). Thus, we specified the analysis with models, particularly SDs, that are directly executed to perform the analysis. Moreover, the analysis is driven by change events caused by the monitoring operation when updating individual component or provided interface elements of the Architectural Model (*i. e.*, the state) with performance data. Such an event points to the updated data in the model and thus directly to the associated architecture element. Consequently, the analysis can be performed right away on these elements and the updated data without having to search and check the whole architectural model, which supports an incremental processing.

For instance, if the performance (*i. e.*, the average computation time or fraction of filtered products) of a filter component in running mRUBiS changes, the monitoring operation updates the performance data of this component in the model. Such an update causes a change event that points to this component and that indicates a potential occurrence of PI1. Therefore, based on the change event, this operation directly analyzes this filter component to check for an occurrence of PI1, that is, whether it is properly located in the pipe based on the performance data of the neighboring filters. For PI2 and PI3, the performance of the overall search in mRUBiS is relevant. If it changes, the monitoring operation updates the performance data of the *BrowseCategoriesService* interface provided by the *Item Management Service* component in the model. Such an update causes a change event that points to this interface and that indicates a potential occurrence of PI2 or PI3. Based on the change event, this operation directly analyzes this updated data to check whether the average response time is above or below certain thresholds to identify an occurrence of either PI2 or PI3. Thus, the change events determine which analyses should be performed and they drive the analyses as they provide the locations in the model where the checks should be performed.

If the analysis has identified a PI in the model, it annotates the model with a corresponding PI marker. A PI1 marker annotates a filter component that is not properly located in the pipe and a PI2 or PI3 marker points to the pipe's first component to annotate the pipe.

Moreover, the implementation of this operation is model-driven since we specified the analyses for PIs with SDs. For each of the three PI types, we specified an SD that checks for an occurrence of this PI type and if needed creates a corresponding PI marker. Thus, based on the change events, the implementation invokes the SD interpreter to execute the corresponding SDs. Since these SDs follow the same principle as the SDs for the self-repair (*cf.* Section 9.3.2), we omit showing and discussing them in detail. All of these SDs constitute the evaluation model called Performance analysis rules in the FLD in Figure 29 on Page 72. The result of this operation is the Architectural Model annotated with PI marker elements that represent occurrences of PIs in the running mRUBiS.

Resolve p-issues. For each of the identified PIs, the planning operation decides which Adaptation Strategy (AS) should be applied to address the PI and it determines the required information for each strategy. For each PI type, we developed one strategy.

(AS6) Relocating the filter that is not properly located in the pipe and thus caused PI1. This operation identifies the filter's optimal position in the pipe based on the performance data of all filters. Changing the order of filters is feasible as they are independent of each other. *(AS7)* Shortening the pipe of filters to reduce the computation time of filtering products. A shorter pipe will reduce the response time—to tackle PI2—but also the quality of the personalized search. This operation identifies which filters should be removed from the pipe based on the filters' performance data to achieve the target response time. *(AS8)* Increasing the pipe of filters to improve the quality of the search, which also increases the average response time that is sufficiently low at the moment (PI3). This operation identifies which filters should be added to the pipe based on the filters' past performance data such that the average response time will increase but not exceeds the target response time.

This operation assigns an AS to each of the identified PIs: AS6 to PI1, AS7 to PI2, and AS8 to PI3. Particularly, it creates the corresponding AS markers in the model that point to the individual PI markers and capture the planned information such as the new position of a filter (PI1) or the number of filters to be removed/added to the pipe (PI2/PI3).

Since the planning involves numerical computations on performance data, SDs supporting structural computations are not a good choice for implementing this operation. Thus, we used only code. Consequently, the FLD in Figure 29 on Page 72 shows no change model that specifies and executes the planning and that is consumed by this operation.

Similar to the analysis step, this operation supports an incremental processing since it does not search or navigate through the whole model to assign adaptation strategies. In contrast, the planning starts with the given PI markers that have been added to the model by the analysis operation. The result of the planning operation is the Architectural Model annotated with AS markers. These markers represent the adaptation strategies that are assigned to the occurred PIs and that are executed by the next operation.

Effect. The last operation of the feedback loop executes the assigned adaptation strategies to effect the running mRUBiS. Similar to the self-repair feedback loop (*cf.* Section 9.3.2), the adaptation is first performed on the Architectural Model and then incrementally synchronized to mRUBiS through the causal connection. However, we used only code to implement this operation while we have used SDs for this purpose in the self-repair feedback loop. Thus, the strategies AS6 (changing the connectors among filter components to relocate the misplaced filter), AS7 (disconnect, stop, and undeploy filters to shorten the pipe), and AS8 (deploy, start, and connect filters to increase the pipe) are implemented with code.

Given the AS markers added by the previous operation to the model, this operation directly obtains the related PI markers and the planned information such as the new position of a misplaced filter or the filters to be removed/added to the pipe. For each AS marker, it executes the corresponding AS using the planned information. Meanwhile, it invokes the causal connection with the TGG Rules (*cf.* FLD in Figure 29 on Page 72) to effect mRUBiS.

Being driven by changes of the Architectural Model in terms of the AS markers, this operation supports an incremental processing that avoids searching the whole model. Particularly, it starts executing an adaptation strategy with an AS marker that provides direct access to the affected components and planned information, which are required by the strategy. Moreover, an adaptation at the model level is incrementally propagated to mRUBiS. Finally, having executed a strategy, this operation removes the corresponding AS and PI markers from the model. Afterwards, the feedback loop terminates.

The realization of this feedback loop illustrates how model operations can be implemented and executed with models such as SDs (see analysis operation) or with code (see

planning and execution operations). EUREMA integrates both kinds of operations in a feedback loop. This option allows developers to select an appropriate formalism to individually implement each operation based on the requirements of each operation.

Experiment. We conducted an experiment to show the effective application of the self-optimization feedback loop and EUREMA for maintaining the response time of mRUBiS.

For the experiment, we deployed mRUBiS as shown in Figure 82 on Page 187 on our platform (*cf.* Section 8.1). Five clients are continuously and concurrently performing a personalized search on mRUBiS. Each client pauses for 100 ms between receiving a search result and sending the next search request. To inject a PI in mRUBiS, we add or remove load from the filter components such that the response time of the personalized search increases (PI2) or decreases (PI3). Thus, the experiment focuses on PI2 and PI3 with their corresponding strategies of decreasing (AS7) or increasing (AS8) the size of the pipe.

The EUREMA interpreter executes at second 150 and then approximately every 180 (s) the self-optimization feedback loop that takes into account the average response time of the personalized search measured on the server side over the previous period of 150–180 s. Depending on this average response time, the feedback loop either performs AS7 or AS8.

The result of the experiment is shown in Figure 90. The x-axis depicts the time in seconds (s). The y-axis on left depicts the average response time of the personalized search in milliseconds (ms) for a time frame of 1 s and measured by the clients.[10] The y-axis on the right depicts the number of filters that are running in mRUBiS. A change of this number is caused by a self-adaptation that adds or removes filters from the pipe. Overall, ten filters exist. We set the target for the average response time to 1000 ms. A self-adaptation should happen if the average response time exceeds 1200 ms (PI2) or falls below 950 ms (PI3).

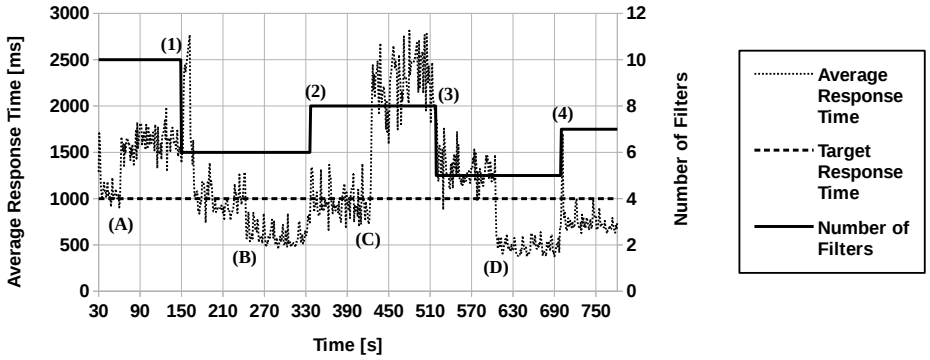


Figure 90: Experiment Results for Self-Optimizing mRUBiS.

The experiment consists of the following steps that are labeled with letters (add or remove load) or numbers (self-adaptation) in Figure 90. The first 30 s are the warm-up for the clients and mRUBiS such that both are initialized (not shown in the figure). During the next 30 s, the average response time is slightly above 1000 ms while ten filter components are running. At second 60 (A), we add load on the filter components and the average response time exceeds 1500 ms. Thus, the self-adaptation performed by the feedback loop at second 150 removes four filter components from mRUBiS (1). As a consequence, the average response time drops to around 1000 ms, which is the target. At second 230, we remove load from the filters (B) and the average response time drops to around 600 ms. Consequently, the self-adaptation at second 330 adds two filters (2), which increases the average response

10 The time when a client receives the search results determines the time frame to which the request is assigned.

time to around 1000 ms. At second 420, we add load on the filters (C) and the average response time exceeds 2000 ms until the self-adaptation at around second 510 removes three filters (3). As a consequence, the average response time drops to around 1250 ms, which approaches but does not meet the target. At second 600, we remove load from the filters (D) and the average response time drops to around 500 ms. Thus, the self-adaptation at around second 690 adds two filters (4), which increase the average response time to around 750 ms.

The result of the experiment visualized in Figure 90 shows that the EUREMA-based self-optimization feedback loop is able to effectively maintain the average response time of the personalized search in mRUBiS around the target of 1000 ms by adapting the pipe of filter components. Every time a PI occurred, the self-optimization feedback loop takes countermeasures such that the average response time approaches the target.

We repeated this experiment five times and obtained consistent results between the different runs except of one situation. In this situation, the feedback loop did not adapt mRUBiS because the peak of the response time caused by the injected load was not high enough to cause an adaptation and it dissolved when the load subsequently decreased (*i. e.*, when load was removed again from mRUBiS). A self-adaptation did not occur due to the combination of the period for executing the feedback loop, the interval for monitoring the average response time, and the thresholds. However, this outcome corresponds to the specified behavior of the feedback loop that would address the peak if the peak remains in mRUBiS. A detailed discussion of this situation is given in Section C.2.

Finally, it has to be noted that the goal of this experiment is to demonstrate the general ability of an EUREMA-based feedback loop to perform self-adaptation. Consequently, we did not tune the feedback loop parameters such as the period, the monitoring interval, or the thresholds and sensitivity of triggering an adaptation to improve the outcome of the self-optimization. Moreover, the need of tuning such parameters is not specific to EUREMA and applies to all approaches. Therefore, the experiment still gives us some evidence that the self-optimization feedback loop developed and executed with EUREMA is able to maintain the average response time of the search in mRUBiS. Moreover, it demonstrates the application of EUREMA to a self-adaptation problem and a running system.

9.3.4 *Coordinating the Self-Repair and Self-Optimization of mRUBiS*

Besides evaluating separately the EUREMA-based feedback loops for the self-repair and self-optimization, we evaluate as well the coordination of both feedback loops for self-managing mRUBiS. The self-management maintains the availability and the response time of mRUBiS at the same time. In Section 5.2, we modeled two design alternatives for coordinating interdependent feedback loops with EUREMA, which sequence either the complete feedback loops or the individual activities of the feedback loops.

To evaluate both alternatives, we conducted an experiment for each of them. For these experiments, we deployed mRUBiS as shown in Figure 82 on Page 187 on our platform (*cf.* Section 8.1). Ten clients are continuously and concurrently requesting services from mRUBiS. Half of them is performing a personalized search on mRUBiS while pausing for 250 ms between receiving a search result and sending the next search request. The other half is requesting the availability of products from the *Bid and Buy Service* component that requires the *Inventory Service* component while pausing for 25 ms between receiving a reply and sending the next request. The smaller pause is motivated by the fact that requesting the product availability is less time-consuming than the personalized search.

Similar to the previous experiments, to inject a PI in mRUBiS, we add or remove load from the filter components such that the response time of the personalized search increases (PI2) or decreases (PI3). The self-optimization loop should consequently either decrease (AS7) or increase (AS8) the size of the pipe of filters. To inject a CF, we remove the *Inventory Service* component from mRUBiS (CF3) such that requests for the product availability cannot be served until the self-repair feedback loop redeploys this component (AS3).

Using this mRUBiS and client configuration, we evaluate both coordination alternatives. The implementations of these two alternatives reuse the same implementations for the individual model operations of the self-repair and self-optimization feedback loops discussed in Sections 9.3.2 and 9.3.3. However, instead of directly executing instances of the FLDs specifying the feedback loops, instances of the FLDs specifying the coordination of both feedback loops are executed. In the following, we evaluate each coordination alternative.

Sequencing Complete Feedback Loops

The sequencing of both feedback loops is modeled with the FLD in Figure 31 on Page 76, which defines that the self-repair is executed before the self-optimization feedback loop. Thus, Critical Failures (CFs) are repaired before Performance Issues (PIs) are addressed.

Experiment. The result of the experiment is shown in Figure 91. The x-axis depicts the time in seconds (s). The y-axis on left depicts the average response time of the personalized search in milliseconds (ms) for a time frame of 5 s and measured by the clients. The y-axis on the right depicts the success rate of the client requests (*i. e.*, the number of proper replies from mRUBiS divided by the number of all requests) for a time frame of 5 s. The success rate is used as a metric for the availability as a rate of 1 (0) denotes that each (no) request is successfully served. We set the target for the availability to 1 and for the average response time to 1000 ms. A self-adaptation should happen if the availability drops (due to a CF3) or if the average response time exceeds 1200 ms (PI2) or falls below 950 ms (PI3). Both feedback loops are executed in a coordinated manner, that is, by executing an instance of the FLD shown in Figure 31 on Page 76, roughly every 180 s immediately after we injected a CF3. Hence, the individual feedback loops are executed one after the other.

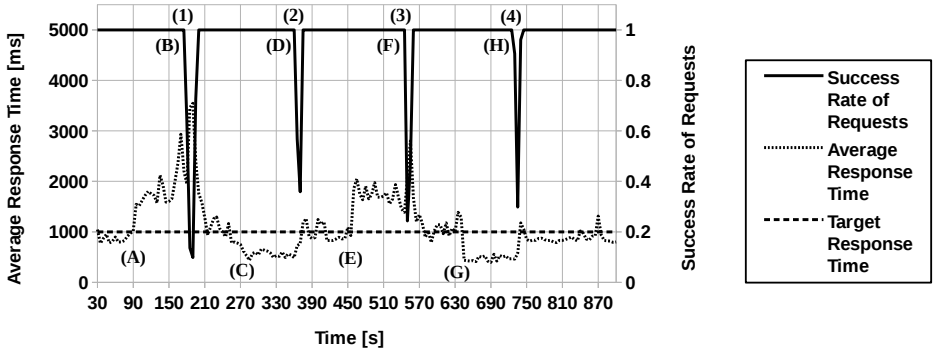


Figure 91: Experiment Results for Sequencing Feedback Loops.

The experiment consists of the following steps labeled with letters (add/remove load or inject a CF3) or numbers (self-adaptation) in Figure 91. The first 30 s are the warm-up phase to initialize the clients and mRUBiS (not shown in the figure). During the next 60 s, the average response time is slightly below the target of 1000 ms and ten filters are running.

At second 90 (A), we add load to the filters to cause an increase of the response time and thus a PI2. At second 180 (B), we inject a CF3 to cause a drop of the availability. The feedback loops are now executed sequentially (1). The self-repair loop repairs CF3 by a redeployment and then the self-optimization loop addresses PI2 by removing four filters. Thus, the availability increases to 1 and the response time decreases toward the target.

At second 270 (C), we remove load from the filters to cause a decrease of the response time toward 500 ms and thus a PI3. At second 360 (D), we inject a CF3 to cause a drop of the availability. Again, the self-repair feedback loop is executed to repair CF3 and the self-optimization feedback loop to add four filters, which addresses PI3 (2). Consequently, the availability increases to 1 and the average response time to around the target of 1000 ms.

At second 450 (E), we add load to the filters to cause an increase of the response time toward 2000 ms that results in a PI2. At second 540 (F), we inject a CF3 to cause a drop of the availability. Again, the self-repair feedback loop is executed to repair CF3 and the self-optimization feedback loop to address PI3 by removing four filters (3). Consequently, the availability increases to 1 and the average response time to around the target of 1000 ms.

At second 630 (G), we remove load from the filters to cause a decrease of the response time and thus a PI3. At second 720 (H), we inject a CF3 to cause a drop of the availability. Again, the self-repair feedback loop is executed to repair CF3 and afterwards the self-optimization feedback loop to address PI3 by adding four filters (4). Consequently, the availability increases to 1 and the average response time closely to the target of 1000 ms.

This experiment investigates how the coordinated feedback loops handle the two scenarios when PI2 and CF3 or PI3 and CF3 occur at the same time. The former is addressed by the (1). and (3). run and the latter scenario by the (2). and (4). run of the feedback loops.

The result of the experiment visualized by Figure 91 demonstrate that the EUREMA-based coordination of the self-repair and self-optimization feedback loops is able to effectively maintain the availability and the average response time of mRUBiS. Every time a PI and a CF occur at the same time, the coordinated execution of both feedback loops takes countermeasures by first repairing the CF and directly afterwards addressing the PI. After a coordinated execution of both feedback loops, the availability and the average response time of mRUBiS achieve their targets of 1 respectively, 1000 ms.

We repeated this experiment five times and obtained consistent results between the different runs. The results of the other four runs are shown in Section C.3.1. Thus, the experiment gives us some evidence that the coordination of sequencing the self-repair and self-optimization feedback loops developed and executed with EUREMA is able to simultaneously maintain the availability and the average response time of mRUBiS. Moreover, it demonstrates the successful application of EUREMA to a self-adaptation problem with two feedback loops and a running software system.

Sequencing Adaptation Activities of Feedback Loops

The sequencing of the individual adaptation activities of both feedback loops is modeled with the FLD in Figure 33 on Page 79, which defines that after the common monitoring activity, the analysis and planning for the self-repair is executed, then the analysis and planning for the self-optimization, and finally the common execute activity.

This coordination design requires achieving a consensus such that the individual planning activities do not project competing adaptations. In this case, we implemented a basic consensus that gives the self-repair a higher priority than the self-optimization. If the self-repair's planning activity has projected an adaptation to repair a CF, then the self-optimization's planning activity does not project any further adaptation to address a PI,

which might otherwise contradict the already projected adaptation. Thus, if a CF and a PI occur at the same time, the coordination will plan and execute an adaptation to immediately address the CF and it will postpone the adaptation to address the PI to the next run of both feedback loops unless another CF has occurred in the meantime.

Experiment. The experiment results are shown in Figure 92. The x-axis depicts the time in seconds (s). The y-axis on left depicts the average response time of the personalized search in milliseconds (ms) for a time frame of 5 s and measured by the clients. The y-axis on the right depicts the success rate of the client requests for a time frame of 5 s. The success rate is used as a metric for the availability as a rate of 1 (0) denotes that each (no) request is successfully served. Again, we set the target for the availability to 1 and for the average response time to 1000 ms. A self-adaptation should happen if the availability drops (due to a CF3) or if the average response time exceeds 1200 ms (PI2) or falls below 950 ms (PI3). Both feedback loops are coordinated by executing an instance of the FLD shown in Figure 33 on Page 79, roughly every 180 s and immediately after we injected a CF3.

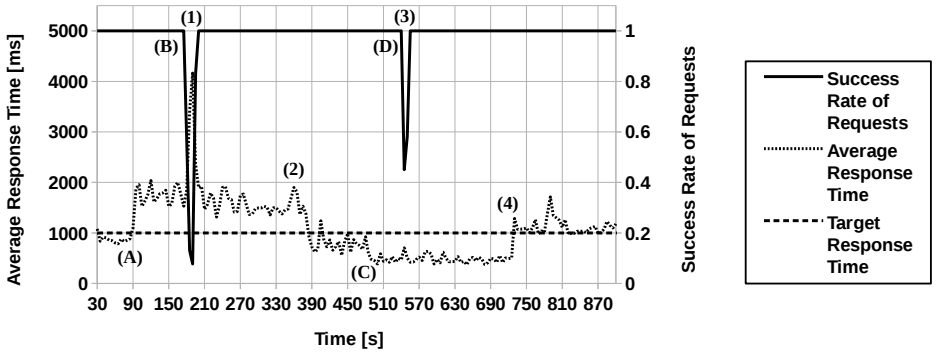


Figure 92: Experiment Results for Sequencing Activities of Feedback Loops.

The experiment consists of the following steps labeled with letters (add/remove load or inject a CF3) or numbers (self-adaptation) in Figure 92. The first 30 s are the warm-up phase to initiate the clients and mRUBiS (not shown in the figure). During the next 60 s, the average response time is slightly below the target of 1000 ms and ten filters are running.

At second 90 (A), we add load to the filters to cause an increase of the response time over 1500 ms and thus a PI2. At second 180 (B), we inject a CF3 to cause a drop of the availability. Then (1), we execute the coordinated feedback loops that repair CF3 by a redeployment but do not address PI2. Hence, the availability increases to 1 while the average response time remains over 1500 ms. At second 360 (2), the coordinated feedback loops are executed again. Since no CF has occurred in the meantime, they now address PI2 by removing six filters. Consequently, the average response time decreases below the target of 1000 ms.

At second 480 (C), we remove load from the filters to cause a decrease of the response time toward 500 ms and thus a PI3. At second 540 (D), we inject a CF3 to cause a drop of the availability. Then (3), we execute the coordinated feedback loops that repair CF3 but they do not address PI3. Hence, the availability increases to 1 while the average response time remains around 500 ms. At second 720 (4), the coordinated feedback loops are executed again. Since no CF has occurred, they now address PI3 by adding six filters, which causes an increase of the average response time to slightly over the target of 1000 ms.

The experiment investigates how the coordinated feedback loops behave when PI2 and CF3 or PI3 and CF3 occur at the same time. In both cases, the feedback loops address CF3

but postpone PI2 respectively PI3 in their (1). respectively (3). execution. The postponed PI2 and PI3 are then addressed by respectively subsequent executions (2) and (4).

The result of the experiment visualized by Figure 92 shows that the EUREMA-based coordination of the self-repair and self-optimization feedback loops is able to effectively maintain the availability and the average response time of mRUBiS. Every time a PI and a CF occur at the same time, the coordinated execution of both feedback loops takes countermeasures by repairing the CF while addressing the PI is postponed to the next run of both feedback loops (this coordination has therefore the drawback that it will never address a PI if CFs are permanently occurring in mRUBiS). Thus, after two runs of the coordinated feedback loops, the CF and PI are addressed such that the availability and the average response time of mRUBiS achieve their targets of 1 respectively, 1000 ms.

We repeated this experiment five times and obtained consistent results between the different runs. The results of the other four runs are shown in Section C.3.2. Thus, the experiment gives us some evidence that the coordination of sequencing the individual activities of the self-repair and self-optimization feedback loops, which we developed and executed with EUREMA, is able to manage mRUBiS. It further demonstrates the successful application of EUREMA to a self-adaptation problem with two feedback loops and a running system.

9.3.5 *Three-Layer Architecture for Self-Repairing mRUBiS*

In Section 5.3, we discussed the modeling of layered feedback loops and the example that a higher-layer feedback loop manages the repair strategies used by the underlying self-repair feedback loop. If the self-repair loop does not succeed in repairing the failures, it will be equipped with new strategies from the higher-layer loop to tackle the untreated failures. The new strategies are selected from a repertoire of strategies. The selection happens either randomly or in a predefined manner by mapping replacing to replaced strategies.

We have modeled the self-repair feedback loop with the FLDs in Figure 19 on Page 65 and Figure 21 on Page 66 as well as the higher-layer feedback loop with the FLD in Figure 36 on Page 83. Applying these two feedback loops on top of mRUBiS results in a three-layer architecture that is described in the LD in Figure 37 on Page 84.

The implementation of the self-repair feedback loop is the same as discussed in Section 9.3.2.[11] The higher-layer feedback loop consists of two model operations (*cf.* Figure 36 on Page 83) that are implemented as follows. First, the Check strategies operation uses Strategy analysis rules expressed with SDs to identify and annotate the repair strategies that are currently used in the self-repair loop. Second, the Select strategies operation uses Strategy selection rules expressed with SDs to select repair strategies from the Strategy repertoire. Afterwards, it replaces the annotated repair strategies with these selected strategies. For the experiment, the selection of the strategies is predefined.

According to the LD (see Figure 37 on Page 84), the higher-layer loop is executed after the self-repair loop has executed its operation Deep check for failures. This happens when the last three consecutive runs and analyses of the self-repair loop have identified failures, which indicates that failures are persisting and calls for testing new repair strategies.

Experiment. To evaluate this three-layer architecture, we conducted an experiment on top of the running mRUBiS. We deployed mRUBiS as shown in Figure 82 on Page 187 on our

11 Although we use now the variant of the self-repair feedback loop that splits up the analysis into two model operations, the whole analysis is conducted by the operation called Check for failures while the other analyzing operation called Deep check for failures does not perform any behavior (dummy operation). Thus, the implementation of the Check for failures operation is the same as described in Section 9.3.2.

platform (*cf.* Section 8.1). However, this deployment includes a faulty *Authentication Service* component that causes failures 80 s after it starts serving requests until it crashes roughly 40 s later (CF3). To tackle this occurrence of CF3, the self-repair feedback loop redeploys the faulty component (AS3). Since the fault is still present, the failure occurs all over again and it is identified by three consecutive runs of the self-repair feedback loop. This causes the higher-layer feedback loop to execute and to replace the repair strategies, particularly AS3 with AS4. Thus, the self-repair feedback loop will not redeploy (AS3) but replace the faulty component with a component of a different type that provides the same functionality (*e. g.*, authenticating users with third-party, social media services) (AS4).

For this experiment, 20 clients are continuously and concurrently sending requests to mRUBiS, particularly to the *User Management Service* component. Each client pauses for 250 ms between receiving a reply for a request and sending the next request. Half of the clients request the *ViewUserInfoService* and the other half the *AboutMeService* interface. While the former requires the *AuthenticationService* component, the latter does not.

The result of the experiment is depicted in Figure 93. The x-axis depicts the time in seconds (s) while the y-axis depicts the success rate of the client requests (*i. e.*, the number of proper replies from mRUBiS divided by the number of all requests) for a time frame of 1 s.[12] The success rate is used as a metric for the availability. A rate of 1 (0) denotes that each (no) request is successfully served. The goal is to achieve an availability of 1.

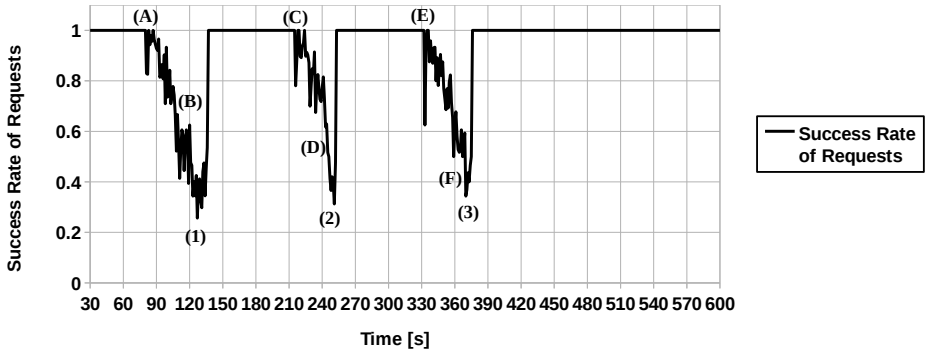


Figure 93: Experiment Results for the Self-Repairing mRUBiS in a Three-Layer Architecture.

The experiment consists of the following steps labeled either with letters (occurrence of failures) or numbers (self-adaptation) in Figure 93. The first 30 s are the warm-up phase to initialize the clients and mRUBiS (not shown in the figure). During the subsequent 50 s, mRUBiS serves all requests properly. At second 80 (A), the authentication component starts causing more and more failures such that the availability drops toward 0.4. At least half of the client requests cannot be served by mRUBiS when the component eventually crashes at around second 120 (B). This CF3 triggers the self-repair feedback loop that redeploys the faulty component (1) such that the availability increases to 1.

However, roughly 80 s after the redeployment (C), the faulty component causes again failures and eventually crashes at around second 250 (D). The self-repair loop again redeploys the faulty component (2), which recovers mRUBiS until the faulty component again causes failures at second 330 (E) and crashes for the third time at second 370 (F).

The self-repair loop is executed again (1). Its analysis identifies this crash. Since the last three analyses have identified such a crash, the higher-layer feedback loop is triggered to replace the AS3 with AS4. The self-repair feedback loop's planning continues execution

12 The time when a client receives a reply or exception determines the time frame to which the request is assigned.

with the new strategies and instead of redeploying, it replaces the faulty component with an alternative component. This replacement recovers mRUBiS such that the availability increases to 1. Moreover, it remains at 1 since the replacing component is not faulty.

The result of the experiment (*cf.* Figure 93) shows that the EUREMA-based three-layer architecture is able to effectively maintain the availability of mRUBiS by dynamically changing the adaptation logic of the self-repair mechanism. If CFs continuously occur, the layered feedback loops take countermeasures by changing the adaptation strategies to tackle persisting failures such that mRUBiS recovers and reaches the target availability. For this experiment, we assume that the replacing strategies are able to cope with the persisting failures. In this context, more elaborated means to dynamically change the adaptation logic are conceivable, for instance, to synthesize novel repair strategies. Such means can be integrated into EUREMA feedback loops when implementing the individual model operations. However, the focus is here on demonstrating the general applicability and effectiveness of EUREMA for developing and executing layered feedback loops and not on evaluating specific analysis and planning techniques for higher-layer feedback loops.

We repeated this experiment five times and obtained consistent results between the different runs. The results of the other four runs are shown in Section C.4. Therefore, the experiment gives us some evidence that the three-layer architecture for self-repairing mRUBiS, which we developed and executed with EUREMA, is able to tackle failures in mRUBiS by using dynamically selected adaptation strategies. Such a dynamic selection is more flexible and powerful than a static selection made at design time. Therefore, the experiments further shows the flexibility of feedback loops that can be achieved with EUREMA. Finally, it demonstrates the successful application of EUREMA to a self-adaptation problem with two layered feedback loops and a running software system.

9.3.6 *Discussion*

In this section, we have demonstrated the effective use of EUREMA to develop and execute feedback loops that manage a running software system. For this purpose, we have conducted experiments with the running mRUBiS to show that the EUREMA-based feedback loops are able to maintain the availability and performance of mRUBiS. To comprehensively evaluate the capabilities of EUREMA, we considered four variants of the adaptation engine that equips mRUBiS either with a self-repair (*cf.* Section 9.3.2), a self-optimization (*cf.* Section 9.3.3), two coordinated self-repair and self-optimization (*cf.* Section 9.3.4), and two layered self-repair (*cf.* Section 9.3.5) feedback loops. Therefore, the focus of the experiments was on evaluating a spectrum of feedback loop solutions developed and executed with EUREMA and not on developing and evaluating specific analysis and planning mechanisms for these feedback loops. Consequently, the feedback loops we developed employ basic rule-based mechanisms for their analysis and planning activities.

The results of the experiments for all variants of the adaptation engine give us some evidence that EUREMA can be effectively used to develop and execute feedback loops that dynamically adapt a running software system and particularly mRUBiS with the goal of maintaining availability and performance. The experiments and their results further demonstrate the capabilities of EUREMA to realize a variety of solutions considering single, multiple coordinated, and layered feedback loops. Despite this evidence, we cannot conclude that EUREMA can be effectively used for *any* self-adaptation problem and adaptable software system. In this context, more experiments are required to tackle other problems (*e. g.*, self-protection) and systems (*e. g.*, running on a service or cloud platform).

9.4 COMPARATIVE STUDY

To further evaluate EUREMA, we conducted a comparative study, in which we investigate the development costs and runtime performance of feedback loops. For this purpose, we compare three solutions to the self-repair and self-optimization feedback loops that handle Critical Failures (CFs) and Performance Issues (PIs) in mRUBiS (*cf.* Sections 9.3.2 and 9.3.3 for a discussion of the potential CFs and PIs). These solutions have been developed and the study has been conducted using our implementation framework (*cf.* Chapter 8).

For this purpose, we customized the framework as shown in Figure 94. Thus, the different solutions operate on a reflection model, particularly on a CompArch model that describes the runtime architecture of mRUBiS. This model is maintained by the simulator that emulates the running mRUBiS and the causal connection. We discussed the CompArch language and the simulator in Sections 8.2 and 8.4. Consequently, the solutions are not concerned with connecting and synchronizing the model and the running mRUBiS.

To test the solutions, we customize the simulator with injectors that introduce CFs or PIs into the reflection model, analyzers that check the model after an adaptation (*e. g.*, whether the feedback loop of a solution has addressed the injected CFs or PIs in the model), and a simulation strategy that determines how many and which types of CF or PI should be injected to which element/component of the mRUBiS architecture. Therefore, one test run consists of three steps:(1) the simulator injects CFs or PIs into the reflection model, (2) a feedback loop solution is executed to handle the injected CFs or PIs by adapting the model, (3) the simulator analyzes the model and the adaptation of the model.

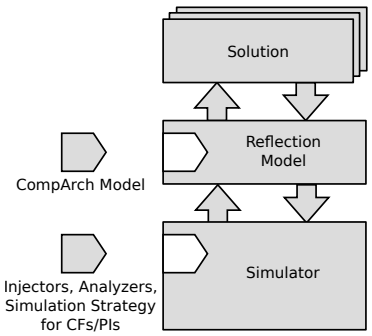


Figure 94: Customized Implementation Framework for the Study.

Each solution that we compare in the study realizes two feedback loops, a self-repair and a self-optimization loop, that follow the MAPE-K blueprint and therefore consist of four adaptation activities. However, the solutions differ as they are realized either with or without EUREMA and their activities are positioned differently within the solution space of model operations (*cf.* Chapter 7). Thus, the activities are either code-based or model-driven and they operate in a state- or event-based manner. The three solutions are as follows:

State-based Solution. The feedback loops of this solution are developed with plain Java (and not with EUREMA). The activities of these feedback loops are code-based, that is, they are implemented and executed with Java (and not with models such as SDs), and they work in a state-based manner on the reflection model capturing the shared knowledge.

Thus, the monitoring activity compares the reflection model with a reference model to identify whether changes of the reflection model have occurred, due to the injections by the simulator, that require analysis. The analysis checks the whole reflection model for any

occurrences of CFs or PIs and annotates them in the model. The planning activity searches for these CF and PI annotations in the model. For each of them, it changes the model according to a predefined adaptation strategy to address the corresponding CF or PI. The execute activity removes the annotations from the model and finishes the feedback loop.

Event-based Solution. Similar to the state-based solution, the feedback loops of this solution are developed with Java and the activities are code-based. However, the activities primarily work in an event-based manner, that is, they exchange specific events to share knowledge and to perform self-adaptation. These events are encoded in the Extensible Markup Language (XML) as used in autonomic computing for this purpose [235, 287].

Thus, the monitoring activity consumes EMF events notifying about changes of the reflection model caused by the simulator. It encodes such events in XML and sends them to the analyze activity. Using the knowledge captured in the events, the analysis checks for CFs or PIs and if needed, sends a change request to the next activity. Based on this request, the planning activity selects a predefined adaptation strategy and sends it to the next activity. The execute activity receives and enacts the strategy by changing accordingly the reflection model, which terminates one run of a feedback loop. Thus, the analysis and planning activities do not work on the reflection model but thy only rely on these events. If they require more knowledge, they request it from the monitoring activity using events.

EUREMA Solution. The feedback loops of this solution are developed with EUREMA and they have been discussed in detail in Sections 9.3.2 and 9.3.3. For the self-repair, it employs operations that are model-driven and particularly implemented and executed by SDs. For the self-optimization, it employs model-driven and code-based operations. Moreover, both feedback loops are event-/state-based, that is, all of their operations work on the reflection model and the processing of these operations is driven by change events that refer to this model. Consequently, the model operations process the reflection model incrementally.

The monitoring operation consumes the EMF events notifying about changes of the reflection model caused by the simulator to identify whether any change has occurred. If so, the analysis operation processes these events to locate changes in the model, which are the starting points for identifying and annotating CFs or PIs in the model. The planning operation uses these annotations to locate the identified CFs and PIs and to assign adaptation strategies to them by creating further annotations. The execute activity uses these annotations to enact the assigned strategies by changing the model according to these strategies.

An overview of the three solutions is given in Table 7. This overview locates each solution in the solution space for adaptation activities that we discussed in Chapter 7. This solution space distinguishes between activities that are realized and executed either by code or models and that operate either in a state-, event-, or combined state- and event-based manner. The state- and event-based solutions adopt a typical approach with code-based activities operating either in a state- or an event-based manner but not in a combined fashion. In contrast, with EUREMA we primarily promote activities that are driven by models and that combine a state- and event-based processing of the knowledge.

Table 7: Locating the Three Solutions in the Space of Solutions.

	State	Events	State & Events
Code	State-based Solution	Event-based Solution	
Models			EUREMA Solution

While we developed the EUREMA solution, students participating in our master's course on Software Engineering for Self-Adaptive Systems have implemented the *State-based* and the *Event-based Solutions*.[13] Based on a specification of the CFs, PIs, adaptation strategies, and how state-based or event-based operations generally work, the students developed and tested the individual feedback loops with the simulator. Each solution has been developed by a group of two to three students. In the following, we compare the three different solutions with respect to their development costs and runtime performance.

9.4.1 *Development Costs*

To discuss the development costs for each solution and feedback loop, we determined the size of each solution in terms of Source Lines of Code (SLOC)[14]. Due to issues of counting and interpreting SLOC [237, 364], we use SLOC only as a rough indicator for the development costs. Additionally, we determined the size of the models, particularly of the SDs that are part of the implementation of the EUREMA solution.

Self-Repair Feedback Loop. Table 8 shows the numbers of SLOC for each solution and the numbers of the models for the EUREMA solution. The numbers are broken down to each MAPE activity and to the *Integration* of these activities to form a feedback loop.

For the integration, the state-based solution required only 82 SLOC as it just invokes the individual MAPE activities sequentially and provides access to the reflection model. In contrast, the integration in the the event-based solution consists of 746 SLOC as it implements the required events that are exchanged between activities to share knowledge. Here, EUREMA comes into play as it realizes the integration of activities to a feedback loop only by models. In this case, one FLD and one LD were required to specify the feedback loop.

Concerning the MAPE activities, the state- and event-based solutions are similar with respect to the monitoring and analysis but different for the planning and execution steps. The event-based solution requires 444 SLOC for the execute as it has to translate events to actions for adapting the reflection model. Such a translation step is not needed by the state-based and EUREMA solutions. Moreover, the event-based solution enacts an adaptation by changing the reflection model in the execution phase, similar to the EUREMA solution, while the state-based solution adapts the model in the planning phase. Thus, the state-based solution shifts tasks and efforts from the execution to the planning step.

The EUREMA solution typically requires the fewest SLOC for the MAPE activities since it additionally uses 30 SDs. The code for the analysis, planning, and execution steps (101, 64, and 73 SLOC) only determines the SDs to be executed based on change events and annotations in the model and it just invokes the SD interpreter to execute these SDs.

To investigate the costs of developing the 30 SDs, we calculated the average number of SPs per SD (each SP specifies a graph transformation) as well as the average number of vertices per SP. These numbers indicate the efforts of modeling an SD since they show how many graph transformations are specified on average in an SD as well as the average size of each graph transformation specification.

13 Overall, we obtained four different solutions from the students, two of them are state-based and the other two are event-based. However, we dismissed two solutions for the comparative study: One state-based solution shifted specific analyses to the monitoring phase, which impedes a comparison of the individual activities. One event-based solution interweaves the implementations of the self-repair and self-optimization feedback loops, which impedes a comparison of the individual feedback loops. Thus, two solutions remain for the study.

14 We counted every line of the source code excluding comments and empty lines after unifying the formatting of the code.

Table 8: Development Costs for the Self-Repair Feedback Loops.

Solution		*Source Lines of Code (SLOC)* Integration	Monitor	Analyze	Plan	Execute	Sum
State-based		82	115	167	236	23	623
Event-based		746	152	207	149	444	1698
EUREMA		0	25	101	64	73	263
		Models					
	Number	1 FLD, 1 LD	–	7 SDs	11 SDs	12 SDs	30 SDs
	SPs/SD		–	2.85	1.00	3.17	2.30
	Vertices/SP		–	2.75	3.55	2.18	2.57

On average an SD consists of 2.30 SPs and each SP specifies a graph transformation with on average 2.57 vertices (*cf.* Table 8). Thus, the individual SDs and SPs are not complex while the high number of SDs is mainly motivated by a modular specification to enable reuse of SDs in other SDs.

For the monitoring, the EUREMA solution requires very few SLOC (25). It just checks for occurrences of events that notify about changes of the reflection model and that are eventually processed by the analysis activity. Such events are provided by EMF and they point to elements in the model that have changed, in this case because of the simulator that injects CFs into the model. In contrast, the monitoring of the event-based solution requires more SLOC (152) since besides consuming such change events, it has to identify and encode all relevant knowledge about the changes into self-contained XML-based events to be processed by the analysis. The monitoring of the state-based solution requires also more SLOC (115) than the EUREMA solution since it compares the reflection model to a reference model to identify any deviation from the reference to trigger the analysis.

Overall, the state-based solution to the self-repair feedback loop consists of 623 SLOC while the event-based solution required more SLOC (1698) due to larger efforts for the integration of the MAPE activities by events and the translation of events to actions performed on the reflection model. The EUREMA solution required the fewest SLOC (263) because of the integration based on the FLD and LD and the use of SDs. Thus, EUREMA requires less coding but shifts efforts to modeling. The development costs of the EUREMA solution seems to be higher than for the state-based solution that, however, does not make use of any change events to improve efficiency of execution. In contrast, the event-based solution uses change events and its development costs seem to be similar to the EUREMA solution.

Self-Optimization Feedback Loop. As before for the self-repair feedback loop, Table 9 lists the numbers for SLOC and models of the three solutions for the self-optimization feedback loop. The results are similar to those for the self-repair feedback loop. Again, the EUREMA solution requires the fewest number of SLOC (675), however, the difference to the other solutions (761 and 1229 SLOC) is smaller than before since it does not use any SD for the planning and execution activities. Only the analyze activity is implemented with three SDs, each of which having on average 2.67 SPs. Each of these SPs specifies a graph transformation with on average 6.63 vertices. While the SPs are more complex than those for the self-repair feedback loop (6.63 vs. 2.57), the total number of SDs is smaller (3 vs. 30) and the average number of SPs per SD is similar (2.67 vs. 2.30).

Table 9: Development Costs for the Self-Optimization Feedback Loops.

Solution		Integration	Monitor	Analyze	Plan	Execute	Sum
		Source Lines of Code (SLOC)					
State-based		81	174	200	294	12	761
Event-based		698	319	96	268	148	1229
EUREMA		0	25	123	330	197	675
		Models					
	Number	1 FLD, 1 LD	–	3 SDs	–	–	3 SDs
	SPs/SD		–	2.67	–	–	2.67
	Vertices/SP		–	6.63	–	–	6.63

As shown in Table 9, the development costs of the EUREMA solution (675 SLOC and 3 SDs) are similar to the state-based solution (761 SLOC) although the former makes use of change events to improve efficiency while the latter does not. The event-based solution is most costly (1229 SLOC), mainly because of its event-based integration of activities to a feedback loop (698 SLOC), which requires implementations of events for the knowledge exchange. In contrast, the state-based solution only requires 81 SLOC to integrate the activities in a control flow and provide access to the reflection model. For the integration, the EUREMA solution does not require any code but it uses one FLD and one LD.

The reason for the differences of the SLOC for the execution activity of the state-based and event-based solutions (12 vs. 148 SLOC) is the same as for the self-repair feedback loop. The state-based solution adapts the reflection model in the planning phase while the event-based solution does it in the execution phase. Thus, the state-based solution shifts tasks and therefore development efforts from the execution to the planning step.

The high numbers of SLOC for the planning and execution steps in the EUREMA solution (330 and 197 SLOC) with respect to the other solutions (294/268 and 12/148 SLOC) is caused by a modular implementation. A distinct class is used to implement the planning as well as the execution of *each* of the three adaptation strategies for self-optimization (AS6, AS7, and AS8). In contrast, the complete logic of each activity of the state-based and event-based solutions is implemented in a single class. The monitoring with only 25 SLOC in the EUREMA solution is the same as for the self-repair feedback loop (*cf.* previous discussion).

Summary. The development costs of the EUREMA solutions to the self-repair and self-optimization feedback loops are similar to the costs of the state-based solutions and lower than the costs of the event-based solutions. This observation roughly indicates the efforts of using EUREMA compared to alternative code-based approaches to develop feedback loops. It gives us some initial evidence that using EUREMA does not make a solution more complex in terms of size but can rather result in a less complex solution. However, this observation only holds for the given study comparing these three solutions for the self-repair and self-optimization feedback loops and therefore cannot be generalized.

Nevertheless, we observe the integration benefits of EUREMA that allows developers to compose adaptation activities to a feedback loop in a domain-specific model (FLD and LD) without the need of any coding.

9.4.2 *Runtime Performance*

To investigate the runtime performance of the different solutions, we run each solution and measured the execution times. For each solution, we individually run the self-repair and self-optimization feedback loop for different sizes of the reflection model. Consequently, we can independently investigate the runtime performance and the scalability of both feedback loops for increasing sizes of the reflection model.

To motivate the scalability, we interpret mRUBiS as a marketplace that hosts multiple shops and we consider each shop as a tenant (*cf.* multi-tenancy). The architecture of one shop is depicted in Figure 82 on Page 187 and consists of 18 components. For each tenant on the marketplace, such an architecture is used. The CompArch language we use to express the reflection model allows us to describe multiple tenant architectures with multiple components in one model (*cf.* Section 8.2). Thus, for each additional tenant on the marketplace, the reflection model contains additional 18 components. Consequently, the feedback loops manage now a larger system and therefore operate on a larger reflection model.

To measure the execution times, we execute the solutions on top of the simulator (*cf.* Figure 94 on Page 204) and consider the cycle of

(1) injecting CFs or PIs into the reflection model,
(2) executing the solution to handle the CFs or PIs in the model, and
(3) analyzing the model after the adaptation.

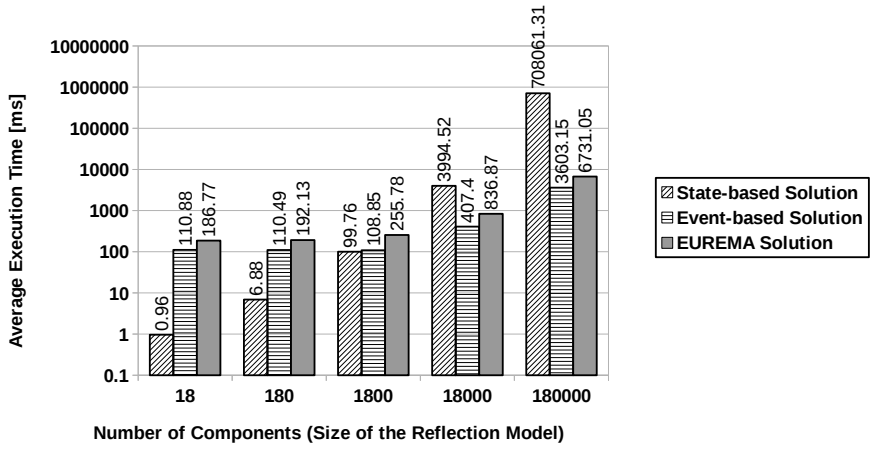
The solutions perform step (2) while the simulator performs the steps (1) and (3) such that we measure only the execution times for step (2). Step (1) is defined by a simulation strategy that determines for each cycle how many and which CFs or PIs are injected as well as the target of the injection. For both feedback loops, we inject exactly one CF or PI in each cycle and the target of an injection is precisely defined. For the self-repair feedback loop, we inject in strict rotation a CF1, CF2, and CF3, and then start again with a CF1. For the self-optimization feedback loop, we inject alternately a PI2 and a PI3. Thus, the simulation is deterministically defined and can be repeated for the different runs of the measurements. The analysis performed in step (3) provides feedback whether the solutions actually addressed the injected CFs or PIs, which was the case for all solutions.

For one measurement run, we execute one feedback loop (self-repair or self-optimization) of one solution (state-based, event-based, or EUREMA) for one size of the reflection model (1, 10, 100, 1000, or 10000 tenants, that is, 18, 180, 1800, 18000, or 180000 components). Within such a run, we repeat the cycle of steps (1), (2), and (3) at least twelve times and up to 250000 times depending on the scale of the execution times and until the measurements stabilized (*i. e.*, the standard deviation of the execution time is less than five percent of the measured average execution time). The measurements were performed on one machine[15].

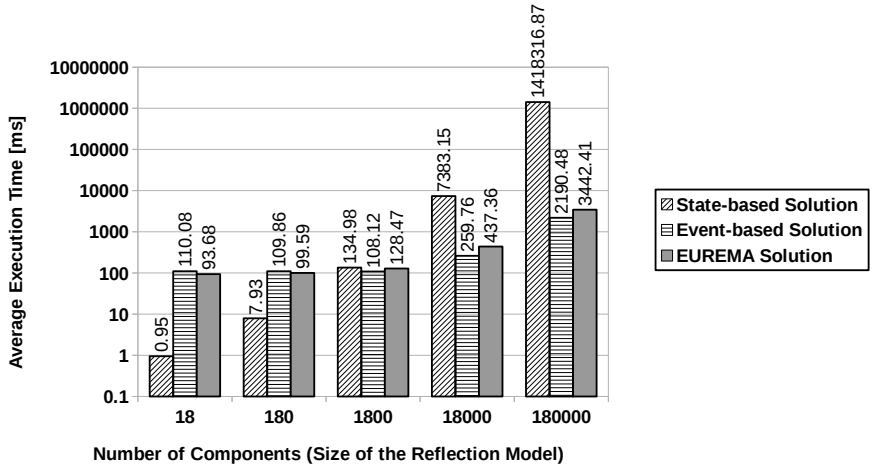
The measurement results are shown in Figure 95 for the self-repair and self-optimization feedback loops. Since the results are similar for both feedback loops, we jointly discuss both charts. The x-axis describes the different sizes of the reflection model in terms of the number of components captured in the model. The y-axis denotes the average execution time of one feedback loop run in milliseconds using a logarithmic scale.

The state-based solution has a very good performance for a small reflection model (*e. g.*, below 1 ms for a model with 18 components) but it does not scale for a large model (*e. g.*, more than 11.8 minutes (708061 ms) for the self-repair and 23.6 minutes (1418317 ms) for

15 Quad-core CPU (Intel Core i5-2400, 3.10GHz), 8GB RAM, Ubuntu 14.04 (Kernel 3.13), Java SE Runtime Environment 1.8.0_40, and EMF Runtime and Tools 2.6.0.



(a) Self-Repair Feedback Loop.



(b) Self-Optimization Feedback Loop.

Figure 95: Average Execution Times and Scalability of the Different Solutions.

the self-optimization loop). This poor scalability is caused by the fact that the state-based solution processes the whole model to perform self-adaptation and it has to maintain a reference model capturing the state of the reflection model before a CF or PI is injected.

In contrast, the event-based solution scales well although it is initially slower than the state-based solution. It takes around 110 ms for one run of the self-repair or self-optimization feedback loop and for a reflection model with 18, 180, 1800 components. The average execution times increase for the self-repair feedback loop to 407 ms (3603 ms) and for the self-optimization feedback loop to 260 ms (2190 ms) when the model consists of 18000 (180000) components. The good scalability of this solution is caused by the event-based processing of the feedback loops that never process the whole reflection model but only events that capture a very small amount of information obtained once from the model in the monitoring step (*e. g.*, the identifier and life cycle state of a single component). Technically, the analysis and planning steps do not use EMF events pointing to the model and they do not access the model at all such that they are not affected by the size of the model.

Finally, the EUREMA solution scales similarly to the event-based solution but typically requires more time for one run of a feedback loop. A run of its self-optimization (self-repair) feedback loop requires 94 ms (187 ms) for the smallest and 3442 ms (6731 ms) for the largest reflection model. Within the EUREMA solution, the self-repair is slower than the self-optimization feedback loop. A reason for the slower performance is that the former

has more model-driven operations that are specified and executed with SDs than the latter. Particularly, the self-repair feedback loop uses 30 SDs while the self-optimization feedback loop only 3 SDs (*cf.* previous section). Thus, the execution of the self-repair feedback loop includes a higher fraction of interpreting SDs than executing code compared to the self-optimization loop. In general, interpreting SDs is more costly than executing code.

Since the EUREMA solution is event- and state-based, it combines the advantages of the other two solutions: improved performance due to events and operating on a rich semantic base due to models capturing the state. Thus, it scales similarly to the event-based solution. However, it is typically slower than the event-based solution as it still processes fractions of the reflection model (state-based processing). Therefore, the EUREMA solution does not process the *whole* reflection model such that it scales better than the state-based solution. However, throughout the feedback loop it uses EMF-based change events that are not self-contained—as in the event-based solution—but refer to the reflection model. Thus, the EUREMA solution and particularly its analysis and planning operations still process fractions of the reflection model to which the change events refer. Thus, the EUREMA solution is more affected by the size of the reflection model than the event-based solution.

Consequently, we may observe that a state-based solution does not scale for large reflection models which calls for incremental approaches to process such models. Such an incremental approach is enabled by (change) events that drive the processing of the reflection model as realized in the EUREMA solution. In this context, EMF itself suffers from scalability issues when querying the contents of large models [68] or modifying large models. For instance, we measured the execution time for consistently deleting a CF1 marker element from the CompArch reflection model using the EMF API, which took on average 78.86 ms (774.13 ms) for a model with 18000 (180000) components. Therefore, those solutions that process more often and extensively the EMF-based reflection model might suffer from such issues. This applies especially to the state-based and EUREMA solutions as all of their operations process (parts of) the model. In contrast, the analyze and planning operation of the event-based solution do not process the model at all.

9.4.3 *Discussion*

The comparative study covering the state-based, event-based, and EUREMA solutions for the self-repair and self-optimization feedback loops gives us some initial confidence that EUREMA provides a good compromise of development costs and runtime performance.

On the one hand, EUREMA is competitive or even more efficient than the other solutions concerning the development costs in terms of SLOC and size of the models. In this context, EUREMA shifts efforts from coding to modeling, which eventually enables flexible solutions that can be dynamically adapted (*e. g.*, layered feedback loops in Section 9.3.5). Models are used to specify and execute individual operations (*e. g.*, SDs) and the whole feedback loop (*cf.* FLD and LD). Interpreting such models may result in shorter turnaround times, which supports developing and testing the feedback loops.

On the other hand, the runtime performance and scalability of the EUREMA solution are acceptable since EUREMA allows feedback loops to operate on a reflection model providing a rich semantic base for the MAPE computations. Although the state-based solution allows the same, it suffers from scalability issues. In contrast, the event-based solution is typically faster and scales a bit better than the EUREMA solution but it does not allow the MAPE computations to operate on a reflection model throughout the feedback loop. Thus, we think that the performance and scalability of the EUREMA solution are acceptable.

These findings obtained from the comparative study are threatened by several aspects. First, the study covered only three types of solutions for two problems (self-repairing and self-optimizing mRUBiS) such that these finding cannot be generalized to other types of solutions or other types of problems. Second, the use of SLOC and size of models as indicators for the development costs can be challenged since we did not take into account any efforts for learning the EUREMA language. However, we developed the EUREMA solution while students implemented the alternative solutions that are not based on EUREMA. Thus, we did not require any knowledge about EUREMA from the students. Finally, the fact that students developed these alternative solutions could impact the findings since they are not experts in the field of self-adaptive software. Thus, it is conceivable that experts could have developed a more cost-effective and runtime-efficient solution than the students did.

9.5 PROTOTYPE

In the previous two sections, we discussed solutions developed with EUREMA to demonstrate the capabilities and effectiveness of EUREMA. In this section, we focus on evaluating the EUREMA prototype itself, that is, the implementation of the EUREMA language and interpreter. While evaluating self-adaptive systems is challenging [370, 434], several researchers proposed criteria for the evaluation. However, most of these criteria target the self-adaptive system, that is, the results of the development [110, 240, 290, 317, 423], and not the development itself. In contrast, Asadollahi et al. [48] propose quality attributes for frameworks that support the development of self-adaptive software systems. These attribute are *flexibility*, *extendibility*, *usability*, *reusability*, *performance*, and *scalability*. Moreover, Krikava [260] who also discussed these attributes for his work suggest *testability* as a further attribute. In the following, we discuss these quality attributes to evaluate the EUREMA prototype. Investigating these attributes for an approach is mostly qualitative and therefore subjective (*cf.* [48, 260, 423]) although we will provide quantitative evidence where possible, for instance, for performance.

9.5.1 *Flexibility*

Flexibility of an approach is defined by Asadollahi et al. [48, p. 66], as "the ability that allows the developer to combine his own mechanism, algorithm, or technique in the design and implementation of the self-management logic". In this sense, EUREMA is flexible at the level of model operations and runtime models as well as at the level of feedback loops.

Considering the first level, EUREMA decomposes a feedback loop into model operations and runtime models that have to be implemented by the developer. For this purpose, the developer can freely decide which mechanism, algorithm, and technique to use for implementing the individual operations for the MAPE phases. Moreover, they can freely decide the modeling languages to express the individual runtime models. For each model operation, they can further decide the style of the implementation (code-based or model-driven) and of its processing (state-based or a combination of event- and state-based) (*cf.* Chapter 7). The only technical restriction is that the implementations of the runtime models are based on the Eclipse Modeling Framework (EMF) and that the implementations of the model operations can be triggered by the EUREMA interpreter through a Java interface.

For instance, we used several techniques to develop the self-repair and self-optimization feedback loops for mRUBiS. We used the CompArch language to express the reflection model of mRUBiS and Triple Graph Grammars (TGGs) to implement the causal connection

between this model and mRUBiS during the monitoring and execute phases (*cf.* Section 8.2). Moreover, we used Story Diagrams (SDs) or only Java to implement the rules and strategies for the analysis, planning, and execution phases (*cf.* Section 9.3). In this context, we reused existing execution engines such as an SD interpreter and a TGG engine to bring the related models to execution within the implementations of the model operations.

Thus, EUREMA fulfills the promise of "DSLs as Enablers of Reuse" [302, p. 319], which can reduce the development efforts (*cf.* Section 9.4.1). Moreover, the reused execution engines are generic and they completely externalize the user-defined inputs in models such as SDs. This paves the way for a library of generic implementations for model operations. Developers can pick a solution from the library and only have to provide the input models such as SDs, which reduces the need for coding.

Although we adopted a rule-based approach to self-adaptation for realizing the mRUBiS example, EUREMA is also open for goal- or utility-based approaches (*cf.* Section 2.2.3) since it does not impose any restriction on the kinds of computations that are performed by the model operations—except of termination (*cf.* Section 6.7).

Considering the level of feedback loops, EUREMA allows developers to freely specify the feedback loops as part of the adaptation engine. Developers decide how many and which model operations and runtime models should be used and how these operations and models are composed to form a feedback loop. Moreover, they freely decide on the number, coordination, triggering, and structuring of feedback loops in the layered architecture of the self-adaptive software. In this context, the only restriction is that EUREMA assumes an external approach and a layered architecture of the self-adaptive software as introduced in Section 2.2.3. Eventually, developers encode their design decisions in FLDs and in an LD provided by EUREMA (*cf.* Chapter 5). Additionally, developers can evolve the design beyond the design time by layered feedback loops and off-line adaptation (*cf.* Sections 5.3 and 5.4) such that the design-time flexibility of EUREMA is extended to runtime.

Thus, EUREMA does not prescribe the design or implementation of the adaptation logic in contrast to typical frameworks for self-adaptive software [355] but it gives developers the flexibility to freely create and evolve the design of the feedback loops.

9.5.2 *Extendibility*

Asadollahi et al. [48, p. 66] consider extendibility as an "attribute [that] permits the developer to extend the framework to add new features or integrate it with other frameworks".

On the one hand, EUREMA requires from developers to extend or rather complement the EUREMA framework as it does not prescribe the techniques or algorithms for the individual adaptation activities such as analysis and planning. In contrast, it provides the flexibility such that developers can integrate their own techniques and algorithms when implementing individual model operations (*cf.* previous section). Thereby, the integrated techniques and algorithms can go beyond code fragments and be frameworks or toolkits. For instance, we used the *Story-Driven Modeling (SDM) Tools*[16] comprising a graphical editor, interpreter, and debugger to develop and execute Story Diagrams (SDs) to implement and execute individual model operations within the EUREMA-based feedback loops for the mRUBiS example (*cf.* Section 9.3).

Moreover, EUREMA does not depend on any specific adaptable software such that it can be reused in other projects to develop feedback loops as it has been done for *SimuLizar* [349]. Since EUREMA is based on EMF, such an integration is eased if it happens within the

16 Story-Driven Modeling (SDM) Tools: `http://www.mdelab.de`.

EMF ecosystem, that is, if the other projects are also based on EMF and work with EMF-based models. Finally, extending the core functionality of EUREMA requires changing the EUREMA language (metamodel and semantics) and its tooling (interpreter and editor). Such kind of changes are typical for any DSL approach when the DSL should evolve. In this context, MDE and particularly the "unification power of models" (*cf.* Section 2.1.1) as realized by EMF supports integrating or extending EUREMA. For instance, there exists approaches that address the evolution of languages and models in the EMF ecosystem [296].

9.5.3 *Usability*

The usability of a framework "determines how easy it is to employ the framework in an application" [48, p. 66]. Assessing the usability of software is generally difficult [394] and particularly the usability of languages as it is influenced by human factors [197, 384] and specifics of the language such as its purpose [384] or notation [304]. Therefore, we discuss qualitatively the usability of EUREMA to develop a feedback loop.

First, EUREMA targets the adaptation engine with feedback loops and not the adaptable software to promote separation of concerns. Moreover, we designed the EUREMA language by following the guidelines for DSLs discussed in Section 9.1.2. For instance, the EUREMA language reflects only the necessary concepts of the domain of self-adaptive software and it is kept rather simple by avoiding unnecessary generality and conceptual redundancy as well as limiting the number of language elements (*cf.* guidelines 8–12). Consequently, developers can focus on the essence when using EUREMA, that is, on developing feedback loops for self-adaptive software.

Additionally, the EUREMA language leverages well-known concepts of the self-adaptive software domain such MAPE-K and layered architectures (*cf.* Section 2.2.3). Thus, developers who are familiar with the domain use such well-known concepts and do not have to learn new concepts for applying EUREMA. Similarly, the notation of EUREMA shares common characteristics of behavior- or structure-dominant models with existing notations. For instance, FLDs shares characteristics with UML Activity and LDs with UML Object or Package diagrams (*cf.* Section 9.1). Therefore, we think that developers who are familiar with MDE or have at least basic knowledge of UML can easily grasp the modeling concepts of FLDs and LDs. In this context, the EUREMA editor is integrated and the EUREMA interpreter can be executed within the Eclipse environment such that developers who are familiar with Eclipse and its modeling framework (EMF) can easily use the EUREMA tools.

Finally, the EUREMA interpreter further supports the developer as it directly executes the resulting FLD and LD models such that the execution happens at the same abstraction level and with the same concepts used for modeling (*cf.* Chapter 6). Thus, developers can directly retrace the execution to the models they created.

Therefore, we think that developers can rather easily adopt EUREMA if they have basic knowledge about the domain of self-adaptive software, MDE, and Eclipse.

9.5.4 *Reusability*

Reusability "is the ability of using the framework in different systems" [48, p. 66], that is, for dynamically adapting different (types of) adaptable software systems.

For experimenting with EUREMA, we applied EUREMA on top of our implementation framework that enables the monitoring and adaptation of any EJB-based software system (*cf.* Section 8). Thus, EUREMA is applicable to different EJB-based systems when the imple-

mentation framework is used. However, EUREMA can also be reused independently of the implementation framework and thus for different types of systems. In this case, developers have to implement accordingly the monitoring and execute operations that causally connect a reflection model and the adaptable system (*i. e.*, integrating the adaptation engine and the system) based on the system's sensors and effectors. As one example, EUREMA has been reused in the SimuLizar project targeting cloud-based systems [349]. Thus, EUREMA can be reused for different types of adaptable software systems but developers have to connect the adaptation engine to the specific system.

Besides reusing the EUREMA language and interpreter, reuse of concrete solutions developed with EUREMA is conceivable. On the one hand, model operations can be generically implemented and thus reused in different EUREMA-based feedback loops (*cf.* Section 9.5.1). On the other hand, using abstract reflection models of the adaptable software might allow reuse of the adaptation logic's analysis and planning mechanisms for different software platforms. Since these models are platform-independent, the mechanisms can be realized in a platform-independent manner and thus they might be reusable for other platforms. For instance, for self-healing a restart or redeployment of a component can be planned at the model level independent of the specific platform or application. However, the extent to which such a reuse is feasible requires further investigations when developing more solutions for different platforms and applications with EUREMA.

9.5.5 *Performance*

Asadollahi et al. [48] consider the overhead that is caused at runtime by the adaptation engine or mechanism as the performance of an approach. To evaluate the overhead of EUREMA, we conducted an experiment to quantify the load and overhead of the EUREMA interpreter compared to a code-based solution to execute the self-repair feedback loop defined by the FLDs in Figure 19 on Page 65 and Figure 21 on Page 66.

For the experiment, we considered the case that each run of the feedback loop instance always identifies failures. A warm-up phase taking place before the measurements executes the instance more than three times, such that the condition branching the control flow in the FLD (see Figure 19 on Page 65) is always fulfilled. Thus, for the measurements, all of the five basic model operations of the self-repair feedback loop are executed in each run of the instance. As implementations for these operations, we provided software modules as mocks that have runtime models as input as it is defined in the FLDs. Moreover, all runtime models that are the output of any operation are already used as input of the same operation. Thus, no new models are produced by the mocks. In contrast, all runtime models are pre-defined and they are not changed at all by the mocks. Each mock can be assigned a duration, for which it generates load to simulate computations of the operation.

To compare the overhead and load of the EUREMA interpreter, we implemented a code-based solution in *Java* that executes the self-repair feedback loop. This solution does not use any EUREMA model but it hard-codes the execution by sequentially invoking the five mocks, one for each model operation of the self-repair feedback loop. Moreover, this code-based solution provides the runtime models required as input for invoking the mocks.

The experiment has two parameters. First, the duration assigned to the mocks defines the internal computation time of the model operations. The same duration is assigned to all mocks for one run of the experiment but it varies between different runs. This results in four runs of the experiment, either assigning a duration of 0 ms, 5 ms, 10 ms, or 20 ms to each mock. Since the self-repair loop has five basic model operations, this constitutes a

total computation time of either 0 ms, 25 ms, 50 ms, or 100 ms for one execution of the loop instance. The second parameter is the frequency of consecutive executions of the instance, which determines the execution rate. The frequency is defined by its reciprocal, that is, the period of time between two consecutive activations of the instance. For each of the four runs of the experiment, we varied the period starting from 15 ms and doubling it until 960 ms. For example, a period of 15 ms means that the feedback loop instance is executed every 15 ms, which is only feasible if the total computation time of the mocks (model operations) plus the time used by the code-based solution or EUREMA interpreter is below 15 ms.

For each feasible combination of these two parameters, we measured the load of the Java virtual machine for the code-based solution and the EUREMA interpreter while executing the self-repair feedback loop for a total time of ten minutes.[17] In the following, we discuss the experimental results. Figure 96 visualizes the average Central Processing Unit (CPU) load of the code-based solution (solid gray lines) and the EUREMA interpreter (dashed black lines) for the different frequencies of executing the feedback loop instance. Moreover, each graph refers to a specific total computation time of the feedback loop instance (see legend). Based on this figure, we may generally observe that the average load decreases for both solutions and all computation times of the feedback loop instance if the period between consecutive runs of the feedback loop instance increases. This observation is not surprising since executing a feedback loop less frequently is supposed to cause less load.

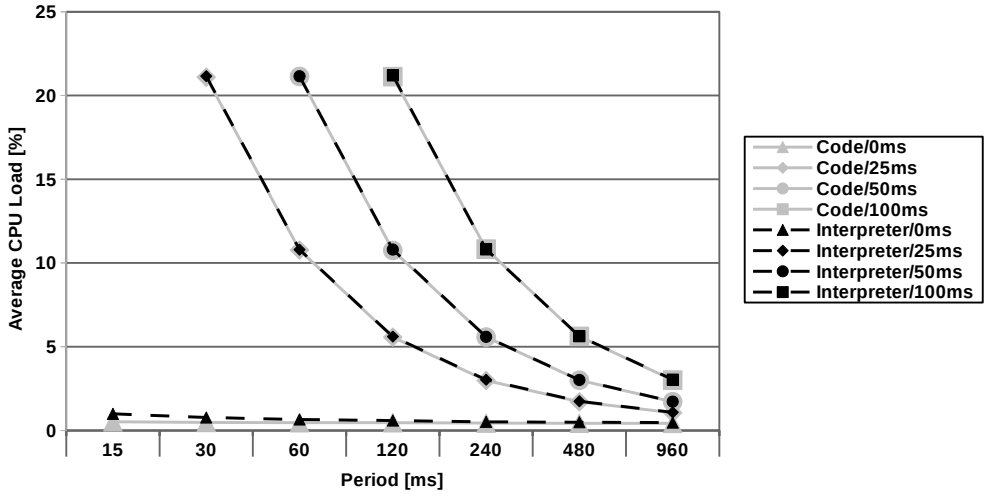


Figure 96: Average CPU Load of the *Code*-based Solution and EUREMA *Interpreter*.

Moreover, we may observe that the EUREMA interpreter causes slightly more load than the code-based solution when the computation time of the feedback loop is 0 ms. However, for the other cases of the computation time, there are no apparent differences between the loads of the code-based solution and the interpreter, and the corresponding graphs overlap. Thus, the overhead of the interpreter is only noticeable for the hypothetical case that the feedback loop does not perform any computations that cause load.

To further investigate the overhead, we quantified it as the difference between the average loads of the interpreter and of the code-based solution for each case of the total computation time (see Figure 97). For all cases, we observe that the overhead of the EUREMA

17 The experiments were conducted on the following platform: quad-core CPU (Intel Core i5-2400, 3.10GHz), 8GB RAM, Ubuntu 12.04 (Kernel 3.2.0-33), Java SE Runtime Environment 1.6.0_31, and EMF Runtime and Tools 2.7.2. The CPU load has been measured with *Java VisualVM* of the Java Development Kit 6 (1.6.0_31).

interpreter with respect to the code-based solution is always below 0.2 percentage points and tends to decrease with increasing frequency periods. This observation is supported by the overhead we predicted (*cf. Prediction* graph), which is the average overhead based on all measurements for all frequencies and computation times normalized for the frequencies.

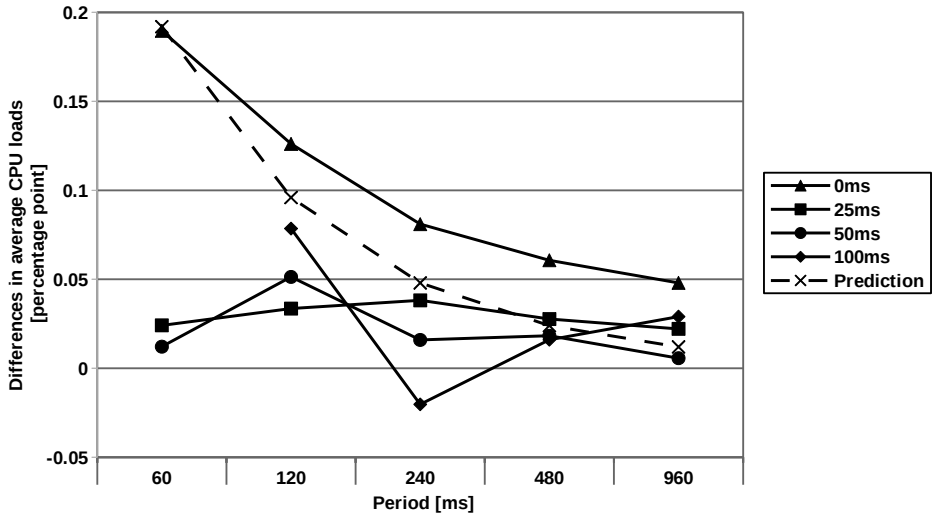


Figure 97: Interpreter Overhead as the Differences in Average CPU Loads.

Consequently, the experiment shows that the overhead of interpreting EUREMA models is negligible. In particular, the hypothetical case when the feedback loop's operations do not perform any internal computations revealed the pure load caused by the interpreter. The average of this pure load was for all experiments below 1% (*cf. Interpreter/oms* graph in Figure 96). Thus, in absolute terms, the EUREMA interpreter works efficiently for the considered case of executing the EUREMA models specifying the self-repair feedback loop.

The validity of the experiment is threatened as we implemented the alternative, code-based solution. Thus, the comparison of this solution with the EUREMA interpreter needs further investigations. Nevertheless, we have shown that the interpreter works efficiently in absolute terms by causing a negligible average load. Another threat of validity is the specific self-repair feedback loop we used. However, we think that this feedback loop is a typical one since it follows the MAPE-K principle similar to other approaches such as Rainbow (*cf.* Section 9.2.1). Moreover, the complexity of the specific feedback loop in terms of the numbers of model operations and runtime models can be questioned and how the interpreter behaves for larger EUREMA models. However, state-of-the-art approaches do not employ significantly more complex feedback loops with respect to the size of the FLDs as the feedback loop considered here (*cf.* Section 9.2). Nevertheless, the scalability of the interpreter, which is determined by the specific EUREMA models and not by the EUREMA language, needs to be further investigated and will be discussed in the following section.

9.5.6 *Scalability*

The scalability "shows the capability of a system in properly handling a growing amount of load" [48, p. 66]. Thus, we consider the scalability of the EUREMA language and interpreter to handle large adaptation engines and large adaptable software systems.

An adaptation engine can be large in terms of the number of feedback loops and the size of the individual feedback loops. In EUREMA, each feedback loop is individually modeled in an FLD and can be further decomposed into multiple FLDs (*cf.* Section 5.1). Moreover, the coordination of multiple feedback loops is modeled in distinctive FLDs in addition to the FLDs describing the feedback loops (*cf.* Section 5.2). Eventually, instances of these FLDs are composed in an LD to form a feedback loop, to coordinate multiple feedback loops, and to structure the feedback loops in the adaptation engine. Therefore, EUREMA supports decomposing and composing an adaptation engine such that the EUREMA language scales to model larger adaptation engines. Since the EUREMA language is not used to model the adaptable software, it is not affected by the size of the software.

In the previous section, we have shown that the EUREMA interpreter causes a negligible overhead when executing two FLDs describing a feedback loop whose size is comparable to feedback loops of state-of-the-art approaches. Thus, FLDs need not to be larger in practice. Nevertheless, we think that the runtime performance of the EUREMA interpreter scales to larger FLDs since the interpreter interprets stepwise the FLDs to execute one operation of the feedback loop after the other. Thus, at one point in time when executing a single feedback loop, multiple coordinated feedback loops, or layered feedback loops, the interpreter processes exactly one model operation in the FLDs to trigger the corresponding implementation of the operation. Processing and executing a model operation are not influenced by the (number of) other operations of the feedback loop(s) (*cf.* Sections 6.4 and 6.5). Thus, the interpreter does not process the other operations in the FLDs or even the whole FLDs to execute a model operation. Consequently, we do not expect scalability issues of the runtime performance when executing large FLDs. However, larger FLDs will naturally cause a larger memory footprint. Nevertheless, if the adaptation engine contains feedback loops that are executed concurrently, scalability can become an issue when the number of feedback loop becomes large. However, such an issue is not specific to EUREMA and might require to distribute the adaptation engine on different machines. In this case, multiple instances of the EUREMA interpreter can be deployed on different machines and they are and must be executed independently of each other.

Finally, the size of the adaptable software does not affect the EUREMA interpreter that executes only the FLDs but does not process the reflection model describing the adaptable software. However, it affects the mechanisms and algorithms used for implementing the model operations that, for instance, analyze the reflection model. Consequently, the size of the adaptable software and reflection model affects specific solutions developed with EUREMA but not EUREMA itself. In Section 9.4.2, we have shown that an EUREMA-based solution to self-repairing and self-optimizing mRUBiS scales acceptably with increasing numbers of components in mRUBiS. Thus, we have some evidence that scalable feedback loops can be built with EUREMA.

9.5.7 *Testability*

Finally, we discuss testability, which is the required effort for testing a software to ensure that it performs its specified functionality [289]. In the context of developing self-adaptive software, it refers to testing the adaptation logic realized by the feedback loops.

Feedback loops developed with EUREMA are amenable for testing. In general, during the modeling of feedback loops, EUREMA checks the model for conformance to the metamodel and well-formedness, which ensures basic constraints (*cf.* Section A.1). For instance, an FLD must have a final state indicating the termination of a feedback loop. More specif-

ically, we proposed a testing scheme for feedback loops operating on architectural reflection models [14]. Thus, this scheme is applicable for EUREMA-based feedback loops. It supports testing individual MAPE activities (similar to unit tests) and fragments of feedback loops. The testing scheme is based on an architectural reflection model (*e. g.*, a model expressed in CompArch) and makes use of the simulator for the adaptable software (*cf.* Section 8.4). This enables early testing of an EUREMA-based feedback loop before the loop has been completely implemented and connected to the adaptable software.

For instance, in our mRUBiS example, testing the analysis and planning operations of the self-repair feedback loop can ensure that the analysis properly detects the critical failures in the reflection model and that the planning properly addresses these failures in the model. To drive such tests, failures can be injected into the reflection model, then the analyze and planning operations are executed to identify and address these failures in the model, and finally the model is compared to a reference model of the reflection model (*i. e.*, the oracle). Similar ways to test the whole MAPE feedback loop are discussed in [14].

Therefore, we can conclude that EUREMA models are amenable for testing, particularly early in the development. Moreover, we mitigate the testing efforts since the tests are based on the reflection model and they do not require a complete implementation and integration of the feedback loop and adaptable software.

9.5.8 *Discussion*

In this section, we assessed EUREMA against the quality attributes of flexibility, extendibility, usability, reusability, performance, scalability, and testability. The assessment was for most attributes qualitative and therefore subjective while we provided quantitative data for the performance and partially for the scalability of EUREMA. Consequently, more evidence is needed to thoroughly assess EUREMA, for instance, concerning the usability of the EUREMA language, which can pinpoint notational issues. Nevertheless, we think that we have achieved with EUREMA a flexible and extensible approach for specifying and executing feedback loops since it allows developers to integrate their own techniques, mechanisms, and algorithms into EUREMA-based feedback loops, which extends the whole approach. Finally, a more thorough assessment of the reusability and testability is required, which, however, will be feasible when more solutions have been developed with EUREMA.

9.6 REQUIREMENTS COVERAGE

Based on the overall discussion and evaluation of EUREMA, we now summarize its capabilities by discussing its coverage of the requirements for engineering self-adaptive software presented in Chapter 3. The requirements are grouped into modeling and execution.

9.6.1 *Modeling*

To identify the essential concepts that the EUREMA language has to cover, we informally analyzed the domain of self-adaptive software. This analysis clarifies and defines the scope of the domain, the domain concepts, and the relationships between the concepts, which should be directly addressed by the EUREMA language. As proposed by Mernik et al. [302] for developing a DSL, we use a feature model to describe the domain concepts, in this case for self-adaptive software as shown in Figure 98. These concepts are derived from

the requirements for self-adaptive software (*cf.* Chapter 3). In the following, we briefly discuss these concepts and refer to the corresponding requirements (R) for each of them.

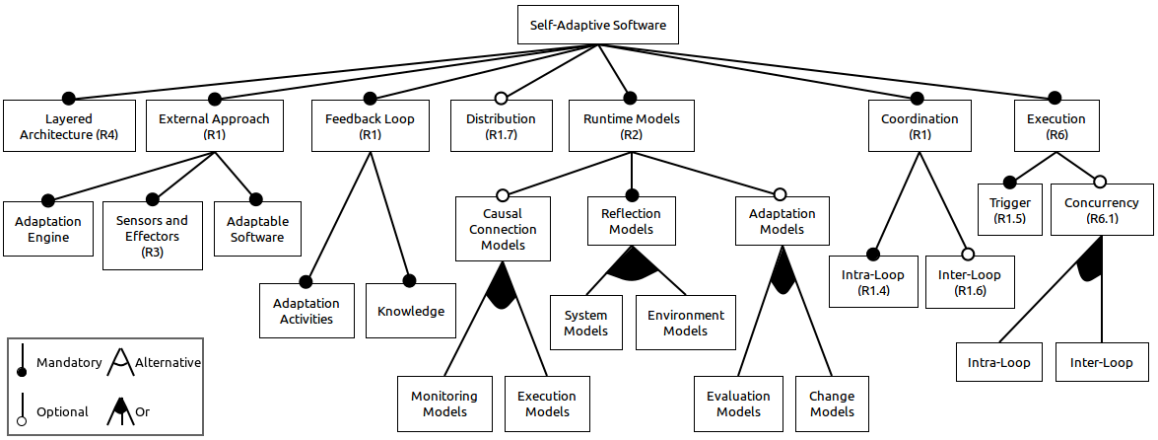


Figure 98: Feature Model for the Domain of Self-Adaptive Software.

With EUREMA, we focus on self-adaptive software that adopts a *layered architecture* (R4) and particularly follows the **external approach** (R1). Therefore, we require a clear separation between the **adaptation engine** and the **adaptable software** while both are connected through **sensors and effectors** (R3). Additionally, the adaptation engine can be split in several layers by stacking feedback loops on top of each other. Hence, we define the scope of the domain as self-adaptive software with such kind of an architecture while other architectural styles (*e. g.*, the internal approach and non-layered styles) are not addressed.

The adaptation engine employs one or more **feedback loops** (R1) that often follow the MAPE-K blueprint and therefore consist of **adaptation activities** (*e. g.*, monitor, analyze, plan, and execute) and **knowledge**. If the self-adaptive system is distributed, the **distribution** of the feedback loops in the system and how adaptation is achieved in a distributed fashion are relevant (R1.7). We refine the knowledge of a feedback loop to a set of **runtime models** (R2). Such models are typically **reflection models** representing the adaptable software and environment (**system** and **environment models**). Moreover, runtime models can be used to specify and run the adaptation activities of feedback loops (**causal connection** and **adaptation models**) such as **monitoring models** for the monitor, **execution models** for the execute, **evaluation models** for the analyze, and **change models** for the planning activity (*cf.* categorization of runtime models in Section 4.1). The **coordination** of runtime models and adaptation activities is relevant as it either establishes the feedback loop (**intra-loop**) or the interplay between multiple feedback loops (**inter-loop**) (R1.4, R1.6).

Finally, the language has to specify concepts that are relevant for the **execution** (R6). Such concepts are **triggers** that define when a feedback loop should be executed (R1.5) and constructs that allow the **concurrent** execution of feedback loops (**inter-loop** concurrency) or of adaptation activities within a feedback loop (**intra-loop** concurrency) (R6.1).

Based on the requirements, we identified these concepts to be directly reflected in the EUREMA language. Therefore, engineers may directly use these known concepts to describe feedback loops and do not have to translate these (mental) concepts to other, less suitable paradigms such as object orientation realized in existing languages. In the following, we outline how EUREMA leverages these concepts in its language. An overview is given in Table 10. Domain concepts are formatted bold and EUREMA concepts italic.

Table 10: Domain Concepts for Self-Adaptive Software and EUREMA Language Concepts.

Domain Concepts	EUREMA Language Concepts
Layered Architecture	An LD reflects a layered architecture with its *layers*, *megamodel modules* (feedback loop instances), and *software modules* (instances of the adaptable software or legacy feedback loops). *Sense* and *effect* relationships among modules cross layers such that modules at a certain layer are only allowed to monitor and control modules at lower layers. This ensures a acyclic structure of modules that sense and effect each other.
External Approach (Adaptation Engine, Sensors and Effectors, Adaptable Software)	An LD captures the external approach since the **layered architecture** comprises this approach. The adaptable software corresponds to the lowest *layer* of an LD and the adaptation engine to the *layers* on top of the lowest one. EUREMA abstracts from sensors and effectors as they are usually application-specific while the EUREMA language is application-independent. However, sensor data is revealed to specify **triggers**.
Feedback Loop (MAPE-K) (Adaptation Activities and Knowledge)	An FLD specifies a feedback loop that can follow the MAPE-K blueprint. The adaptation activities are captured as *model operations* in an FLD, which can be further stereotyped, for instance, with ≪Monitor≫, ≪Analyze≫, ≪Plan≫, and ≪Execute≫ to point out the kind of MAPE activity they realize. The knowledge is materialized by **runtime models** and captured by *runtime models* in FLDs.
Distribution	The EUREMA language does not capture the distribution of feedback loops and assumes a centralized solution of the adaptation engine.
Runtime Models (Reflection, System, Environment, Causal Connection, Monitoring, Execution, Adaptation, Evaluation, and Change Models)	In an FLD, runtime models are directly reflected as *runtime models* regardless of the kind or purpose they are fulfilling. To distinguish the purpose of a runtime model, engineers can apply stereotypes such as ≪ReflectionModel≫, ≪SystemModel≫, ≪EnvironmentModel≫, ≪CausalConnectionModel≫, ≪MonitoringModels≫, ≪ExecutionModel≫, ≪AdaptationModel≫, ≪EvaluationModel≫, and ≪ChangeModel≫.
Coordination (Intra-Loop and Inter-Loop)	EUREMA captures the intra-loop coordination among **adaptation activities** and **runtime models** by reflecting the *control flow* among model operations and the *model usage*, that is, the *input* and *output* models of operations. The usage can be refined with labels c, d, w, r, and a standing for creating, destroying, writing, reading, and annotating models. The inter-loop coordination of feedback loops within a layer is captured by the same concepts (*i. e.*, *model operations*, *runtime models*, *control flow*, and *model usage*) as for modeling feedback loops and intra-loop coordination in FLDs. In LDs, *use* relationships among coordinating and coordinated megamodel modules make the interplay visible. The coordination of feedback loops across layers is reflected in LDs by *sense* and *effect* relationships among modules and by **triggers** for intercepting modules.
Execution (Trigger, Intra-Loop and Inter-Loop Concurrency)	LDs allow engineer to specify *triggering conditions* using *events* (including *event types*) or *periods* defining when a megamodel module should be executed. Individual conditions for different feedback loops and omitting any **inter-loop coordination** enable the concurrent execution of feedback loops (inter-loop concurrency). Intra-loop concurrency (*i. e.*, the concurrent execution of **adaptation activities** within a loop) is not supported.

Layered Architecture. EUREMA supports the engineering of self-adaptive software that adopts a layered architecture. The adaptable software is located at the lowest layer and feedback loops can be stacked on top at various higher layers. Such architectures are captured by LDs for which the EUREMA language provides the concepts of *layers*, *megamodel modules* representing feedback loop instances, and *software modules* representing instances of the adaptable software or legacy feedback loops. An LD structures such modules in layers and wires them by *sense* and *effect* relationships. These relationships have to cross layers such that modules at a certain layer are only allowed to monitor and control modules at lower layers. This constraint determines and maintains the strict layering of an architecture and avoids cyclic adaptation relationships between modules. Finally, *use* relationships among modules make functional dependencies among megamodel modules visible.

The LD has been introduced and refined in Sections 5.1.1 and following. Particularly, the layering of feedback loops has been discussed in Section 5.3.

External Approach. The **layered architecture** as supported by EUREMA comprises the external approach such that the *layers* in an LD can be mapped to this approach. The lowest *layer* contains only and therefore corresponds to the **adaptable software**. All the other *layers* on top of the lowest one address adaptation concerns and thus correspond to the **adaptation engine**. In EUREMA, we extend the external approach by structuring the adaptation engine into multiple layers, which supports the stacking of feedback loops (*cf.* Section 5.3) and off-line adaptation (*cf.* Section 5.4). Thus, the domain concepts of adaptation engine and adaptable software are indirectly reflected in the language by the *layers* in LDs. The **sensors and effectors** used to connect the adaptation engine and the adaptable software are often specific to the application realized by the software and to the technical platform (*cf.* Section 2.2.3). To keep EUREMA generic, they are not captured in the language that, however, reflects by *sense* and *effect* relationships in LDs where sensors and effectors are used for monitoring and controlling the adaptable software. Nevertheless, sensor data is revealed and captured in the language when specifying **triggers** for feedback loops.

Feedback Loop (MAPE-K). The EUREMA language captures the feedback loop following the MAPE-K blueprint in FLDs. Particularly, the **adaptation activities** such as monitor, analyze, plan, and execute are reflected by *model operations* in FLDs. Such operations can be stereotyped with ≪Monitor≫, ≪Analyze≫, ≪Plan≫, and ≪Execute≫ to emphasize the kind of adaptation activity they are realizing. These stereotypes can be exchanged or extended to address other blueprints for feedback loops and adaptation activities without affecting the core concept of a model operation. The **knowledge** part of a feedback loop is materialized by **runtime models**. With EUREMA, we address self-adaptive software that uses runtime models such that no other means to capture knowledge is explicitly supported. The FLD has been introduced and refined in Sections 5.1.1 and following.

Distribution. Considering a distributed self-adaptive software system, the distribution of feedback loops and the decentralization of self-adaptation are essential for the design of the system. Currently, we do not consider distributed adaptation engines and therefore, the EUREMA language does not reflect the distribution of feedback loops. In contrast, we assume a centralized solution for the adaptation engine.

Runtime Models. EUREMA supports feedback loops whose **knowledge** is materialized by runtime models and whose *model operations* (**adaptation activates**) may further be specified and executed by runtime models. Hence, such runtime models are either the working data or the executable specifications of model operations and they are directly reflected as

runtime models in FLDs. Thus, engineers who are familiar with idea of runtime models may intuitively map the known domain concepts to *runtime models* and *model operations* in FLDs.

Finally, *runtime models* in FLDs can be stereotyped to emphasize their purpose in a feedback loop (*cf.* Table 10) according to the categorization of runtime models discussed in Section 4.1. This supports the perception of a feedback loop design. The stereotypes can be exchanged or extended to address other kinds of runtime models, which are not captured in the given categorization, without affecting EUREMA's core concept of a *runtime model*.

Coordination. The interplay of **adaptation activities** (*model operations*) and **runtime models** (*runtime models*) eventually forms a feedback loop and is therefore captured by FLDs. For the **intra-loop coordination**, the EUREMA language provides the concepts of *control flow* between model operations and *model usage* by operations. The latter refers to specifying whether runtime models are the *input* or *output* of operations, which can be further refined by labels (*e. g.*, with `c`, `d`, `w`, `r`, and `a` for creating, destroying, writing, reading, and annotating runtime models). Such labels can be exchanged or refined by engineers if needed.

The **inter-loop coordination** refers either to the interplay between feedback loops of the same layer or at different layers. For the first case, the same language concepts of FLDs (*i. e.*, *model operations*, *runtime models*, *control flow*, and *model usage*) as for modeling feedback loops and **intra-loop coordination** are used (*cf.* Section 5.2). Moreover, the interplay of the megamodel module realizing the coordination and the megamodel modules realizing the feedback loops is reflected by *use* relationships among the modules in the LD. The second case considers feedback loops that control other feedback loops at lower layers. Such an interplay is reflected by *sense* and *effect* relationships among megamodel modules in LDs as well as by **triggers** that allow the interception of megamodel modules (*cf.* Section 5.3).

Execution. To execute feedback loops, EUREMA supports **triggers** that are specified as *triggering conditions* in LDs (*cf.* Section 5.1.3). Such triggers can be based on events and time and they are attached to *sense* relationships among modules, which reflect flows of sensor events from sensed to sensing modules. EUREMA provides the concepts of *event* and *event types* for specifying event-based triggers and of *period* to specify time-based triggers.

The EUREMA language concepts allow engineers to specify **inter-loop concurrency**, that is, the concurrent execution of multiple feedback loops operating at the same layer. In this case, there is no **inter-loop coordination** in place and each loop has its individual *triggering condition* activating its execution regardless of the other loops. **Intra-loop concurrency**, that is, the concurrent execution of **adaptation activities** within a feedback loop is not supported by EUREMA. Hence, the language does not provide concepts to specify concurrent *model operations* in FLDs as it prescribes a sequential execution of the operations.

Thus, all domain concepts except of the distribution of feedback loops and the intra-loop concurrency are reflected in the EUREMA language and developers can use them to model their solutions. Leveraging domain concepts in the language, EUREMA addresses the corresponding requirements from which we derived these concepts. Some of the other requirements are addressed by the language design or by allowing a flexible use of EUREMA.

R1: Requirements for Feedback Loops. The FLD with its four language concepts *model operations*, *runtime models*, the *control flow* among the operations, and the *model usage* by the operations describes a feedback loop (*cf.* Sections 5.1.1 and 5.1.2). Moreover, an LD structures such feedback loops in the adaptation engine. Thus, FLDs and LDs make the feedback loops explicit in the design of self-adaptive software, which addresses requirement R1.1.

Since FLDs capture the individual elements of a feedback loop, they further support a modular specification (*cf.* Section 5.1.4). Thus, a single feedback loop can be decomposed and specified in multiple FLDs, which addresses requirement R1.2.

In this context, a modular specification of feedback loops supports modeling variability of feedback loops. For instance, a feedback loop fragment specified in an FLD can be replaced by another fragment specified in another FLD. Additionally, such a variability can be made explicit in the LD (*cf.* Section 5.1.5), however, not to an extent as feature models would do with respect to capturing alternatives and constraints among alternatives. Therefore, EUREMA partially addresses requirement R1.3.

Besides capturing the individual adaptation activities and runtime models, an FLD describes the control flow among the activities and the usage of the models by the activities, which forms a feedback loop. If a feedback loop is specified in multiple FLDs, an LD composes these FLDs to eventually form the overall feedback loop (*cf.* Sections 5.1.2 and 5.1.4). Thus, FLDs and LDs realize the intra-loop coordination, which addresses requirement R1.4. The LD further supports specifying triggering conditions that determine when a feedback loop should be executed (*cf.* Section 5.1.3), which addresses requirement R1.5.

Besides a single feedback loop, EUREMA supports modeling multiple feedback loops and their coordination. The inter-loop coordination is realized with the same language concepts as intra-loop coordination (*cf.* Section 5.2). Therefore, EUREMA addresses requirement R1.6. Despite supporting multiple feedback loops, the EUREMA language does not provide concepts to model distributed feedback loops. Thus, distribution of feedback loops and therefore requirement R1.7 is not supported.

R2: Requirements for Runtime Models. The FLDs capture the runtime models that describe the knowledge in a feedback loop, which otherwise remains abstract and vague. Thus EUREMA addresses requirement R2.1. Thereby, EUREMA is open for any type of runtime model. Thus, developers can use any language to express the runtime models. The only restriction is technical and requires EMF-based languages. This openness of EUREMA addresses requirements R2.2. For instance, we used several languages such as CompArch, TGGs, or SDs to express the models (*cf.* Sections 8.2 and 9.3). Moreover, EUREMA does not impose any restriction on the abstraction level of the runtime models, which therefore allows the use of abstract runtime models and addresses requirement R2.3. For instance, we used a reflection model that targets a higher level of abstraction by describing the adaptable software at the architectural level and independently of the platform (*cf.* Section 8.2).

R3: Requirements for Sensors and Effectors. We assume that the adaptable software provides sensors and effectors that can be used by EUREMA-based feedback loops. Thus, EUREMA does not address technical details of the sensors and effectors but only captures in the LD which feedback loops sense and effect the adaptable software. These loops are directly connected to the adaptable software in the LD, which addresses requirement R3.1.

Whether the sensors and effectors support parameter and structural adaptation as well as whether they can be dynamically changed depend on the adaptable software and platform rather than on EUREMA. For instance, for our experiments (*cf.* Section 9.3) we use a platform (*cf.* Section 8.1) that supports parameter and structural adaptation and the dynamic (de)activation of sensors, for instance, for performance monitoring. The feedback loops developed with EUREMA exploits these capabilities by performing parameter and structural adaptation (*cf.* Section 9.3.1) and by activating the sensors as needed. For instance, the self-repair feedback loops does not require and therefore does not activate the sensors for performance monitoring in contrast to the self-optimization loop. Consequently, EUREMA does not directly support but is open for the requirements R3.2 and R3.3.

R4: Requirements for Layered Architectures. The EUREMA language supports modeling layered architectures. An LD captures such an architecture by structuring modules into layers (*cf.* Section 5.1.2). In this context, EUREMA supports feedback loops that operate on top of each other, that is, a higher-layer feedback loop controls a lower-layer feedback loop. Particularly, EUREMA-based (lower-layer) feedback loops are adaptable by construction while (higher-layer) feedback loops are specified similarly to any other feedback loop (*cf.* Section 5.3). This addresses requirement R4.1. In addition to stacking feedback loops, EUREMA allows reflection on EUREMA-based feedback loops either in a procedural (*cf.* Section 5.3.1) or declarative (*cf.* Section 5.3.2) manner, which addresses requirement R4.2. An experiment demonstrating the benefits of such layered feedback loops in terms of dynamically adjusting the adaptation logic has been discussed in Section 9.3.5.

R5: Requirements for Off-line Adaptation. EUREMA provides a mechanism that allows engineers to adapt EUREMA-based self-adaptive software that is already running (*cf.* Section 5.4). This supports maintaining the self-adaptive software in way that has not been foreseen during development and therefore addresses requirement R5.1. When executing such an adaptation that has been planned off-line by an engineer to the running self-adaptive software, EUREMA ensures that this adaptation does not interfere with any self-adaptation that is occurring at the same time. Hence, EUREMA coordinates the execution of changes due to maintenance and self-adaptation, which addresses requirement R5.2.

Summing up, EUREMA addresses all modeling requirements except of the distribution and partially the variability of feedback loops. In the next section, we discuss EUREMA's coverage of the requirements for executing self-adaptive software.

9.6.2 *Execution*

In general, the requirements for executing self-adaptive software comprises the ability of an engineering approach to execute the feedback loops according to the specification of the feedback loops, thus according to the addressed requirements for modeling. EUREMA supports executing feedback loops according to its modeling capabilities (*cf.* previous section) but does not support the execution of distributed feedback loops. Additionally, requirements specific for execution are introduced in Chapter 3 and discussed in the following.

R6: Requirements for Execution. EUREMA supports the concurrent execution of independent feedback loops (*cf.* Sections 5.2.1 and 6.3). However, it does not support concurrent executions of model operations that belong to the same or to coordinated feedback loops (*cf.* Sections 6.4 and 6.5). Nevertheless, this does not prevent any concurrency within an implementation of a model operation (*cf.* Section 6.7). Thus, EUREMA partially addresses requirement R6.1 calling for a concurrent execution of feedback loops.

EUREMA-based feedback loops operate on a reflection model capturing the state such as the runtime architecture of the adaptable software. The individual model operations of such a feedback loop can process the reflection model in state-based or event-based manner (*cf.* Chapter 7). State-based operations work in a batch mode while event-based operations are driven by change events and therefore, they can process incrementally the reflection model (*cf.* results of the comparative study in Section 9.4). EUREMA allows developers to decide on using a state- or event-based approach, which addresses requirement R6.2.

Finally, EUREMA achieves a runtime-efficient execution. The EUREMA interpreter causes a negligible overhead and therefore does not disturb the performance of the adaptable software (*cf.* Section 9.5.5). Moreover, we have shown that a concrete EUREMA-based

feedback loop scales acceptably concerning runtime performance when the adaptable software grows in size (*cf.* Section 9.4.2). Consequently, EUREMA interpreter achieves a runtime-efficient execution of feedback loops, which addresses requirement R6.3.

9.6.3 *Discussion*

As summarized in Table 11, EUREMA addresses most of the requirements for engineering self-adaptive software. It only does not support the distribution of feedback loops (R1.7) and it supports only partially the variability (R1.3) and concurrent execution (R6.1) of feedback loops. We excluded distribution and partially concurrency from the EUREMA language since they are cross-cutting concerns and might have a major impact on the whole language. Particularly, distribution and concurrency could make the language more complex and therefore less easier to use. For instance, supporting concurrency for model operations also calls for means to synchronize concurrently executing operations. Moreover, modeling comprehensively the variability of feedback loops can easier be achieved by using another language (*e. g.*, for feature modeling) rather than extending EUREMA.

Table 11: EUREMA's Full (✔), Partial (✓/✗), or No (✘) Coverage of the Requirements.

	R1							R2		
	Explicit Feedback Loops	Modularity of Feedback Loops	Variability of Feedback Loops	Intra-Loop Coordination	Triggers of Feedback Loops	Inter-Loop Coordination	Distribution of Feedback Loops	Capturing Runtime Models	Openness for Languages of Runtime Models	Abstract Runtime Models
	R1.1	R1.2	R1.3	R1.4	R1.5	R1.6	R1.7	R2.1	R2.2	R2.3
EUREMA	✔	✔	✓/✗	✔	✔	✔	✘	✔	✔	✔

	R3			R4		R5		R6		
	Connecting Feedback Loops and the Adaptable Software	Parameter and Structural Adaptation	Dynamic Sensors and Effectors	Adaptable and Adapting Feedback Loops	Procedural and Declarative Reflection on Feedback Loops	Maintenance of Self-Adaptive Software	Coordination of Self-Adaptation and Maintenance	Concurrent Execution of Feedback Loops	State- and Event-Based Execution	Runtime-Efficient Execution
	R3.1	R3.2	R3.3	R4.1	R4.2	R5.1	R5.2	R6.1	R6.2	R6.3
EUREMA	✔	✔	✔	✔	✔	✔	✔	✓/✗	✔	✔

Nevertheless, covering most of the requirements, EUREMA supports a wide range of self-adaptive software, for instance, employing a single feedback loop, multiple feedback loops within a layer, or layered feedback loops for adaptive control. In this context, EUREMA allows engineers to freely express a specific self-adaptive software, for instance, concerning the number of layers and feedback loops or the structure of feedback loops, as long as the software falls within the supported scope, that is, self-adaptive software adopting a layered architecture, following the external approach, influenced by the MAPE-K blueprint, and using runtime models. Thereby, most domain concepts are directly reflected as language concepts in EUREMA. This supports the engineering of self-adaptive software since engineers can easily identify known domain concepts in EUREMA and use the corresponding language concepts to express a self-adaptive software with its feedback loops.

We will come back to the requirements for engineering self-adaptive software when discussing related work in the next chapter.

9.7 SUMMARY

In this chapter, we have evaluated EUREMA from different angles to provide evidence for the quality, expressiveness, effectiveness, costs, performance, and capabilities of EUREMA.

For this purpose, we have carefully designed the EUREMA language taking into account 26 design guidelines proposed by Karsai et al. [241, p. 7] who argue that following these guidelines supports achieving a "better quality of the language design and a better acceptance among its users". Following and assessing these guidelines for EUREMA give us some evidence that we have achieved a good language design (*cf.* Section 9.1).

Moreover, we investigated the expressiveness of the EUREMA language by examples. That is, we applied EUREMA to three third-party approaches that have different characteristics. We have shown that the language is expressive enough to describe Rainbow, DiVA, and PLASMA (*cf.* Section 9.2). This gives us some evidence that the EUREMA language is applicable beyond our application example to a wider range of self-adaptation problems.

Additionally, we have instantiated EUREMA to several self-adaptation problems for our mRUBiS application example and conducted experiments with the instance. Such an instantiation is suitable for demonstrating the feasibility of an approach and of the developed solution [219]. Therefore, we obtained evidence that EUREMA can be effectively applied to self-repair (*cf.* Section 9.3.2), to self-optimize (*cf.* Section 9.3.3), to coordinate the self-repair and self-optimization (*cf.* Section 9.3.4), and to achieve an adaptive control solution for self-repairing (*cf.* Section 9.3.5) the running mRUBiS. Consequently, we effectively applied EUREMA to dynamically adapt a running software system. Thus, we think that EUREMA can be effectively applied to different self-adaptation problems that consider either single, multiple coordinated, or layered feedback loops as well as to different self-adaptation capabilities such as self-repair and self-optimization.

To complement the instantiation of EUREMA for the mRUBiS application example, we conducted a comparative study. We compared solutions that we developed with EUREMA to solutions that students developed without EUREMA. In the study, we analyzed the development costs and runtime performance of the different solutions. Based on the resulting data, we concluded that the EUREMA solutions provide a good compromise concerning development costs and runtime performance (*cf.* Section 9.4).

Moreover, we assessed the EUREMA prototype by discussing a set of quality attributes (flexibility, extendibility, usability, reusability, performance, scalability, and testability) in Section 9.5. We discussed most of these attributes qualitatively, which makes the assessment rather subjective. Thus, the evidence that we obtained from the assessment should be considered as complementary to the other evaluation efforts we conducted. However, we provided quantitative data for the assessing the performance and concluded that the EUREMA interpreter causes only a negligible overhead compared to a compiled solution for executing a feedback loop. Therefore, we obtained some solid evidence that the EUREMA interpreter achieves a good performance (*cf.* Section 9.5.5).

Finally, we discussed the capabilities of EUREMA by comparing them against the requirements for engineering self-adaptive software in Section 9.6. This discussion complements the concrete experiments and examples we have conducted and created with conceptual viewpoints. Collecting all the evidence we obtained from the evaluation, we are confident that we have achieved with EUREMA a high-quality, expressive, effective, and competitive (with respect to development costs and runtime performance) approach for developing and executing feedback loops in self-adaptive software.

10 RELATED WORK

In this chapter, we discuss the state of the art in engineering self-adaptive software that is related to EUREMA. In general, there exists a lot of work on feedback loops to control systems such as in autonomic computing that applies control theory to parameter adaptation of software systems [152, 215, 251]. However, self-adaptation is often conducted at the architectural level (*cf.* [86, 169, 227, 258, 286, 334, 335]), which calls for dynamically and structurally changing the software architecture [292]. Such changes prevent a direct application of control theory and requires other means for engineering self-adaptive software. Popular means are development approaches that use frameworks and models [370].

Therefore, we discuss such approaches in the following by using the requirements introduced in Chapter 3. These requirements allow us to compare these approaches to EUREMA. In this context, we focus on approaches that provide *generic* support for developing self-adaptive software by addressing the whole feedback loop. Consequently, most of these approaches can be considered as alternatives to EUREMA. Before that, we discuss related work that proposes *specific* solutions for individual feedback loop activities and self-adaptation problems. This work can be considered as complementary to EUREMA.

10.1 COMPLEMENTARY APPROACHES TO EUREMA

In the following, we discuss approaches that are complementary to EUREMA while distinguishing between techniques, languages, and related research fields.

10.1.1 *Techniques*

In general, EUREMA is well in line with the seminal work by Oreizy et al. [334–336] who propose an architecture-based approach to self-adaptive software that consistently evolves the adaptable software and the architectural model of the software. Moreover, they consider model-based adaptations to be performed automatically by scripts or manually in a shell. The architectural model is comparable with the architectural reflection models (that can be) used by EUREMA feedback loops (*cf.* Section 8.2). Similarly, we consider autonomic self-adaptation controlled by feedback loops (*cf.* Section 5.1) and off-line adaptation controlled by engineers (*cf.* Section 5.4). However, we support the engineering of feedback loops, which is not specifically addressed by Oreizy et al. [334–336].

As with any software, developing self-adaptive software starts with requirements. Consequently, the development of self-adaptive software must also be addressed from a requirements engineering perspective [111]. In this context, work has been done to characterize and capture requirements that reflect the uncertainty inherent in self-adaptive software. For instance, Whittle et al. [440] propose the RELAX language to specify requirements taking explicitly uncertainty into account. Qureshi and Perini [351, 352] investigate the use of goal-oriented techniques to model requirements. Silva Souza et al. [388] propose awareness requirements that constraint and refer to the behavior of other requirements, typically of the adaptable software, and that eventually determine the feedback loop functionality. We consider such work as complementary to EUREMA. Thus, we assume that the require-

ments have been elicited, analyzed, and specified before engineers operationalize these requirements and use EUREMA to design and develop the feedback loops.

For the initial design of a self-adaptive software, reference models and architectures have been proposed. Examples are the MAPE-K model for feedback loops [246], the architectural reflection pattern [96], and the three-layer architecture for self-managed systems [258], which we discussed in Section 2.2.3. To explore different reference models and architectures, developers can use FORMS [435] that provides a well-defined and unified vocabulary for describing and reasoning about architectural aspects of self-adaptive software. Thus, engineers can create and analyze alternative architectural blueprints with FORMS, which supports early design decisions. To realize a particular blueprint, an engineering approach such as EUREMA is required. Consequently, FORMS and EUREMA are complementary approaches. The realization of a blueprint can be supported by design patterns for self-adaptive software that have been proposed by Ramirez and Cheng [355].

Using EUREMA, we developed examples that follow the MAPE-K model (*cf.* Sections 5.1 and 9.3.2) and the three-layer reference architecture (*cf.* Sections 5.3 and 9.3.5). Concerning the three-layer architecture, we have not realized a goal management layer with sophisticated planning techniques since our aim was to support and demonstrate the possibility of using stacked feedback loops in three or more layers with EUREMA. However, EUREMA is open for such techniques as engineers have to provide user-defined implementations for individual adaptation activities. Consequently, we consider techniques for individual activities as complementary to EUREMA.

Concerning the monitoring activity, we developed an advanced solution to maintain abstract reflection models using incremental model synchronization techniques (*cf.* Section 8.2 for a specific discussion of related work for the solution), which, however, is limited with respect to adapting the monitoring mechanism. We only considered the activation of sensors to enable performance monitoring by the self-optimization feedback loop. In contrast, the self-repair feedback loop does not activate these sensors as it does not require performance data. In this context, there exist complementary work on adaptive monitoring that can be beneficial. For instance, Dawson et al. [128] realize monitoring with proxies in the adaptable software that can be dynamically composed to adjust the intensity of monitoring. Similarly, Ehlers et al. [144] dynamically adjust the monitoring coverage based on OCL rules specifying conditions when sensors should be (de)activated. Ramirez et al. [356] use a genetic algorithm to generate sensor configurations that determine the frequency of monitoring to balance costs and accuracy of monitoring. Moreover, to continuously improve the quality of runtime models used in a feedback loop, monitoring can be complemented with learning techniques to infer knowledge from sensed data [147, 151, 406].

For the analyzing and planning activities of the feedback loops we developed with EUREMA, we employ a rule-based approach that defines failures or performance issues that can be directly identified and localized as faults or bottlenecks in the architectural reflection model. Moreover, it defines adaptation strategies that can address these failures or issues (*cf.* Sections 9.3.2 and 9.3.3). In this context, advanced solutions to analyze failures or performance issues and to plan corresponding adaptations exist and can be integrated into an EUREMA-based feedback loop. For instance, Casanova et al. [104] provide a technique to diagnose architectural failures and to identify correlated faults among components. These components can be the target of an adaptation. Such a technique supports dynamically localizing faults in the architecture while we prescribe the locations of faults depending on the failures. Similarly, we use simple techniques to analyze the performance of the adaptable software, basically by comparing the measured response time to given

thresholds. In contrast, Huber et al. [228] use and solve analytical models that supports detailed performance analysis and prediction. Thereby, they dynamically select the level of granularity of the models to trade-off accuracy and costs of the analysis and prediction. Analytical models further pave the way for runtime model checking to verify requirements such as reliability and performance, which supports analysis and planning [97, 98, 153].

To plan an adaptation, we precisely defined the adaptation strategies that should be executed given an identified failure or performance issue. In this context, there exists sophisticated techniques that generate target architectures/configurations of the adaptable software to optimize the quality of the software as defined by a utility function. For this purpose, heuristics [295] and genetic algorithms [341, 359] can be used. Other techniques to automatically obtain target architectures is model checking [407, 410] and redesigning the architecture based on provided design decisions and the goals of the software [109]. Similarly, controller synthesis techniques [138, 315, 338] are applicable for behavioral instead of architectural adaptation. Recently, we started extending the planning approach that we employ in the EUREMA-based feedback loops to self-repair and self-optimize mRUBiS by dynamically selecting architectural adaptation strategies based on utilities and costs [7].

Concerning the execution of an adaptation to the adaptable software, we execute an adaptation strategy on the reflection model that is automatically and incrementally synchronized with the adaptable software (*cf.* Section 8.2). Such a model-based adaptation corresponds to the idea of runtime models (*cf.* Section 2.1.5) and is well in line with the work by Oreizy et al. [334, 335]. However, we execute the individual steps of an adaptation in a predefined order without taking resource limitations, quality of service, or failures into account when reconfiguring the adaptable software. In this context, Silva and Lemos [389] and Ramirez et al. [357] propose techniques to generate suitable adaptation paths depending on the current conditions of the resources or quality of service properties. Boyer et al. [82] propose a robust reconfiguration protocol with recovery policies to handle failures of applying an individual adaptation step during a reconfiguration. Such techniques for executing an adaptation can complement the EUREMA-based feedback loops we developed.

Summing up, we consider the techniques that we discussed in this section as complementary to EUREMA since they can be integrated into EUREMA-based feedback loops. Besides the feedback loops, the techniques can also benefit from the integration. For instance, Calinescu et al. [98] realize a feedback loop for dynamic service selection with formal models and model checking by gluing together a series of tools. In this context, a systematic development of feedback loops as provided by EUREMA can be beneficial.

10.1.2 *Languages*

As we propose with EUREMA a language to specify feedback loops, we review in the following complementary languages that have been proposed for developing self-adaptive software and particularly adaptation mechanisms. All these languages and their related approaches have in common that they are not used to specify feedback loops.

For instance, Zhang and Cheng [446] propose a model-based development approach for dynamically adaptive software, which focuses on modeling, validating, and verifying the behavior of adaptive programs. For this purpose, each steady-state behavior of the program is modeled with a petri net. Switching the program from one to another steady-state behavior (*i. e.*, switching from a source to a target petri net) is considered as an adaptation. It is modeled with a further petri net capturing transitions from a source to a target petri net. Thus, the behavior before, during, and after adaptation is modeled and can be verified by

model checking invariants. The verification has been optimized by using a modular model checking technique [447]. Finally, generating and testing of prototypes are supported.

Petri nets are further used in other approaches. For instance, Camilli et al. [100] extend the work of Zhang and Cheng [446] to address real-time behavior. Ding et al. [137] extend petri nets with an adaptation transition that is associated to a neural network and that performs local decisions for branching the behavior. Cardozo et al. [102] use petri nets to specify the behavior of adaptive software at the implementation level while coupling the software to the context-oriented programming paradigm. Related to petri nets, the use of communicating sequential processes to specify adaptive programs has been proposed [59]. All these approaches have in common that they target the specification and verification of the adaptive behavior of the software, that is, they describe how the adaptable software changes but not how these changes are controlled. Thus, they do not address the specification and development of explicit feedback loops that control the adaptation. Therefore, they are complementary to EUREMA as they allow us to specify and verify the overall adaptive behavior while the feedback loops can subsequently be developed with EUREMA. Moreover, they often follow the internal approach to self-adaptation (*cf.* Section 2.2.3) by jointly modeling the behavior of the whole system while EUREMA follows the external approach by modeling the adaptation engine with the feedback loops.

Besides these approaches that use one language to model the behavior of the overall adaptive software, specific languages have been proposed to express adaptation models. Such models govern the adaptation by defining the analysis, decision-making, and planning of self-adaptation (*cf.* Section 4.1 and [27]). Such adaptation models are often policies typically expressed as ECA rules (*cf.* [230]). Such rules specify events whose occurrences trigger adaptation actions if certain conditions are met. In this context, the actions can also be triggered externally such that no events are specified. Accordingly, languages to express such rules have been proposed and used for self-adaptive software systems [114, 176, 279, 283, 314, 421]. Such languages are closely related to languages for dynamic architectures, which specify architectural changes by considering the initiation, selection, and implementation of changes to an architecture (*cf.* [86]). To execute rules, policy engines exist that are connected to the adaptable software by user-defined sensors and effectors [37, 193]. Moreover, to execute an adaptation plan as a set of actions in a coordinated manner, process technologies have been proposed by Silva and Lemos [389] and Valetto and Kaiser [416] who, however, do not consider the whole feedback loop as an executable process—in contrast to EUREMA (*cf.* Section 4.2)—but only the plan executed by the feedback loop.

Thus, such languages express rule-based adaptation models but they do not specify the feedback loops controlling the adaptation as it is addressed by EUREMA. Therefore, these languages are complementary to EUREMA. Particularly, we used Story Diagrams (SDs) to create rule-based adaptation models that are used within the EUREMA-based feedback loops (*cf.* Section 9.3.2). These models are similar to ECA rules except that SDs are a visual language and based on graph transformations. Besides SDs, other graph transformation languages have been proposed and used for expressing adaptation rules [401, 412, 432].

Besides creating the rules manually, they can be automatically generated from descriptions of goals, adaptation options, and impact of options on goals [363] or required adaptation actions can be generated from domain-specific constraints [398]. Such approaches use search-based capabilities to derive adaptation plans similar to the planning techniques discussed in the previous section that generate suitable target configurations. In this context, rule- and search-based adaptation models can be combined by using rules to reduce the configuration space that is subsequently searched for an optimal target configuration [159].

EUREMA supports integrating rule- and/or search-based adaptation models. Our decision to use SDs is motivated by an assessment of this language to express and execute adaptation models in a models@run.time setting [27] and by our goal to use a basic but flexible approach that allows us to evaluate EUREMA in different settings (*cf.* Section 9.3).

Finally, languages or the use of models have been proposed for adaptive systems in specific domains. For instance, Bocanegra et al. [80] propose a visual DSL to design adaptive web-based systems. Similar to rules, the resulting models capture how the information, navigation, and presentation of a website are adapted to match user or context preferences without specifying the feedback loop controlling the adaptation. De Wolf and Holvoet [130, 131] use UML activity diagrams to model information flows in emergent self-organizing systems, that is, the interactions between agents before realizing the specific coordination mechanisms (*e. g.*, digital pheromones). The flows should form a feedback loop such that agents performing actions get feedback from other agents and stimulate or inhibit each other. Such closed feedback loops eventually realize the self-organization of agents. However, the activity diagrams do not make the feedback loops of information flows explicit. In contrast, developers have to manually analyze the flow of actions and objects in the activity diagram to identify proper loops. Thus, this notion of feedback loops is implicit and addresses information flows between agents but not self-adaptation. Thus, both approaches [80, 130, 131] do not target the modeling of feedback loops for self-adaptation.

10.1.3 *Related Research Fields*

In this section, we briefly discuss research fields that are related to EUREMA. One field is autonomic computing [230, 246] that primarily focuses on the hardware, operating system, network, middleware layers of software systems while work on self-adaptive software such as EUREMA focuses on the middleware and application levels (*cf.* [370]). Thus, autonomic computing rather adopts a system than a software perspective on self-adaptation.

Similarly, the organic computing field primarily takes a system perspective while considering a feedback loop comprising an observer and a controller [374]. However, the work on the software part is based on the agent-oriented paradigm and in contrast to EUREMA, it does not explicitly address and specify the feedback loops for self-adaptation [199, 313].

Finally, an area in the software engineering field that is related to EUREMA is the work on controller synthesis (*e. g.*, [85, 138]). Given a behavioral model of the environment and a goal, controller synthesis generates a behavioral model (*i. e.*, the controller) for the software. The behavioral models are typically some form of transition systems such as state machines. The generated model is directly executed and while traversing the states, it invokes methods on the software in a controlled manner to achieve the goals in the given environment. Besides synthesizing controllers, controller can also be dynamically updated [181, 315, 338]. A similar approach is proposed by Ghezzi et al. [183] who transform a user-defined activity diagram into a markov decision process that is directly executed to drive the execution of the software. Moreover, using model-checking techniques, the process is dynamically manipulated to achieve an execution that meets the goals. Consequently, such approaches are similar to EUREMA with respect to keeping executable models alive at runtime as well as directly executing and dynamically updating these models. While the EUREMA models specify and realize feedback loops for self-adaptation, the behavioral models (*i. e.*, the controllers) in the other approaches specify and realize the coordination of individual behavior for the business logic. Therefore, these other approaches and EUREMA target different problems while sharing the property of using executable runtime models.

10.2 ALTERNATIVE APPROACHES TO EUREMA

In this section, we discuss approaches that provide *generic* support for developing self-adaptive software by addressing the whole feedback loop. Consequently, such approaches are alternatives to EUREMA. In the following, we discuss such approaches using the requirements introduced in Chapter 3. These requirements allow us to compare these approaches to EUREMA. Afterwards, we provide a summarizing discussion.

10.2.1 *Coverage of the Requirements*

In general, approaches for developing self-adaptive software often provide a framework and use some form of models [370]. Overall, we identified 26 of such approaches in the literature that generically support the development of self-adaptive software and cover the whole feedback loop. Thus, these approaches are alternatives to EUREMA and we discuss each of them with respect to their coverage of the requirements introduced in Chapter 3.

In this context, we point out that although these approaches address the whole feedback loop, some of them focus more on certain parts of the feedback loop such that they do not completely support the other parts in a generic manner. Consequently, some approaches do not have the claim of equally addressing all parts of a feedback loop. In this case, they do not explicitly address those requirements that are not relevant for their specific design goals, which therefore affects their overall coverage of the requirements. For instance, PLASMA [410] focuses on the planning part using a planning-as-model-checking mechanism, FUSION [145, 147] integrates learning mechanisms to continuously improve the decision-making step, or Descartes [228, 229, 254] provides comprehensive runtime models of the adaptable software as well as sophisticated analysis and planning mechanisms for the on-line prediction of performance. Nevertheless, we included such approaches in the discussion as they still address the whole feedback loop while the specific mechanisms that are the major research contributions of these approaches are rather complementary to EUREMA and can thus be integrated into EUREMA-based feedback loops.

An overview of the discussion is given in Table 12 listing the 26 approaches[1] with the years when they have been published and showing how well each of them covers the sub-requirements for Feedback Loops (R1), Runtime Models (R2), Sensors and Effectors (R3), Layered Architectures (R4), Off-line Adaptation (R5), and Execution (R6). A detailed discussion of the requirements can be found in Chapter 3. We rate the individual approaches by distinguishing full (✔), partial (✓/✗), no (✘), and unknown (**?**) coverage of the individual requirements. In this context, we point out that the rating should be seen as a rough indicator of how well an approach fulfills the requirements while a more detailed discussion is provided by the descriptions of each approach. Moreover, we point out that such a rating and discussion are often difficult to make. A major reason for this is that the papers presenting the approaches often do not provide sufficient details and data as well as do not report on limitations and comprehensive assessments of the approaches (*cf.* [434]).

In the following, we individually discuss the 26 alternative approaches to EUREMA based on their coverage of the requirements in the same order as they are listed in Table 12.

Tune. Tune, the successor project of Jade, aims for the model-driven autonomic management of systems such as server landscapes [120, 201]. Each system element is wrapped as a Fractal software component providing sensor and effector interfaces for its management.

1 If an approach has no dedicated name, we use the name of the authors proposing the approach.

Table 12: Related Work and their Full (✔), Partial (✓/✗), No (✘), or Unknown (?) Coverage of the Requirements for Feedback Loops (R1), Runtime Models (R2), Sensors and Effectors (R3), Layered Architectures (R4), Off-line Adaptation (R5), and Execution (R6).

Approach	R1							R2			R3			R4		R5		R6		
	Explicit Feedback Loops	Modularity of Feedback Loops	Variability of Feedback Loops	Intra-Loop Coordination	Triggers of Feedback Loops	Inter-Loop Coordination	Distribution of Feedback Loops	Capturing Runtime Models	Openness for Languages of Runtime Models	Abstract Runtime Models	Connecting Feedback Loops and the Adaptable Software	Parameter and Structural Adaptation	Dynamic Sensors and Effectors	Adaptable and Adapting Feedback Loops	Procedural and Declarative Reflection on Feedback Loops	Maintenance of Self-Adaptive Software	Coordination of Self-Adaptation and Maintenance	Concurrent Execution of Feedback Loops	State- and Event-Based Execution	Runtime-Efficient Execution
	R1.1	R1.2	R1.3	R1.4	R1.5	R1.6	R1.7	R2.1	R2.2	R2.3	R3.1	R3.2	R3.3	R4.1	R4.2	R5.1	R5.2	R6.1	R6.2	R6.3
Tune [120, 201] (2008)	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘	✔	✓/✗	✘	✘	✘	✘	✘	✘	✓/✗	?
MADAM [161, 175] (2006)	✘	✘	✘	✘	✔	✓/✗	✓/✗	✓/✗	✘	✔	✔	✔	✓/✗	✘	✘	✓/✗	✘	✓/✗	✓/✗	✔
MUSIC [160, 202, 367] (2009)	✘	✘	✘	✘	✔	✓/✗	✓/✗	✓/✗	✘	✔	✔	✔	✓/✗	✘	✘	✓/✗	✘	✓/✗	✓/✗	✔
ASSL [419, 420] (2009)	✘	✓/✗	✘	✘	✓/✗	?	✓/✗	✘	✘	✘	✔	?	✘	✘	✘	✘	✘	✓/✗	✓/✗	?
Genie [63] (2009)	✘	✓/✗	✘	✘	✔	✘	✘	✓/✗	✘	✔	✓/✗	✔	✘	✘	✘	✓/✗	✘	✘	✓/✗	?
Rainbow [113–115, 170] (2004)	✘	✓/✗	✘	✘	✓/✗	✘	✘	✔	✘	✔	✔	✔	✔	✘	✘	✓/✗	✘	✘	✓/✗	✔
GRAF [40, 133] (2011)	✘	✓/✗	✘	✘	✓/✗	✘	✘	✔	✓/✗	✘	✔	✔	✔	✘	✘	✓/✗	✘	✘	✓/✗	✔
DiVA [306–309] (2008)	✘	✓/✗	✘	✘	✔	✘	✘	✔	?	✔	✔	✔	✘	✘	✘	✓/✗	✘	✘	✓/✗	✓/✗
PLASMA [410] (2010)	✘	✓/✗	✘	✘	✘	✓/✗	✓/✗	✘	✘	✘	✔	✔	✘	✓/✗	✓/✗	✓/✗	✘	✓/✗	✓/✗	✓/✗
FUSION [145, 147] (2010)	✘	✓/✗	✘	✘	✘	✘	✘	✔	✘	✔	✓/✗	✔	✘	✘	✘	✘	✘	✘	✔	✓/✗
Alférez and Pelechano [39] (2012)	✘	✓/✗	✘	✘	✓/✗	✘	✘	✔	✘	✔	✓/✗	✔	✘	✓/✗	✘	✘	✘	✘	✓/✗	?
Chen et al. [109] (2014)	✘	✓/✗	✘	✘	✓/✗	✘	✘	✔	✘	✔	✓/✗	✔	✘	✓/✗	✘	✘	✘	✘	✓/✗	✓/✗
J3 [438] (2008)	✘	✓/✗	✘	✓/✗	✓/✗	✘	✓/✗	✘	✘	✘	✔	✓/✗	✘	✘	✘	✘	✘	✓/✗	✓/✗	?
FESAS [263, 264, 365] (2015)	✘	✓/✗	✓/✗	✓/✗	✘	✓/✗	✓/✗	✘	✘	✘	✔	✔	✘	✘	✘	✘	✘	?	✓/✗	?
Descartes [228, 229, 254] (2014)	✘	✓/✗	✓/✗	✘	?	✘	✘	✔	✘	✔	✓/✗	✔	✘	✘	✘	✘	✘	✘	✔	✓/✗
StarMX [48] (2009)	✘	✔	✘	✔	✔	✓/✗	✘	✘	✘	✘	✔	✔	✘	✘	✘	✘	✘	✓/✗	✓/✗	?
Solomon et al. [393] (2008)	✘	✔	✘	✓/✗	✘	✘	✘	✘	✘	✘	✔	✓/✗	✘	✘	✘	✘	✘	✘	✓/✗	?
Berkane et al. [69] (2015)	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	?	?
MechatronicUML [187, 218] (2004)	✘	✔	✘	✔	✓/✗	✔	✓/✗	✘	✘	✘	✔	✓/✗	✘	✓/✗	✓/✗	✘	✘	✓/✗	✓/✗	?
SCA-ASM [44–46] (2015)	✘	✔	✘	✔	✘	✔	✓/✗	✘	✘	✘	✔	✓/✗	✘	✘	✘	✘	✘	✘	✘	✘
ActivFORMS [232, 437] (2014)	✘	✔	✓/✗	✔	✔	✘	✘	✓/✗	✘	✔	✔	✓/✗	✘	✓/✗	✓/✗	✓/✗	✔	✓/✗	✔	✔
SimSOTA [34, 35] (2013)	✓/✗	✔	✘	✔	✘	✔	✘	✘	✘	✘	✔	✓/✗	✘	✓/✗	✘	✘	✘	✓/✗	✓/✗	?
Control Loop UML [210] (2010)	✔	✘	✘	✘	✘	✓/✗	✓/✗	✘	✘	✘	✔	✔	✘	✔	✘	✘	✘	✘	✘	✘
FAME [204] (2016)	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✔	✓/✗	✘	✘	✘	✘	✘	✘	✘	✘
ACML [280, 281] (2013)	✔	✓/✗	✘	✔	✔	✘	✓/✗	✘	✘	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘
ACTRESS [260–262] (2012)	✔	✔	✘	✔	✔	✘	✓/✗	✘	✘	✘	✔	✓/✗	✘	✔	✓/✗	✘	✘	✓/✗	✓/✗	?
EUREMA (2012)	✔	✔	✓/✗	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✓/✗	✔	✔

Thus, the whole system is managed as a component-based architecture. To employ the Tune framework, developers use four provided modeling languages to individually specify the system architecture, the sensor and effector interfaces, the deployment, and the policies that are similar to ECA rules. Policies consist of UML state machines describing the life cycles of system elements and UML activities defining reconfiguration actions operating on effectors. Transitions in a state machine are triggered by monitored events and execute associated actions. For execution, Tune maps the models to the Fractal infrastructure.

Using such policies to specify adaptation, the feedback loop is not made explicit (R1.1). Tune treats the feedback loop as a monolithic component that executes the policies. Thus, modularity (R1.2), variability (R1.3), and intra-loop coordination (R1.4) are not addressed. Triggers are defined by sensor events, for instance, notifying about a server crash (R1.5). Tune focuses on a single, centralized feedback loop such that it does not address the inter-loop coordination (R1.6) and distribution of feedback loops (R1.7).

Runtime models are not used and therefore not captured (R2). Supporting the specification of sensor and effector interfaces, Tune addresses the connection of the feedback loop to the adaptable system (R3.1). Adaptation is conducted by actions that invoke methods on effectors, for instance, to start a server. In contrast to parameter adaptation, structural adaptations that modify the architecture are not supported (R3.2). Likewise, dynamic sensors and effectors are not supported as they are part of the architecture description (R3.3).

Tune does not address stacked feedback loops in layered architectures (R4), off-line adaptation (R5), and the concurrent execution of feedback loops (R6.1). The execution is solely driven by events without using and maintaining any knowledge (state) of the system (R6.2). The runtime efficiency and scalability of the feedback loop are not investigated (R6.3).

MADAM. MADAM is a model-driven development approach for self-adaptive mobile applications [161, 175]. It provides a UML-based architectural language to specify the component types of an application, multiple realizations for each type with different quality of service properties to obtain variability, the context, and the resources. From this model, code is generated for the MADAM middleware. At runtime, the middleware observes the context and it uses a utility function to find the most promising variant of the application for the current context by selecting appropriate realizations for the component types. The middleware then reconfigures the application to enact the selected variant.

The architectural model describes the variability of the adaptable software while the middleware realizes the feedback loop that is therefore hidden in the implementation (R1.1). Modularity (R1.2), variability (R1.3), and intra-loop coordination (R1.4) of feedback loops are not addressed since developers can only specify the utility function but not the individual feedback loop elements. Triggers are defined by modeling context entities whose changes initiate the feedback loop (R1.5). The inter-loop coordination is partially addressed since MADAM supports a specific coordination scheme, in which one feedback loop realizes the decision making and delegates the execution to other feedback loops running on other nodes. Other schemes and multiple feedback loops on one node are not supported (R1.6). Thus, MADAM supports the distribution of feedback loops across nodes although there does not seem to be any modeling support for describing the distribution (R1.7).

The architectural model of the adaptable software is used as a code-based and therefore implicit runtime model by the middleware (R2.1). Other languages to express the model are not supported (R2.2). Tackling the architectural level, the model is abstract (R2.3).

The connection of the feedback loop to the adaptable software is realized by the middleware and developers can implement their own sensors and effectors (R3.1) that may realize

parameter and structural adaptation (R3.2). Context sensors can be dynamically added to the middleware, which is, however, not discussed for system sensors and effectors (R3.3).

Stacked feedback loops in layered architectures are not supported (R4). The maintenance of the self-adaptive software is partially addressed since developers can dynamically add and remove realizations of component types. However, such changes do not target the adaptation logic (R5.1) and they do not seem to be coordinated with self-adaptation (R5.2).

The concurrent execution is partially addressed for feedback loops deployed on different nodes while there exists only one loop per node. Concurrency within one loop is not supported (R6.1). The execution of the feedback loop is driven by events notifying about context changes. Any state of the adaptable software is not used for the reasoning except for obtaining a reconfiguration script by comparing the current with the target architecture of the application (R6.2). MADAM achieves a runtime-efficient execution for small-sized applications with a limited variability, which, however, can still take "the order of a few seconds" [175, p. 413]. In this context, the authors acknowledge scalability problems (R6.3).

MUSIC. MUSIC is the successor project of MADAM to extend the capabilities to ubiquitous applications [160, 202, 367]. Particularly, an application deployed on a node may use external services that are provided by applications on other nodes instead of local components. MUSIC takes such external service into account when searching for a variants that fits best the current context. The development methodology and how self-adaptation is realized in MUSIC is very similar to MADAM. The major extensions have been done to the middleware to cope with ubiquitous applications. Consequently, MUSIC addresses the requirements similarly to MADAM such that we refer to the discussion of MADAM.

ASSL. ASSL is the autonomic system specification language that supports modeling of self-managing embedded systems and generating Java skeletons [419, 420]. Using ASSL, developers specify autonomic elements that control managed elements. The former are specified by policies (condition-action rules) and the latter by sensor and effector interfaces (to observe metrics and perform actions). The interaction among both elements is defined by events. Multiple autonomic elements can be specified and coordinated by a central autonomic element. Using the specification, Java code is generated and manually completed by developers. The generated code realizes a feedback loop for each autonomic element.

ASSL does not make the feedback loops explicit since the self-adaptation is expressed only by policies (R1.1). Thus, modularity is only addressed for the policies (R1.2) without covering the variability (R1.3) and intra-loop coordination (R1.4) of feedback loops at the model level. Only the generated code decomposes a feedback loop into different steps to execute the policies. The feedback loops can only be triggered periodically (R1.5). The coordination of multiple feedback loops is mentioned but not detailed or exemplified (R1.6). Multiple autonomic elements can be specified and distributed in the architecture but not in the deployment. The distribution is not explicitly specified (R1.7).

ASSL does not consider runtime models (R2). Specifying sensor and effector interfaces, the connection of a feedback loop to the adaptable system is covered (R3.1). The actions of a feedback loop use the effectors to initiate some behavior. It is unclear whether such a behavior performs an adaptation (R3.2). Dynamic sensors and effectors (R3.3), the stacking of feedback loops in layered architectures (R4), and off-line adaptation (R5) are not covered. The feedback loops seem to be executed concurrently and the activities of one loop sequentially (R6.1). The execution is solely driven by events (R6.2). The runtime efficiency of the execution is not evaluated (R6.3).

Genie. Genie promotes the model-driven development of middleware-based self-adaptive software [63]. Developers specify configurations of the adaptable software and adaptations in terms of conditional transitions between these configurations. Using these models, Genie generates configuration files and ECA rules for the software running on the middleware.

The feedback loop is defined within the middleware and therefore implicit (R1.1). Modularity is only addressed for the ECA rules (R1.2) but without variability (R1.3). The intra-loop coordination is implicit in the middleware (R1.4). Triggers are specified by events that notify about context changes to initiate the ECA rules (R1.5). Supporting a single feedback loop, Genie does not cover the inter-loop coordination (R1.6) and distribution (R1.7).

Genie captures the ECA rules as runtime models without using a reflection model (R2.1). These models must be expressed in a language specific to the middleware (R2.2) targeting the architectural level (R2.3). The connection of the feedback loop to the adaptable software is prescribed by the middleware (R3.1) that provides sensors and effectors for parameter and structural adaptation (R3.2). Dynamic sensors and effectors are not considered (R3.3).

Stacking feedback loops in layered architectures is not supported (R4). However, Genie allows developers to add newly generated component configurations and ECA rules to the middleware at runtime, which partially supports the maintenance. It does not support extensive changes (*e. g.*, changing the feedback loop structure) (R5.1) and the coordination of self-adaptation and maintenance (R5.2). Genie does not support the concurrent execution of feedback loops (R6.1). The execution is driven by events notifying about context changes but without exploiting the benefits of a reflection model capturing the state (R6.2). The authors of Genie do not report about the runtime efficiency of the feedback loop (R6.3).

Rainbow. Rainbow aims for the cost-effective development of self-adaptive software by providing a reusable infrastructure [113–115, 170]. It realizes a single fixed feedback loop with the MAPE activities operating on an architectural model of the adaptable software. We described this feedback loop in detail in Section 9.2.1.

The feedback loop remains implicit in the infrastructure (R1.1) because Rainbow offers customization points targeting only the models used in the loop (*i. e.*, rules defining constraints for the architectural model, as well as strategies, tactics, and operators defining the adaptation) but no means to design the entire feedback loop. Thus, Rainbow partially addresses the modularity based on these models (R1.2) but it does not consider specific means for the variability of these models (R1.3). Rainbow prescribes the intra-loop coordination since the customization points do not target the activities with their structuring and processing of knowledge, which form the feedback loop (R1.4). It further does not provide an explicit mechanism for triggers (R1.5), which have to be realized within the implementation of probes and gauges that sense the adaptable software and update the architectural model. Supporting a single feedback loop, Rainbow does not address the inter-loop coordination (R1.6) and distribution (R1.7). Initial work has been done for an abstraction layer between the adaptable software and the feedback loops, through which multiple loops have consistent access to the software but without coordinating these loops [117].

While Rainbow captures the runtime models (*e. g.*, the architectural model, analysis rules, and adaptation strategies) (R2.1), it is not open for but prescribes their languages (R2.2) (*e. g.*, Acme for the architectural model, Acme constraints for the rules, and Stitch for the strategies). These languages support abstract models (R2.3). Moreover, Rainbow addresses the connection between the adaptable software and the feedback loop since it defines interfaces for probes (sensors) and effectors that have to be implemented by the developer and that are used by infrastructure (R3.1). Thus, developers can realize parameter and structural adaptation (R3.2) and dynamic sensors and effectors (R3.3) (*cf.* [173]).

Rainbow does not address stacked feedback loops (R4). Nevertheless, it allows developers to update the rules and strategies at runtime, which supports the maintenance of these models but not more extensive changes affecting the number or structure of feedback loops (R5.1). The coordination of self-adaptation and maintenance is not covered (R5.2).

Concerning the execution, Rainbow does not address the concurrent execution of feedback loops (there is only one) and of the individual activities (R6.1). Except of updating the architectural model, the feedback loop is executed in a state-based manner, for instance, all constraints are evaluated on the architectural model when the model changes without using the changes to drive the evaluation. Hence, an event-based execution is not supported (R6.2). Finally, Rainbow enables a runtime-efficient execution for rather small adaptable systems while the performance and scalability are not comprehensively evaluated [113].

GRAF. GRAF [40, 133] provides a reusable adaptation engine that realizes a feedback loop and tackles the fine-grained adaptation of software at the level of fields and methods.

The feedback loop remains implicit in the engine's implementation (R1.1). Developers provide a reflection model, constraints, and ECA rules but they cannot design the entire feedback loop. Thus, the modularity is limited to the provided models and does not cover the structure and activities of the feedback loop (R1.2). Variability of these models is not covered (R1.3). GRAF prescribes the intra-loop coordination that cannot be adjusted by developers (R1.4). Triggers are defined by fields in the adaptable software whose changes initiate adaptation. Elaborated triggers are not addressed (R1.5). Supporting one feedback loop, GRAF does not cover the inter-loop coordination (R1.6) and distribution (R1.7).

GRAF captures the reflection model, constraints, and rules as runtime models (R2.1). It is only partially open for languages of these models since it allows a user-defined metamodel for the reflection model while prescribing the languages for the constraints and rules (R2.2). Focusing on fine-grained adaptation, abstract runtime models are not supported (R2.3).

Connecting the feedback loop to the adaptable software is supported by weaving aspects into the software for the desired fields and methods (R3.1). The aspects support parameter and structural adaptation such as changing a field's value or replacing a method (R3.2). The aspect can be woven dynamically to realize dynamic sensors and effectors (R3.3).

GRAF has not been instantiated to a layered architecture with stacked feedback loops (R4) although it provides an interface to externally manage the adaptation engine. Developers can use this interface to maintain (observe and change) the models they provided to GRAF (R5.1). More extensive changes as well as the coordination of changes due to maintenance and self-adaptation are not covered (R5.2).

Concerning the execution, GRAF does not address the concurrent execution of feedback loops (there is only one) and seemingly of the individual activities (R6.1). The feedback loop is executed in an event-based manner: a change of the adaptable software changes the reflection model and then causes the execution of the appropriate ECA rules. These rules may further operate on the whole reflection model capturing the state. Thus, an event- and state-based execution would be feasible although it is not directly addressed by the framework (R6.2). Finally, GRAF enables a runtime-efficient execution particularly since it targets local, fine-grained adaptations that do not necessarily require global knowledge and that are therefore not affected by the size of the application (R6.3).

DiVA. As described in Section 9.2.2, this framework addresses the dynamic variability in complex, adaptive systems by modeling the adaptive software and using the resulting models together with aspect-oriented modeling techniques at runtime to realize self-adaptation [306–308]. Developers using DiVA provide four models to customize the framework: a reflection model of the adaptable software, a feature model for the variability of

the software, a context model of the environment, and an ECA rule- or goal-based adaptation model. The last three models are expressed in a DSL [158, 159] that, however, does not express the feedback loop. Thus, the feedback loop is hidden in the implementation and therefore implicit (R1.1). Modularity of the feedback loop is addressed for the provided models but not for the structure and activities of the feedback loop (R1.2). In this context, means to capture the variability is not provided (R1.3). The framework implements the feedback loop as a set of interacting components that prescribe the intra-loop coordination that cannot be changed by developers (R1.4). Triggers are specified as queries on sensors events, which support complex event processing and therefore many kinds of triggers (R1.5). Focusing on a single feedback loop, the inter-loop coordination (R1.6) and distribution (R1.7) are not covered.

DiVA captures the runtime models provided by developers (R2.1). Although the authors claim that DiVA is open for different languages to express these models, they do not discuss the feasibility and the impact on the framework (*e. g.*, which components are reusable or need to be adjusted) (R2.2). Tackling architectural adaptation, abstract runtime models are supported using the metamodels given by DiVA (R2.3).

DiVA addresses the connection of the feedback loop to the adaptable software (R3.1) as it provides a mechanism to map model-level changes to the effectors. This mechanism has to be adjusted when targeting a different platform for the adaptable software. Moreover, developers have to implement listeners that update the reflection model based on sensor events. In general, DiVA relies on existing sensors and effectors that realize parameter and structural adaptation (R3.2). Dynamic sensors and effectors are not considered (R3.3).

Focusing on a single feedback loop, DiVA does not address stacked feedback loops in layered architectures (R4). It partially supports the maintenance of the self-adaptive software by allowing developers to modify the provided models but not the structure and number of the feedback loops (R5.1). For instance, developers can modify the reflection model to change the adaptable software and specify conditions of the software determining when the change can eventually be enacted to the software [309]. However, enacting such changes is not directly coordinated with the running feedback loop (R5.2).

DiVA does not support the concurrent execution of feedback loops and of individual activities (R6.1). It leverages events to update the reflection model while the analysis, planning, and execution activities work in a state-based manner. Thus, DiVA supports the event- and state-based execution (R6.2), however, it does not combine both to achieve an efficient solution. While the analysis and planning based on model weaving seem to be efficient for small models, the execute activity performs an expensive state-based comparison of two reflection models, which will likely not scale. For instance, the comparison takes 430 ms for a model with only about 20 components [305]. In general, the runtime efficiency is only shown for such small systems/models without investigating the scalability (R6.3).

PLASMA. PLASMA extends the three-layer architecture for adaptation [141] with planning capabilities [410]. The lowest layer contains the adaptable software that is adapted by the middle layer while the highest layer performs planning. Developers provide the goal (problem description) of the application as well as ADL descriptions and implementations of the components for the two lower layers (application and adaptation domain descriptions). Using these artifacts, the planning generates plans for adapting and executing the application. PLASMA is realized by a middleware supporting the three-layer architecture. We refer to Section 9.2.3 for a detailed description of PLASMA.

PLASMA does not allow developers to freely design the feedback loops that therefore remain implicit (R1.1). While the planning loop is prescribed in the middleware, the middle-

layer loop can be customized by user-defined components. Therefore, modularity is addressed for the middle-layer feedback loop but not for the planning loop. In this context, PLASMA prescribes the general structure of a feedback loop that must consist of three components: a collector for sensing, an analyzer for computing, and an admin for controlling. Moreover, the knowledge is not explicitly captured but mainly considered as events exchanged between components (R1.2). Variability is not addressed (R1.3). The intra-loop coordination cannot be adjusted by developers as it is prescribed by the middleware using event-based interactions (R1.4). Triggers are not specified in a dedicated manner (R1.5). The inter-loop coordination is addressed for the stacked feedback loops but only partially for multiple loops at the distributed middle layer by prescribing the coordination (centralized planning) (R1.6). PLASMA can be instantiated on multiple nodes while all nodes share a planning loop. Other kinds of distribution and coordination are not supported (R1.7).

PLASMA does not capture the artifacts as dedicated runtime models, for instance, the architectural model of each layer is encoded in the middleware (R2). However, the middleware supports connecting a feedback loop at one layer to the adaptable underlying layer by providing access to the underlying architecture (R3.1). The middleware supports parameter and structural adaptation (R3.2) without considering dynamic sensors and effectors (R3.3).

The stacking of feedback loops is partially addressed (R4.1). On the one hand, the extent of adapting the middle layer is not clear since the middle-layer architecture is predefined by developers and not planned by PLASMA. Moreover, the planning loop is predefined by PLASMA and not customizable by developers. Finally, the number of layer and of local feedback loops for each layer seems to be immutable. The middleware supports procedural reflection such that a feedback loop operates directly on the architecture realizing the underlying layer. Other forms of reflection are not addressed (R4.2). PLASMA partially supports off-line adaptation. Developers can adjust the goals and the application domain description after deployment. Such changes address the application at the lowest layer but not the feedback loops (R5.1). PLASMA does not coordinate the off-line adaptation, that initiates replanning, with self-adaptation but shifts this task to the user (R5.2).

The execution is concurrent at the level of feedback loops but not at level of activities (R6.1) and driven by events. The state captured by the middleware (*e. g.*, the architecture of each layer) is used by the feedback loops, however, without connecting the events to it (R6.2). The runtime efficiency of the execution is investigated for the planning but not for the overall feedback loops. For the application example, the planning is efficient (R6.3).

FUSION. FUSION is a framework for engineering self-adaptive software that tackles adaptation at the level of features and that continuously learns the impact of features on goals to improve the decision making [145, 147]. Developers using the framework specify goals as utility functions, a feature model defining the variability, and an architectural model of the adaptable software. Moreover, they have to develop transformations to synchronize the feature model, the architectural model, and the running system. The feedback loop realized by the framework work consists of two cycles, one for learning and one for adaptation. The latter uses the learned knowledge to decide which features should be (de)activated if a goal is violated. Changes of the feature selection are propagated to the architectural model and to the system with the user-defined transformations. In general, the authors state that the FUSION framework has been developed with a focus on the specific application examples and that efforts are required to apply it to different systems and goals. Thus, the framework does not provide fully generic support for engineering self-adaptive software.

The feedback loop is prescribed in the framework and therefore remains implicit (R1.1). Modularity is partially addressed since developers specify the runtime models (*e. g.*, the

feature and architectural models) but not the structure and behavior of the feedback loop (R1.2). Only by changing the framework upfront, developers can add, for instance, other learning algorithms. Means to capture the variability of the models is not provided (R1.3). Prescribing the feedback loop structure and how the feedback loop processes the models, FUSION prescribes the intra-loop coordination (R1.4). FUSION does not provide a formalism to specify triggers (R1.5). Supporting a single feedback loop, FUSION does not cover the inter-loop coordination (R1.6) and distribution of feedback loops (R1.7).

Runtime models are captured while the learned knowledge is stored in a database (R2.1). FUSION is not open for other languages to express the models (R2.2). The runtime models used in FUSION are abstract as they target the levels of architectures and features (R2.3).

Support for connecting the feedback loop to the adaptable software is not provided at the model level, which requires realizing the connection in the implementation by transformations and glue code (R3.1). While parameter and structural adaptations are supported (R3.2), dynamic sensors and effectors are not (R3.3). Stacking feedback loops in layered architectures (R4) and off-line adaptation of the self-adaptive software (R5) are not covered.

The concurrent execution of feedback loops or adaptation activities is not considered (R6.1). FUSION uses events and the state captured in runtime models to drive the execution (R6.2). For instance, individual changes drive the synchronization of the models and scope the state used for analysis and planning. FUSION achieves a runtime-efficient execution for the planning activity solving an optimization problem (*i. e.*, which features should be (de)activated), the execute activity finding a path to reconfigure the system, and for the learning. The performance of the whole feedback loop is not evaluated (R6.3).

Alférez and Pelechano [39]. This framework uses artificial intelligence (forward chaining) and runtime models to plan the adaptation of systems facing unknown contextual situations. For this purpose, developers provide several runtime models: a model of the architecture, a feature model of the variability, and a requirements model of the goals of the adaptable software as well as a context model with conditions to be analyzed, and a set of tactic models each capturing an adaptation strategy as features for the variability model. The framework realizes a feedback loop that monitors the context, evaluates the context conditions, and reconfigures the architecture using the variability model. If an unknown contextual situation occurs (*i. e.*, it is not addressed by the current variability model), the feedback loop uses forward chaining to find suitable tactics that are woven into the variability model. The adapted variability model guides the subsequent reconfiguration.

The framework defines internally the feedback loop that therefore remains implicit (R1.1). Modularity is partially addressed as developers provide the runtime models but they cannot define the adaptation activities (R1.2). Variability of the models is not considered (R1.3). The intra-loop is prescribed by the framework (R1.4). The feedback loop can only be triggered periodically (R1.5). Focusing on a single feedback loop, the framework does not cover the inter-loop coordination (R1.6) and the distribution of feedback loops (R1.7).

The provided models are captured as runtime models (R2.1) while their languages are prescribed (R2.2). However, targeting the architecture and requirements of the software, they are abstract (R2.3). The framework provides no explicit support for connecting the feedback loop to the adaptable software (R3.1). While parameter and structural adaptations are supported (R3.2), dynamic sensors and effectors are not considered (R3.3).

The stacking of feedback loops in layered architectures is not supported, however, the planning mechanism that adapts the variability model can be seen as a feedback loop on top of the mechanism (feedback loop) for adapting the architecture (R4.1). Reflection (R4.2) and off-line adaptation (R5) of feedback loops are not supported. The framework does not

support the concurrent execution of feedback loops and of activities (R6.1). The execution is driven by the state (*e. g.*, to compare models) without using change events (R6.2). The runtime efficiency of the feedback loop is not evaluated (R6.3).

Chen et al. [109]. This work proposes a framework that combines requirements-driven and architecture-based self-adaptation. For this purpose, developers provide several runtime models: a goal model, an architectural model, and a design decisions model. The latter captures the variability of the adaptable software and relates the requirements and the architectural design. The framework realizes a MAPE-K feedback loop with an advanced planner that changes the selection of the requirements if it cannot find an optimal target configuration using architectural adaptation. Eventually, it generates a transformation script that changes the architectural model to the designated target configuration.

The feedback loop is defined within the framework such that it remains implicit (R1.1). The modularity is partially covered as developers provide various models and realizations of the monitor and execute activities but they cannot influence the analysis and planning activities as well as the processing of the models (R1.2). Variability of the models is not addressed (R1.3). The framework prescribes the intra-loop coordination (R1.4) while supporting only periodical triggers (R1.5). Since the focus is on a single feedback loop, the inter-loop coordination (R1.6) and distribution (R1.7) are not addressed.

The framework captures the provided runtime models (R2.1) but it is not open for other languages for these models than the prescribed ones (R2.2). Capturing the requirements and architecture, the models are abstract (R2.3). The framework relies on other approaches to connect the feedback loop to the adaptable software (R3.1), which also support parameter and structural adaptation (R3.2). Dynamic sensors and effectors are not covered (R3.3).

The stacking of feedback loops in layered architectures is not supported but the mechanism for adapting the requirements can be considered as a higher-layer feedback loop that selects the goals for the lower-layer feedback loop to adapt the architecture (R4.1). Reflection (R4.2) and off-line adaptation (R5) of feedback loops are not supported.

The framework does not support concurrency (R6.1). The operation of the feedback loop is driven by the state of the software captured in the models (R6.2). The framework achieves a runtime-efficient and scalable execution for the planning that nevertheless may take several seconds. The performance of the whole feedback loop is not evaluated (R6.3).

J3. J3 is a tool suite for developing autonomic EJB applications [438]. It provides a modeling language to specify structurally the beans of an application as well as quality of service assertions and adaptation actions for individual or multiple beans. J3 generates code from this model for the J3 framework that provides components realizing a feedback loop. Such a feedback loop checks the assertions and triggers the actions if needed.

J3 does not make the feedback loops explicit in the architectural design as the model is at the abstraction level of the implementation and just maps assertions to actions (R1.1). The modularity is partially addressed by allowing developers to specify assertions (analysis) and actions mapped to assertions (planning). (R1.2). Assertions and actions can be hierarchically specified resulting in a corresponding hierarchy of analysis and planning steps in a loop. Their variability is not addressed (R1.3). The intra-loop coordination is realized and prescribed by the framework. Developers can only implicitly influence the coordination structure by modeling hierarchies of assertions and actions (R1.4). Triggers are prescribed by the framework that continuously monitors the application by intercepting each call to initiate the analysis. User-defined triggers are not supported (R1.5). Although multiple feedback loops that manage locally different beans exist, the inter-loop coordina-

tion is not addressed (R1.6). In this sense, the distribution of feedback loops is supported but not made explicit at the architectural and deployment levels (R1.7).

J3 does not use any runtime model as it generates code from the model and for the framework (R2). The connection of a feedback loop to a bean is defined in the model and realized by this code. Developers might have to extend the code although they do not model dedicated sensors and effectors (R3.1). J3 seems to address only parameter adaptation (R3.2). Keeping the sensors and effectors implicit, they cannot be dynamically changed (R3.3).

The stacking (R4) and off-line adaptation (R5) of feedback loops are not covered. In this context, J3 allows developers to change only the mapping of actions to assertions at runtime but not the assertions or actions that actually determine the self-adaptation capabilities.

J3 allows feedback loops to run concurrently but without any coordination and not at the level of activities within one loop (R6.1). The execution is driven by events such as the occurrence of a call or exception. The feedback loop does not maintain any state (R6.2). The performance and scalability of executing the feedback loops are not investigated (R6.3).

FESAS. FESAS is a framework for engineering adaptation engines by reusing MAPE components [264]. It provides a deployment template that structures the MAPE components, sensors, and effectors in a feedback loop, as well as a component template that realizes the communication among components using a messaging middleware and that must be completed by developers with the MAPE functionalities. The framework and the feedback loop are configured by a deployment descriptor. Tool support is provided for implementing the MAPE functionality and for specifying the deployment descriptor [263].

The feedback loop is not made explicit in the design (R1.1) as it is only described for configuring and deploying the framework. The modularity is partially addressed by supporting the individual MAPE components while neglecting the knowledge part (R1.2). Variability is not modeled and only partially captured by so called logic contracts of MAPE components (R1.3). The intra-loop coordination is realized by the framework according to the templates. It considers the communication among MAPE components but without detailing the processed knowledge (R1.4). A mechanism to specify triggers is not provided (R1.5). The inter-loop coordination is partially addressed by supporting generic message exchanges among components of different loops, however, without focusing on coordination mechanisms (R1.6). The distribution of feedback loops is partially addressed by the middleware and deployment, however, it is not systematically addressed in the design. Moreover, most application examples use a centralized feedback loop while all examples have only been simulated as opposed to applying it to a real, distributed system (R1.7).

Runtime models are not captured as the knowledge components are not detailed and their development is not supported (R2). The framework defines sensor and effector interfaces that have to be implemented by developers. Consequently, the connection of the feedback loop to the adaptable software (R3.1) as well as parameter and structural adaptation (R3.2) can be realized. Dynamic sensors and effectors are not considered (R3.3).

The stacking of feedback loops in layered architectures is not supported but only envisioned for a three-layer architecture [365] (R4). Similarly, off-line adaptation is not considered (R5). The execution model of the feedback loops is not discussed such that the level of concurrency is not clear (R6.1). The execution is driven by events (messages) sent between MAPE components and by the state kept in the components, however, without combining both to improve efficiency (R6.2). The runtime efficiency is not investigated (R6.3).

Descartes. Descartes provides a modeling languages and on-line performance prediction techniques for self-aware performance and resource management of software systems [228,

229, 254]. The language is used to describe the resource infrastructure, application architecture, deployment of the application on the infrastructure, usage profiles, and adaptation of the system. The latter is specified as strategies comprising tactics that perform atomic reconfiguration actions. The prediction techniques are used to analyze the impact of changes of the usage profiles and to plan an adaptation strategy by predicting the impact of alternative tactics. The prediction techniques enable proactive adaptations. Descartes provides a framework implementing a feedback loop that operates on runtime models expressed in Descartes and that employs the prediction techniques.

The Descartes models do not specify the feedback loop that therefore remains implicit in the framework's implementation (R1.1). Modularity is partially addressed since developers provide the Descartes models (knowledge) while the structure of the feedback loop with four MAPE activities and the use of the model by the activities are prescribed by the framework (R1.2). Descartes addresses the variability of feedback loops by supporting models that capture the adaptable software at different co-existing abstraction levels. At runtime, the appropriate abstraction level can be selected as a trade-off between accuracy and computation time of the prediction. Moreover, variability of adaptation strategies are captured by supporting multiple strategies that address the same goal. Moreover, alternative prediction techniques are implemented within the framework. However, variability of the techniques for other activities and of the feedback loop structure is not addressed (R1.3). The intra-loop coordination is prescribed by the framework that composes the components of the individual MAPE activities to a feedback loop and that defines which knowledge is used by these components (R1.4). Triggers of the feedback loop are not discussed (R1.5). Addressing a single centralized feedback loop, Descartes does not address the inter-loop coordination (R1.6) and distribution of feedback loops (R1.7).

Descartes supports capturing runtime models (R2.1). Since the whole knowledge of the feedback loop is represented in a unified Descartes model, other runtime models are not used and cannot be captured. The runtime model must be expressed with Descartes (R2.2). The Descartes model describes the adaptable software at different abstraction levels, that can be coarse-grained such that abstract runtime models are supported (R2.3).

The connection of the feedback loop to the adaptable system is realized in the framework but there does not seem to be any support by the Descartes language (R3.1). Parameter and structural adaptations are supported (R3.2) in contrast to dynamic sensors and effectors (R3.3). Descartes focuses on specific techniques for a single feedback loop (*e. g.*, sophisticated models of the adaptable system and prediction) such that it does not address layered architectures (R4) and off-line adaptation (R5).

Descartes does not concurrently execute a feedback loop (R6.1) but it supports a state- and event-based execution (R6.2). While the state is captured in the Descartes model, change events of this model are used to map model-level to system-level changes. Finally, Descartes achieves a runtime-efficient execution as it tailors the costly prediction process by selecting an appropriate prediction technique and an appropriate granularity of the model (R6.3). Nevertheless, the time to compute a prediction can vary "between seconds and a few minutes" [254, p. 60], which can be acceptable for the kind of systems targeted by Descartes. However, the performance of the whole feedback loop has not been investigated.

StarMX. StarMX is an implementation framework for adaptation engines whose adaptation logic is realized with plain Java code or condition-action rules [48]. StarMX considers a feedback loop as a chain of processes. Each process realizes (parts of) one or more adaptation activities that operate on sensors and effectors of the adaptable software. Developers configure the chain of processes in an XML file used by StarMX to execute the chain.

The feedback loop is not made explicit in the design as the chain of processes is specified late in the development with the configuration file of StarMX (R1.1). StarMX addresses modularity by allowing developers to decide which activities and shared data stores (*i. e.*, objects) are used (R1.2). Variability of these elements is not covered (R1.3). The intra-loop coordination is specified by developers who structure the processes in a chain and grant access to shared data stores when configuring StarMX (R1.4). Triggers based on time and events are supported in the StarMX configuration (R1.5). The inter-loop coordination is defined in the StarMX configuration that allows a feedback loop to trigger another loop and to share data (R1.6). The distribution of feedback loops cannot be specified (R1.7) and is only addressed by isolated instances of the framework running on different nodes.

StarMX does not use runtime models (R2). It supports connecting the feedback loop to the adaptable software based on the Java Management Extensions (JMX). Developers have to implement corresponding sensors and effectors, which are used by activities according to the StarMX configuration (R3.1). Supporting user-defined sensors and effectors, parameter and structural adaptations are feasible (R3.2). Dynamic sensors and effectors are not supported since the StarMX configuration cannot be dynamically changed (R3.3). Based on the static configuration, StarMX does not support stacked feedback loops (R4) and off-line adaptation (R5). Independent feedback loops are executed concurrently and the activities within one loop sequentially (R6.1). The execution is driven by events triggering an activity that may operate on shared data stores. Mechanisms to relate events to the data (state) are not considered (R6.2). The runtime performance of StarMX is not investigated (R6.3).

Solomon et al. [393]. This work proposes an MDA approach (*cf.* Section 2.1.2) to develop a control-theoretical self-optimization feedback loop. Using UML, the authors develop a PIM consisting of components that constitute the feedback loop and how these components interact. Refining the PIM, the authors develop a PSM targeting web services. Code is generated from the PSM and manually completed by the authors. The resulting feedback loop is applied to control the performance of a server cluster.

The UML models do not make the feedback loop explicit (R1.1) because the feedback loop elements are treated like any software element and because the adaptable software is not modeled. Hence, the feedback loop is not visible in the overall design. Modularity is addressed by decomposing the feedback loop into components for individual adaptation activities (R1.2). Variability of these components is not considered (R1.3). The intra-loop coordination is modeled with interactions among feedback loop components while neglecting the knowledge (R1.4). Specifying triggers is not supported (R1.5). Focusing on a single feedback loop, inter-loop coordination (R1.6) and distribution (R1.7) are not covered.

Runtime models are not captured (R2). The control model is represented by a black-box component and refined during implementation. Modeling sensors and effectors, the connection of the feedback loop to the adaptable software is addressed although the adaptable software is not represented in the models (R3.1). Based on the control-theoretical feedback loop, parameter but no structural adaptation is supported (R3.2). Dynamic sensors and effectors are not considered despite the support for dynamically creating components (R3.3).

The stacking (R4), off-line adaptation (R5), and concurrent execution (R6.1) of feedback loops are not covered. The execution is driven by events (messages) sent among the feedback loop components while any state knowledge is encoded in the mathematical control model and not accessible through the events (R6.2). The authors do not investigate the runtime efficiency of the execution (R6.3).

Berkane et al. [69]. This work combines feature modeling and design patterns for developing a MAPE-K feedback loop to achieve variability and reusability. Using a feature model,

a feedback loop is decomposed into four MAPE features that are further decomposed into design patterns proposed by Ramirez and Cheng [355]. These patterns are further decomposed to general design patterns for implementation, which are described with UML class diagrams. To develop a feedback loop, developers refine the feature model, select the appropriate patterns, refine the class diagrams, and implement the feedback loop accordingly.

The feature model and class diagrams represent either options or the implementation but not the architectural design such that the feedback loop remains implicit (R1.1). This approach addresses modularity by distinguishing the four MAPE activities (R1.2) as well as variability by capturing alternative patterns for each activity (R1.3). The intra-loop coordination is realized by knowledge classes shared between different patterns of each activity or by invocations (R1.4). Triggers of feedback loops are not modeled but addressed later in the implementation phase (R1.5). The approach focuses on a single feedback loop such that it does not cover the inter-loop coordination (R1.6) and distribution (R1.7).

Runtime models are not used (R2) as the knowledge is addressed only at the level of implementation classes. The connection of the feedback loop to the adaptable software is addressed by considering appropriate patterns for the monitoring and reconfiguration (*cf.* [355]) (R3.1). The sensors and effectors are completely user-defined, which enables parameter and structural adaptation (R3.2) as well as dynamic sensors (*cf.* sensor factory pattern) and likely dynamic effectors (R3.3). The stacking (R4), off-line adaptation (R5), and concurrent execution (R6.1) of feedback loops are not covered. The driver (events vs. state) (R6.2) and runtime efficiency (R6.3) of the execution are not discussed.

MechatronicUML. MechatronicUML is a refined and well-defined subset of UML for the model-driven development of safe self-optimizing mechatronic systems [187, 218]. For this purpose, UML is extended for modeling and verifying real-time and hybrid behavior. A mechatronic system is structured hierarchically into modules, each having three layers that are bottom-up: the controllers for the plant, a feedback loop switching between these controllers, and a feedback loop for the long-term planning. Using MechatronicUML, the hierarchical structure of the system is modeled with components and the behavior of components with state machines. In this context, a state machine specifies the adaptation by encoding the different modes and component configurations of a layer (*e. g.*, which controller should be used) as states and the conditions for adaptation as transitions. For instance, the behavior of the middle layer is defined by a state machine that dynamically selects the controller of the lowest layer. Based on these models, code is generated for the execution.

MechatronicUML does not provide concepts to make the feedback loops explicit in the design (R1.1). Modularity is addressed by following a component-based approach (R1.2), however, without capturing variability of feedback loop components (R1.3). The intra-loop coordination is specified by composing components via interfaces to exchange data (R1.4). Triggers are implicitly addressed by events/signals sent among components (R1.5). The inter-loop coordination is locally addressed by modeling the structure and behavior of each module and optionally by hierarchically structuring modules. Additionally, top-level modules of different nodes (agents) are coordinated in a peer-to-peer fashion (R1.6). The distribution is partially addressed by supporting multiple agents with their local modules while a single module and therefore a feedback loop cannot be distributed (R1.7).

Runtime models are not supported (R2). The connection of feedback loops to adaptable elements is addresses since all layers are specified including (sensor and effector) ports to exchange events (R3.1). Focusing on real-time systems, adaptation is mainly achieved by switching modes, which can be realized by parameter adaptation, while support for dynamic architectural changes is limited (R3.2). Dynamic sensors and effectors are not

discussed (R3.3). The stacking of feedback loops in layered architectures is partially supported since MechatronicUML prescribes a three-layer architecture for each module and a hierarchical control scheme for the whole system. More layers or a different scheme are not covered (R4.1). Support for structural reflection on feedback loops is not provided (R4.2). In this context, the adaptation is completely defined at deployment time and cannot be changed afterwards either by stacked feedback loops or off-line adaptation (R5). The concurrent execution is supported for feedback loops but not for the activities within one loop (R6.1) and driven by events (R6.2). The runtime efficiency of the feedback loops is not investigated but likely addressed targeting real-time and safety-critical systems (R6.3).

SCA-ASM. SCA-ASM is a modeling language for service-oriented applications which combines the service component architecture (SCA) for graphical, structural models and abstract state machines (ASMs) for textual, behavioral models. The language is intended for the early use in development to specify, validate, and verify applications without providing any implementation support. It has been applied to modeling feedback loops in self-adaptive software [44–46]. For this purpose, a feedback loop is modeled as a component together with the components of the adaptable software in the SCA. All of these components are represented as agents in the ASM. The behavior of a feedback loop is modeled by ASM transition rules while one rule realizes one or more MAPE activities. The ASM states correspond to system configurations.

SCA-ASM does not make the feedback loops explicit in the architectural design (R1.1). A feedback loop is modeled as a regular component/agent whose name can be the only indicator that this component/agent realizes a feedback loop. Such an approach does not make a feedback loop visible (*cf.* [210]). The individual MAPE activities are also not visible in the design since they are specified as regular rules that can only be enriched with comments to denote which MAPE activities they realize. Consequently, the SCA-ASM language does not leverage concepts of feedback loops and self-adaptive software as first-class citizens. In contrast, developers have to map such concepts to general-purpose SCA-ASM concepts.

Modularity is addressed by allowing developers to freely model the rules for the MAPE activities and shared functions for the knowledge (R1.2). The variability of rules or functions is not covered (R1.3). The intra-loop coordination is modeled by rules directly modifying shared functions (knowledge) and by invocations of rules by other rules (R1.4). The language does not provide means to specify triggers for feedback loops that are in contrast entangled into conditions of the rules for the MAPE activities (R1.5). Inter-loop coordination is supported by sharing functions (knowledge) among multiple feedback loops (R1.6). The distribution of feedback loops (R1.7) is supported with respect to decentralization of control at the architectural level, which, however, remains rather implicit due to R1.1.

Runtime models are not used and captured (R2) since the knowledge (base) is modeled as a regular component/agent or shared functions. Sensors and effectors of managed components are also modeled as functions of ASMs, which are used to connect the managing to the managed components (R3.1). Using SCA-ASM to model the managing and managed components while it does not enforce a strict separation of both kinds of components, adaptation and business concerns could easily be intertwined. Structural adaptation reconfiguring the architecture is not supported. The application examples focus on changing modes of components through parameter adaptation (R3.2). Accordingly, dynamic sensors and effectors are not supported (R3.3).

The stacking of feedback loops in layered architectures (R4) and off-line adaptation (R5) are not covered. SCA-ASM supports the early design and simulation without concretely addressing the implementation and execution of feedback loops (R6).

ActivFORMS. ActivFORMS proposes using active (*i. e.*, executable), formal models for self-adaptation [232, 437]. For this purpose, a feedback loop is modeled as a network of timed automata. Particularly, each MAPE activity is specified by one or more automata that all use C-like functions/actions to manipulate shared fields and structs capturing the knowledge. The automata of the different activities communicate by signals and the whole network is executed by a virtual machine [231]. The automata make the behavior of the individual MAPE activities explicit but not the overall feedback loop in the architectural design (R1.1). For instance, ActivFORMS does not provide means to explicitly structure the automata in a feedback loop. In contrast, the automata are implicitly structured by exchanging signals. For instance, an automaton specifies the monitoring behavior as well as the triggering of the automaton for analysis, which entangles behavior of individual MAPE functionality and the intra-loop coordination. Similarly, which knowledge is used by an automata is hidden in the actions.

ActivFORMS supports modularity by modeling separately the individual activities and the knowledge. However, the whole knowledge is captured in a single monolithic model and it is not clear whether ActivFORMS supports other feedback loop schemes than MAPE-K, for instance, concerning the number and structure of the activities (R1.2). The variability of feedback loops is addressed for the automata but not for the knowledge and captured in a goal model that describes alternative automata for different goals (R1.3). The intra-loop coordination is specified by signals sent between automata and by actions modifying the shared knowledge (R1.4). Triggers are realized by signals sent to the first automata of the feedback loop (R1.5). ActivFORMS does not address the inter-loop coordination (R1.6) and it does not support specifying the distribution of feedback loops (R1.7). Distribution is only addressed ad hoc by instantiating ActivFORMS on each node that runs locally a feedback loop without any coordination between the loops.

ActivFORMS captures each automaton as a runtime model while the knowledge is encoded in a set of fields and structs (R2.1). It prescribes the languages to express these models such that it is not open for other languages (R2.2). The captured models are abstract as they typically refer to a few parameters of the adaptable software (R2.3).

ActivFORMS prescribes interfaces for sensors and effectors that have to implemented by developers to connect the feedback loop and the adaptable software (R3.1). Based on the captured knowledge and on the application examples of ActivFORMS, the focus seems to be on parameter adaptation (R3.2). Dynamic sensors and effectors are not covered (R3.3).

Layered architectures are partially addressed by adding a goal management layer on top of the feedback loop. This layer maintains a goal model and supports exchanging the automata of the underlying feedback loop with other automata if the goals are changing. Except of the goal model, this higher layer is predefined in the virtual machine of ActivFORMS such that developers cannot design and integrate their own feedback loop here. Finally, more than two layers of feedback loops are not supported (R4.1). ActivFORMS partially supports procedural and declarative reflection (R4.2). The goal management layer keeps track of the automata that are currently executed in the underlying feedback loop and it allows directly inspecting these automata. However, an automaton cannot be directly changed but only be completely exchanged. Moreover, reflection on the structure of the feedback loop is not supported.

Off-line adaptation is partially addressed since developers can provide new goals and automata to the virtual machine but they cannot change the knowledge and the structure of the feedback loops (R5.1). Such maintenance activities are coordinated with self-adaptation by enacting the changes when the adaptation engine is in a quiescent state (R5.2).

Since the virtual machine executes a single feedback loop, there are no concurrently running feedback loops. The simultaneous execution of the automata of a feedback loop allows the concurrent execution of individual MAPE activities although this is not much exploited in the examples [232, 437]. It is not clear whether the virtual machine supports concurrent runs of the same feedback loop, that is, whether it immediately processes or buffers the signals coming from the adaptable software (R6.1). The execution of the feedback loop is driven by signals (events) and by the state of the adaptable software as captured in the knowledge base while the events direct the processing of the state. (R6.2). Finally, ActivFORMS achieves a runtime-efficient execution for a small adaptable system but the authors do not investigate the scalability [437] (R6.3).

SimSOTA. SimSOTA is a UML-based modeling and simulation approach for self-adaptive software, which describes feedback loops with activity and component diagrams [35]. Concepts that are specific to self-adaptation have been addressed by assigning corresponding names and colors to model elements. Recently, the approach has been extended with a UML profile that supports stereotypes of activities and actions in the activity diagram, for instance, to assign the MAPE steps to activities. Additionally, model templates and a code generator for the activity diagrams have been introduced. The latter supports generating code skeletons [34]. However, these extensions do not target the component diagram.

Using the SimSOTA profile, the feedback loop is made explicit in the activity diagram (behavioral view), however, not in the structural view as the profile does not customize the component diagram. Moreover, the scope of the profile is unclear as the example models do not make use of the profile (R1.1). Modularity is addressed by decomposing the feedback loop according to MAPE-K, that is, each MAPE step is modeled as an activity consisting of actions and the knowledge as a set of objects used by the actions (R1.2). Variability of the MAPE-K elements is not addressed (R1.3). The intra-loop coordination is realized by modeling the flow of actions and the flow of objects (R1.4). Triggers of feedback loops are not specified (R1.5). Multiple feedback loops can be specified and their coordination is defined similarly to the intra-loop coordination such that all feedback loops are specified within one activity diagram (R1.6). Though targeting distributed systems, the modeling and development of distributed feedback loops are not addressed (R1.7).

SimSOTA considers events/signals as knowledge but no runtime models (R2). Sensors and effectors of the adaptable software are modeled as actions in the activity diagram supporting the connection of the feedback loop to the software (R3.1). SimSOTA seems to focus on parameter adaptation without addressing structural adaptation (R3.2) and dynamic sensors and effectors (R3.3).

SimSOTA supports stacking feedback loops in layered architectures, however, it prescribes a hierarchical scheme that does not necessarily considers adaptation of feedback loops as witnessed by the examples (R4.1). Reflecting on feedback loops (R4.2) and off-line adaptation (R5) are not supported. The concurrent execution of feedback loops is supported by modeling fork and joins the activity diagram while the actions within one loop are sequentially executed (R6.1). The execution is solely driven by events (R6.2). The authors do not investigate the runtime efficiency of SimSOTA-based feedback loops (R6.3) and do not report about an evaluation of SimSOTA using the application example.

Control Loop UML. *Control Loop UML* is a UML profile to support modeling architectures of self-adaptive software [210]. It enriches UML to model controllers, process components (*i. e.*, components of the adaptable software), sensors, and effectors. Such elements can be annotated to show whether they are controllable or not, whether parameter or structural adaptations occur, and how adaptation effects are propagated in the software. Based on

such effect propagations, the feedback loop among component becomes visible. The profile supports modeling multiple feedback loops including the existence of interferences.

The profile makes the feedback loop visible in the architectural design (R1.1). It considers controllers (*i. e.*, adaptation engines) as black boxes such that the it does not address the modularity (R1.2), variability (R1.3), and intra-loop coordination (R1.4). Despite modeling sensors, the profile does not capture triggers (R1.5). The inter-loop coordination between feedback loops is partially addressed since the existence of interferences can be expressed as well as the fact when such an interference has been resolved (R1.6). However, the coordination mechanism itself is not specified with the profile. The profile captures the distribution of feedback loops in the architecture but not in the deployment (R1.7).

Focusing on architectural modeling, runtime models are not addressed at all by the profile (R2). The connection of feedback loops to the adaptable software is captured by modeling sensors and effectors (R3.1), which can support parameter and structural adaptation (R3.2). However, dynamic sensors and effectors are not covered (R3.3).

The stacking of controllers is supported while the profile is not restricted to layered architectures (R4.1). Different forms of reflecting on feedback loops are not considered in the profile (R4.2). Finally, the profile does not address the maintenance of self-adaptive software (R5) and does not provide any execution support (R6).

FAME. FAME proposes a UML profile to model self-adaptive software that employs fuzzy control in a sense-plan-act feedback loop [204]. The profile provides three views extending UML diagrams: A use case diagram to model and refine the goals and to map the goals to the activities of the feedback loop. A class diagram to model the detailed structural design of the feedback loop. A sequence diagram to model the interactions among the classes. The resulting models support the design and they can be used for code generation.

The models make the feedback loop explicit although they do not provide an architectural view (R1.1). Modularity is addressed by decomposing the feedback loop into three classes (sense, plan, and act) that can be further refined to capture sensors and effectors while the knowledge is only represented as data stores (R1.2). FAME does not capture the variability of such elements (R1.3). The intra-loop coordination is modeled by structuring the classes in a cycle and by associating classes to data stores (R1.4). Triggers can be specified based on time and events (R1.5). FAME supports modeling single feedback loops such that the inter-loop coordination (R1.6) and distribution (R1.7) are not covered.

FAME does not use runtime models to detail the knowledge but considers only knowledge stores (R2). Modeling sensors, effectors, and the adaptable software in the class diagram, FAME specifies how the feedback loop is connected to the software (R3.1). Effectors can be labeled to show whether they realize parameter or structural adaptation although only the (de)activation of components seems to be eventually supported (R3.2). Dynamic sensors and effectors are not covered (R3.3). Support for stacked feedback loops (R4), offline adaptation (R5), and execution (R6) is not provided.

ACML. ACML is the adapt case modeling language that is based on UML and tackles the early design of self-adaptive software [280]. It extends former work of the authors [281]. A feedback loop is modeled as a use case and refined with two UML activities that together specify the behavior of the loop as an adaptation rule. The first activity covers the monitoring and analysis steps while the second captures the planning and execution steps. The structure of the self-adaptive software is specified in a component diagram that captures the adaptable software, sensors, effectors, and the parametric knowledge about (past) sensor values. The adaptation rule uses these components to perform self-adaptation, for instance, by requesting data from sensors and invoking effectors.

The feedback loop is made explicit in the ACML models although it is not represented in the structural diagram. Developers have to relate the use case and activity diagrams to the component diagram to make the feedback loop visible in the architecture (R1.1). Modularity is partially addressed as the feedback loop is decomposed to only two coupled activities, monitoring/analysis and planning/execute, and parametric knowledge (R1.2). Variability of feedback loop elements is not considered (R1.3). The intra-loop coordination is modeled by invocations (the monitoring/analysis activity invokes the planning/execute activity) and shared knowledge (R1.4). However, the use of the knowledge is not directly visible since the activities are modeled separately from the knowledge in different diagrams. ACML supports specifying time- and event-based triggers (R1.5). Despite supporting multiple feedback loops, the inter-loop coordination is not addressed (R1.6). The distribution of feedback loops is only visible in the architecture when considering and relating all diagrams while neglecting the deployment (R1.7).

Runtime models are not used (R2). Modeling the sensors and effectors, the connection of the feedback loop to the adaptable software is supported (R3.1). Parameter and structural adaptation can be performed on the knowledge (R3.2) while dynamic sensors and effectors are not considered (R3.3). ACML does address stacked feedback loops (R4) and off-line adaptation (R5). Focusing on the early design, it does not cover the execution (R6).

ACTRESS. ACTRESS is a domain-specific modeling approach for specifying and executing feedback loops that use control-theoretical concepts and whose individual activities are actors according to the actor programming paradigm [260–262]. The target domain is self-optimization to manage the resource efficiency and throughput of applications. While an informal graphical notation is introduced, developers actually use a textual syntax for the specification. They extend the specification with Xbase/Java-based implementations of the actors. From the specification, executable code is generated for the Akka actor framework.

An ACTRESS model makes the feedback loop explicit in the design (R1.1). However, the feedback loop with its cyclic structure in terms of connections among actors cannot be easily identified in the textual model, especially if the actors are hierarchically decomposed. ACTRESS supports the modularity of feedback loops since developers can freely decide on the actors and the composition of these actors (R1.2). ACTRESS does not capture the variability of a feedback loop such as alternative actors (R1.3). The intra-loop coordination is supported as developers connect the actors to form a feedback loop. A connection wires two ports of different actors, which also define the type of data that is exchanged between these actors (R1.4). Triggers of a feedback loop are supported by specifying and implementing specific actors that execute based on external events or periodically (R1.5). The inter-loop coordination is theoretically addressed by allowing one feedback loop to trigger another feedback loop, which, however, has not been evaluated and is considered as future work [262] (R1.6). The distribution of feedback loops is partially supported since the actors can be distributed. However, the distribution is not specified in the ACTRESS model (R1.7).

In ACTRESS, the knowledge of a feedback loop is only implicitly captured by typed ports of actors, which support exchanging data among actors. Such data are Java-typed parameters of the adaptable software. Thus, explicit runtime models are not used (R2).

ACTRESS addresses the connection of the feedback loop to the adaptable software by treating sensors and effectors as actors that have to be specified and implemented by developers (R3.1). Capturing parameters and using control-theoretic concepts, ACTRESS focuses on parameter adaptation (R3.2). Dynamic sensors and effectors are not discussed but they might be possible as the actor paradigm supports dynamically creating actors (R3.3).

ACTRESS supports the stacking of feedback loops in layered architectures by allowing actors to adapt the parameters of other actors (R4.1). Structural adaptations of feedback loops are discussed theoretically. Focusing on parameter adaptation, the higher-layer feedback loops do not structurally reflect on the lower-layer feedback loops but only observe individual parameters (R4.2). ACTRESS does not address off-line adaptation (R5).

The concurrent execution of feedback loops is partially addressed (R6.1). While concurrent feedback loops operating at the same layer are not discussed, the individual actors of a feedback loop may run concurrently. The execution of a feedback loop is driven by events (messages) sent among the actors without considering any state (runtime models) (R6.2). The runtime efficiency of ACTRESS has not been evaluated. The authors refer only to the runtime efficiency of the underlying Akka framework (R6.3).

10.2.2 *Discussion*

In the previous section, we have presented the alternative approaches to EUREMA and how well they cover the requirements (*cf.* Table 12 on Page 235). To complete the presentation, we refer to Section 9.6 where we assessed in detail how well EUREMA fulfills the requirements. In summary, EUREMA addresses all requirements except of partially supporting the variability (R1.3) and concurrent execution (R6.1) of feedback loops and neglecting the distribution of feedback loops (R1.7). Accordingly, EUREMA is listed in Table 12.

Based on this presentation, we summarize and discuss the state of the art in engineering self-adaptive software in the following. For this purpose, we use the requirements R1–R6.

R1: Feedback Loops. In line with the observation made in a 2009 survey on self-adaptive software by Salehie and Tahvildari [370], we observe that development approaches to self-adaptive software use some form of models. However, such models often focus on the knowledge part of a feedback loop (*i. e.*, they describe the adaptable software and the feedback loop's working data such as adaptation rules) and neglect the individual MAPE functionalities and their composition to a feedback loop. Consequently, the models typically do not specify the feedback loop that therefore remains implicit in the architectural design of self-adaptive software (R1.1).

This is typical for frameworks that often hide the feedback loop in their implementations. Examples are MADAM, MUSIC, ASSL, Genie, Rainbow, GRAF, DiVA, PLASMA, FUSION, Alférez and Pelechano, Chen et al., J3, FESAS, and Descartes. Such frameworks prescribe a certain feedback loop in terms of the number and structure of adaptation activities, how these activities coordinate, and how they process and share knowledge. Developers can customize such feedback loops only by providing models of the knowledge part. This limits the free design of the entire feedback loop. Consequently, such frameworks provide less modular feedback loops (R1.2) and do not allow developers to customize the intra-loop coordination (R1.4). The usual goal of such frameworks is to ease the development by enabling reuse at the costs of prescribing the design of the feedback loop with its activities and intra-loop coordination. Consequently, the "creative freedom may be lost because many design decisions have already been made by framework developers" [355, p. 50].

However, there are approaches typically accompanied with modeling languages[2] that allow developers to design and implement feedback loops more freely. This is enabled by fully supporting modular feedback loops (R1.2) and intra-loop coordination (R1.4), that is, developers can design the individual elements of a feedback loop and the coordination

2 The only exception is StarMX that is an implementation framework without a modeling language.

of these elements to form a feedback loop. Such approaches can be further distinguished whether they make the feedback loop explicit in the architectural design or not (R1.1). Those that do not are StarMX, Solomon et al., Berkane et al., MechatronicUML, SCA-ASM, ActivFORMS, and SimSOTA. However, all of them except of SCA-ASM achieve implemented feedback loops that can be executed (R6). In contrast, approaches that make the feedback loop explicit such as Control Loop UML[3], FAME, and ACML focus on the design and do not address the implementation and execution of feedback loops (R6). An exception is ACTRESS that provides code generation capabilities for the corresponding models to obtain executable code. In this context, EUREMA allows developers to freely design the individual elements of a feedback loop (R1.2) and the composition of these elements to form a feedback loop (R1.4) while making the feedback loop explicit in the design (R1.1).

The variability of feedback loop elements (R1.3) is neglected by almost all of the approaches. An exception is the work by Berkane et al. [69]. Descartes and ActivFORMS partially address variability by maintaining alternative models that are selected at runtime. FESAS and EUREMA describe contracts or signatures of elements, however, without capturing the variability of these elements.

The triggers of feedback loops (R1.5) are often addressed by providing a mechanism for their specification or implementation (*e. g.*, events in Tune, Genie, DiVA, StarMX, and ActivFORMS, sensors in Rainbow, or actors in ACTRESS). However, there are approaches that do not explicitly support the triggering that therefore has to be realized somehow in the implementation (*e. g.*, as in PLASMA, FUSION, FESAS, SCA-ASM, SimSOTA, Control Loop UML). EUREMA provides a language to specify triggers based on time or events.

Approaches that do not cover the inter-loop coordination (R1.6) target single or independent feedback loops that do not require any coordination. Other approaches that partially cover this aspect either prescribe a coordination scheme (*e. g.*, centralized planning in MADAM, MUSIC, and PLASMA), provide technical support (*e. g.*, messaging and triggering among feedback loops as in FESAS and StarMX), or just denote the existence of coordination (Control Loop UML). In contrast, approaches such as MechatronicUML, SCA-ASM, SimSOTA, and EUREMA allow developers to specify the inter-loop coordination.

The distribution of feedback loops (R1.7) is not comprehensively addressed by all approaches. Typically, they prescribe a certain kind of distribution or they address the distribution at the level of models (*e. g.*, to specify decentralization) but not in the subsequent development. In EUREMA, we completely excluded the distribution of feedback loops.

R2: Runtime Models. Considering R2.1 in Table 12 on Page 235, we see that runtime models are captured and used by frameworks (*e. g.*, Rainbow, GRAF, DiVA, FUSION, and Descartes) while those approaches with languages that make the feedback loop explicit in design (R1.1) do not capture and use runtime models. The only exception is EUREMA.

Similarly, EUREMA is the only approach that is completely open for user-defined languages to express runtime models (R2.2). Most approaches prescribe the languages while GRAF allows user-defined languages only for one specific model, the reflection model, and DiVA claims openness but provides no argument or evidence for it. As most approaches tackle self-adaptation at the architectural level, the used runtime models are usually abstract (R2.3) as they capture and refer to the architecture of the adaptable software.

R3: Sensors and Effectors. All approaches provide some support for connecting a feedback loop to the adaptable software (R3.1). The connection is either defined in the models or

3 With respect to R1.1 and R1.2, Control Loop UML is an exception as it makes the feedback loop explicit in the architectural design although it considers non-modular (black-box) feedback loops abstracting from the intra-loop coordination.

directly realized in the framework or middleware. As most approaches tackle architectural self-adaptation, they support parameter and structural adaptation (R3.2). Dynamic sensors and effectors (R3.3) are addressed by several approaches including EUREMA. However, they are realized solely at the implementation level in these approaches without providing any support for their specification in the design.

R4: Layered Architectures. The stacking of feedback loops in layered architectures (R4) is covered by several approaches. However, most of them prescribe the number of layers and sometimes even the feedback loops at certain layers (*e. g.*, PLASMA, Alférez and Pelechano, Chen et al., MechatronicUML, and ActivFORMS) or they prescribe a hierarchical control scheme that need not necessarily perform self-adaptation (*e. g.*, Alférez and Pelechano, Chen et al., SimSOTA, and MechatronicUML). The only approaches that do not restrict the number of layers are Control Loop UML that, however, focuses only on the design as well as ACTRESS and EUREMA that also realize the stacked feedback loops.

Support for procedural and declarative reflection on feedback loop (R4.2) that covers all aspects of a feedback loop is only achieved by EUREMA. In contrast, other approaches only support a specific kind and scope of reflection. For instance, PLASMA support procedural reflection on the structure of a feedback loop, MechatronicUML and ACTRESS focus on reflecting on parameters, or ActivFORMS reflects upon the models for individual MAPE activities without taking the whole feedback loop into account.

R5: Off-line Adaptation. We identified several approaches that allow developers to change the self-adaptive software after deployment for maintenance (R5.1). These approaches are typically frameworks that are customized by developers who provide model for the knowledge part. These models can be changed by developers at runtime and updated in the self-adaptive software. Examples of these approaches are MADAM, MUSIC, Genie, Rainbow, GRAF, DiVA, PLASMA, and ActivFORMS. As these approaches prescribe the number of feedback loops and layers as well as the structure of feedback loops in terms of adaptation activities and runtime models, the evolution of the software by off-line adaptation is constrained by such prescriptions. Thus, the software cannot escape from the prescribed framework. In contrast, EUREMA supports changing individual runtime models used in a feedback loop as well as more drastic changes such as adding or removing a feedback loop at any layer and patching the adaptable software. Therefore, EUREMA provides a more comprehensive support for maintenance than the other approaches.

The coordination of enacting an off-line adaptation (*i. e.*, a maintenance change) to the self-adaptive software is typically not coordinated with self-adaptation (R5.2). For instance, PLASMA shifts the responsibility of the coordination to the developer. In contrast, ActivFORMS and EUREMA support quiescence for the adaptation engine such that an off-line adaptation can be safely enacted on the feedback loops.

R6: Execution. In general, some approaches do not provide any execution support as they focus on the design (*e. g.*, SCA-ASM, Control Loop UML, FAME, and ACML).

Concerning R6.1, the execution of the adaptation activities within a feedback loop is usually sequential. Exceptions are ActivFORMS and ACTRESS that theoretically support the concurrent execution of automata respectively actors. Thus, the concurrent execution typically happens at the level of multiple feedback loops. As some approaches target single feedback loops, they do not address any concurrency (*e. g.*, Tune, Genie, Rainbow, GRAF, DiVA, FUSION, Alférez and Pelechano, Chen et al., Descartes, Solomon et al., and Berkane et al.). In contrast, SimSOTA, J3, StarMX, ASSL, and EUREMA allow the concurrent execution of independent feedback loops while PLASMA and MechatronicUML consider the

concurrent execution of stacked feedback loops. Finally, MADAM and MUSIC address the concurrency when a feedback loop is distributed on multiple nodes. Finally, we identified no approach that covers the concurrent execution at both levels, that is, at the level of adaptation activities and at the level of feedback loops.

The execution of a feedback loop is often either based on events (*e. g.*, notifying about changes of the adaptable software or context) or on the state (*e. g.*, a reflection model of the adaptable software) (R6.2). Only a few approaches such as FUSION, Descartes, ActivFORMS, and EUREMA combine both to improve runtime efficiency. Thereby, EUREMA allows developers to decide which activities of a feedback loop exploit change events to improve the processing of reflection models.

Finally, the runtime efficiency of executing feedback loops (R6.3) is often not investigated and therefore remains unknown. Approaches that achieve a partially efficient execution either investigate only the performance of an individual adaptation activity but not of the whole feedback loop (*e. g.*, PLASMA, FUSION, Chen et al., and Descartes) or they achieve efficiency for a subset of the activities (*e. g.*, DiVA). Finally, the remaining approaches achieve an efficient execution although they often do not investigate the scalability. In contrast, we have shown the runtime efficiency of EUREMA and EUREMA-based feedback loops by discussing the performance and scalability (*cf.* Sections 9.4.2, 9.5.5, and 9.5.6).

Summary. Summing up, EUREMA covers most of the requirements in contrast to the state of the art. Its unique features are the combination of allowing developers to freely design the entire feedback loop when developing a specific self-adaptive software (R1.2 and R1.4), making the feedback loop explicit in the architectural design (R1.1), using runtime models while being open for languages of runtime models (R2.1 and R2.2), supporting layered architectures and off-line adaptation (R4 and R5), and achieving a runtime-efficient execution (R6.3) by exploiting event- and state-based techniques (R6.2).

To achieve these features, we do not address in detail the variability of feedback loops and their elements (R1.3). Moreover, we neglect the distribution of feedback loops (R1.7) and the concurrent execution of coordinated (dependent) feedback loops and of individual adaptation activities within a feedback loop (R6.1). Particularly, the distribution and concurrency are major concerns that would cross-cut the EUREMA language and make it more difficult to address the other requirements.

A notably aspect of EUREMA is that we use the same models for designing *and* for executing feedback loops. For the execution, we developed an interpreter. No other approach whose models make the feedback loops explicit in the design use these models at runtime for execution. Close to this idea is ACTRESS that generates executable code from the models while the code can be traced back to the models as well as ActivFORMS that uses an interpreter to execute automata that, however, capture the behavior of individual adaptation activities but not the overall feedback loop. In this context, the interpretation of automata in ActivFORMS is similar to the interpretation of the executable Story Diagrams (SDs) specifying the analysis and planning behavior in the EUREMA-based feedback loops we developed for the application example (*cf.* Section 9.3.2).

Overall, EUREMA goes beyond the state of the art concerning frameworks because it does not prescribe any structure of the adaptation activities or feedback loops and it does not limit the number of feedback loops and layers. In contrast to existing approaches using modeling languages, EUREMA provides improvements by keeping the models alive at runtime for executing feedback loops, which eases changing the feedback loops and layers either dynamically by self-adaptation or by off-line adaptation.

11 CONCLUSION AND FUTURE WORK

11.1 CONCLUSION

In this thesis, we have tackled the research problem of engineering self-adaptive software with feedback loops and Models@run.time. For the engineering, we identified requirements that address the specifics of feedback loops and runtime models. A major requirement is to make the feedback loop explicit in the architectural design. This enables engineers to freely design and reason about the control/adaptation strategy realized by the feedback loop. To comprehensively capture the feedback loop, the engineering should address the feedback loop's individual elements such as adaptation activities (modularity), the variability of these elements, the interactions of these elements (intra-loop coordination), the triggers for the execution, and the interactions with other feedback loops (inter-loop coordination). A further requirement is to address the runtime models with their use in the feedback loop and their causal connection to the adaptable software via sensors and effectors. An explicit runtime model of the software allows engineers to decide on how to reflect on the software according to the selected adaptation strategy. The engineering may further consider the stacking of feedback loops to increase the flexibility, the maintenance to address the long-term evolution, and the runtime environment to enable the execution of self-adaptive software.

Therefore, we have proposed EUREMA, a Model-Driven Engineering (MDE) approach for self-adaptive software that addresses these requirements. EUREMA provides a domain-specific modeling language to specify feedback loops that therefore become explicit in the design. The language leverages well-known concepts of the self-adaptive software domain such as the external approach, MAPE-K feedback loops, layered architectures, and runtime models. Consequently, EUREMA enables developers who are familiar with the domain to directly use these well-known concepts to specify self-adaptive software. Developers can freely design a feedback loop as they specify the individual elements comprising adaptation activities and runtime models, the coordination of these elements, and the trigger of the feedback loop. Thereby, the variability of feedback loop elements and the connection of the feedback loop to the adaptable software via sensors and effectors are taken into account. Moreover, EUREMA allows developers to freely decide about the number of feedback loops and how these loops are coordinated. In this context, feedback loops can also be stacked in layers to achieve adaptive control architectures to improve the flexibility of self-adaptation. The resulting EUREMA models are kept alive and they are directly executed by the EUREMA interpreter to run the feedback loops. The EUREMA language and interpreter jointly support the maintenance of the self-adaptive software in terms of off-line adaptation. Corresponding to a free design of a feedback loop created by a developer, off-line adaptation may target all aspects of this design.

In contrast to other proposals for engineering self-adaptive software, EUREMA is a seamless approach that covers the specification and the execution of feedback loops. Particularly, EUREMA models allow developers to describe designs of feedback loops at a higher level of abstraction referring to well-known concepts of the domain while they are precise enough to be executable. Thus, EUREMA contrasts other proposals by keeping the models

specifying feedback loops alive at runtime to execute, adapt, and evolve feedback loops. Furthermore, addressing feedback loops that heavily use runtime models, EUREMA promotes the rigorous use of models and MDE techniques at runtime. In this context, other proposals for explicitly modeling feedback loops in self-adaptive software either do not provide any runtime support at all or they do not keep the corresponding models alive at runtime. On the other hand, frameworks for self-adaptive software provide runtime support for executing feedback loops either in an event-based or state-based manner. However, they do not allow developers to freely design feedback loops (*e. g.*, with respect to the structure and number of feedback loops as well as the number of layers) that are on the contrary prescribed and implicit in the framework implementation. Thus, most of the design decisions are already made by the framework. Accordingly, any support for adapting and evolving such feedback loops is restricted to the few design decisions that a framework leaves open for the developer. This combination of supporting a free design of feedback loops (including the adaptation activities and runtime models), making the feedback loops explicit in the design, and keeping the feedback loop specifications (*i. e.*, the EUREMA models) alive at runtime for execution, adaptation, and evolution makes EUREMA unique.

Therefore, EUREMA makes the following contributions to the state of the art in engineering self-adaptive software.(1) Being an *integrated* MDE approach, EUREMA rigorously and consistently uses models for engineering self-adaptive software and throughout the software's life cycle, that is, for the specification, implementation (*e. g.*, the model-driven adaptation activities operating on reflection models), execution, adaptation, and evolution. Thereby, EUREMA does not impose any restrictions on the design of feedback loops. (2) EUREMA is an *open* approach as it allows developers to integrate their own techniques for realizing individual adaptation activities and their own languages to express the runtime models used within the feedback loops. (3) EUREMA seamlessly integrates the development and runtime environments since it allows the use of the same models in both environments and even the exchange of the models between the environments. This avoids any translation step of the models from the development to the runtime environment. (4) EUREMA enables the adaptation and evolution of feedback loops by allowing developers to stack feedback loops in an arbitrary number of layers and to maintain the running feedback loops after deployment. The related changes of feedback loops may target the whole design of a feedback loop created by developers. (5) EUREMA combines a state- and event-based execution of feedback loops by exploiting runtime models capturing the state of the adaptable software and events notifying about changes of the state. Relating the events to the state, EUREMA supports an incremental processing of the individual adaptation activities to improve the runtime performance of the feedback loops.

To provide evidence for the contributions and benefits of EUREMA, we comprehensively evaluated EUREMA by assessing the design and expressiveness of the language, instantiating EUREMA to several self-adaptation problems, conducting experiments with the running mRUBiS system, comparing EUREMA-based solutions with alternative solutions for feedback loops, and discussing how well EUREMA covers quality attributes and fulfills the requirements for engineering self-adaptive software. All the evidence collected from the evaluation indicates that we have achieved with EUREMA a high-quality, expressive, effective, and competitive (with respect to development costs and runtime performance) approach for developing and executing feedback loops in self-adaptive software.

11.2 FUTURE WORK

Finally, we outline future work to address limitations of EUREMA. The aspects we discuss in the following improve or extend the evaluation, the language, the development support, the execution support, assurances, and the coverage of variants of self-adaptive software.

Evaluation. Among others, we evaluated EUREMA by instantiating it to the concrete problem of engineering feedback loops for the self-healing and self-optimization of mRUBiS (*cf.* Chapter 5). This instantiation demonstrated the feasibility of EUREMA and of the developed solutions to solve this problem (*cf.* Sections 9.3 and 9.4). To obtain more evidence for the general feasibility of engineering self-adaptive software with EUREMA, future work should instantiate EUREMA to other problems such as self-protection [445] and other adaptable software such as Znn.com[1]. Moreover, the developed solutions should adopt other approaches to adaptation since we focused in this thesis on architecture-based adaptation. Other approaches that should be investigated are adaptations based on control [4, 5], requirements [430], and the combination of architecture and requirements [43].

In this context, empirical studies should investigate the quality attributes (*e. g.*, extendability, usability, and reusability) that we discussed qualitatively for EUREMA (*cf.* Section 9.5). Such studies should aim for quantitatively validating the benefits of EUREMA as a DSL although this is "in general as well as in particular cases [...] hard and an important open problem" [302, p. 317] for DSLs. Finally, future work on the evaluation of EUREMA should address off-line adaptation that we have not investigated in depth. On the one hand, the evolution of self-adaptive software is not the primary focus of EUREMA. On the other hand, the evolution of software in general is a research field on its own [66, 353] and the seamless integration of evolution and self-adaptation is an open research problem [1].

EUREMA Language. In the short term, the EUREMA language can be improved in several dimensions. First, we required that a feedback loop has exactly one trigger (*cf.* Section 5.1.3 and 6.4.2), which requires to combine all desired triggering conditions to one trigger specification. For instance, a trigger can have at most one period influencing the frequency of a feedback loop such that it is not possible to relate different events to different periods. Thus, the use of multiple triggers for a feedback loop should be investigated. Moreover, feedback loops that are stacked on top of each other are currently triggered bottom-up (*cf.* Section 5.3 and 6.4.2). A top-down triggering as used in the reference architecture by Kramer and Magee [258] when the system goals are changing should be investigated.

Another dimension is the stereotyping and labeling of elements in EUREMA models. We stereotype adaptation activities with the MAPE steps they are realizing and runtime models with their purpose. Moreover, we label the usage of runtime models by activities to indicate how the models are used (*cf.* Section 5.1.2). Conceptually, such stereotypes and labels are not part of the EUREMA language but they belong to a profile extending the language. Technically, we encode the stereotypes and labels in attributes of EUREMA model elements without using a profile mechanism. We will investigate the use of such a mechanism [268] to systematically define and integrate profiles (*e. g.*, for different feedback loop schemes such a MAPE-K, observe-decide-act, or learn-reason-act) in EUREMA.

Furthermore, the EUREMA language does not comprehensively capture the sensors and effectors of the adaptable software, which would be required if we want to make the monitoring more adaptive. In this case, the characteristics of the sensors such as their accuracy and sampling rate are needed to adapt the monitoring infrastructure.

1 Znn.com: http://www.self-adaptive.org/exemplars/model-problem-znn-com/.

Finally, a long-term improvement of the EUREMA language would be an interface mechanism for adaptation activities and runtime models that ease the reuse, modularity, analysis, and composition of these elements to form a feedback loop. The same applies to FLDs realizing (fragments of) feedback loops and whose instances (*i. e.*, megamodel modules) are composed in a layered architecture. In this context, we currently consider the syntactic signature of an FLD respectively megamodel module such that a complex model operation can invoke the module (*cf.* Section 5.1.4). However, an interface should additionally cover semantics, for instance, expressed by invariants, pre- and post-conditions. With such interfaces in place, we can thoroughly address the variability of feedback loops.

Development Support. Considering the development support of EUREMA, we did not address the requirements before designing the feedback loops with EUREMA. Thus, future work should investigate the integration of EUREMA with existing approaches to requirements engineering for self-adaptive software [352, 388, 440]. For instance, how can we systematically transform RELAX models specifying the requirements for self-adaptive systems [440] to initial EUREMA models when moving from the requirements to the design?

In this context, templates (*cf.* [233]) for EUREMA models can be helpful, which provide an initial design based on existing reference models and architectures (*e. g.*, MAPE-K, the external approach, or the three-layer architecture by Kramer and Magee [258]). Such templates can be accompanied with design patterns for self-adaptive software to ease the design and development of the feedback loop, particularly, of the adaptation activities that have to be realized by developers in EUREMA. Therefore, we will investigate the design patterns proposed by Ramirez and Cheng [355] and their integration into EUREMA.

Finally, the development can be further supported by a library of reusable model operations (adaptation activities) and runtime models—the building blocks of a feedback loop. Reuse is conceivable at two levels. First, at the modeling level by providing fragments of EUREMA models capturing the operations and runtime models. Second, at the implementation level by providing tools and engines realizing operations and materialized runtime models. Consequently, future work should research such a library, which requires beforehand well-defined interfaces for the reusable elements (*cf.* previous discussion).

The ultimate goal is to create a design space for self-adaptive software [92] that takes specifics of self-adaptive software such as uncertainty [148, 358] and coordination of feedback loops (*e. g.*, different coordination schemes [129, 200, 245]) into account.

Execution Support. The execution support of EUREMA can be improved with respect to concurrency. Currently, EUREMA supports the concurrent execution of independent feedback loops but not of individual adaptation activities within a single or multiple coordinated feedback loops. Therefore, the need and benefits of concurrency at the level of activities should be investigated. Particularly, concurrency of activities likely requires synchronization constructs in the EUREMA language, which complicates the language. Thus, a trade-off between efficiency of execution and ease of use must be found.

Moreover, the execution of EUREMA models does not consider any error handling at runtime when the execution of a feedback loops fails. So far, EUREMA shifts this responsibility to developers who implement the individual adaptation activities and therefore can introduce corresponding mechanisms such as a robust protocol to enact an adaptation [82]. However, we will investigate whether any error handling mechanism should be provided at the level of EUREMA models—at least to gracefully degrade the execution and notify the user or developer when a failure occurs. Similarly, we may incorporate the user or developer at runtime to supervise, visualize, and control the self-adaptive software as envisioned by Kephart [244]. For this purpose, EUREMA has the advantage that is keeps

the EUREMA models created by the developer alive at runtime for execution. Thus, developers can directly inspect the models used in the adaptation engine without having to translate any concepts otherwise used for the implementation and execution back to the design models. Besides inspection, future work may address how a user can interact with the adaptation expressed and executed with EUREMA.

Assurances. Throughout the thesis, we rather neglected assurances for the self-adaptive software developed with EUREMA. In this regard, we analyze the conformance of EUREMA models to the metamodel including the well-formedness constraints (*cf.* Section A.1). Additionally, the executability of EUREMA and the use of runtime models within feedback loops enable the simulation (*cf.* Section 8.4) and testing (*cf.* [14]) of feedback loops to validate the self-adaptation at design time.

However, to comprehensively provide assurances for EUREMA-based feedback loops, we have to establish a formal foundation for EUREMA. One potential formalism is graph transformations [57, 8] that we used to specify the execution semantics of EUREMA (*cf.* Appendix B). Therefore, we want to investigate how this formalism can be exploited to analyze and verify EUREMA models specifying feedback loops. A static verification of a feedback loop as a composition of model operations and runtime models further requires well-defined interfaces for these elements (*cf.* previous discussion). Besides such design-time assurances, runtime analysis and verification are required since the feedback loops may dynamically change. A basic runtime analysis may check whether operations actually use the runtime models in the way as it is specified by the labels for the model usage relationships in the EUREMA model. For instance, if the EUREMA model specifies that an operation reads a specific runtime model, then this operation must not change the model at runtime. Thus, analyzing the conformance of the execution to the specification is conceivable. In this context, the effective combination of design-time an runtime verification and therefore the composition of assurances become relevant (*cf.* [17]).

To guide the work on assurances for EUREMA, we will investigate different types of properties proposed by Iglesia and Weyns [233]. They consider three types that address either whether an individual adaptation activity works properly, whether two or more adaptation activities interact properly, and whether the whole feedback loop satisfies its goals. Based on such property types and a formal foundation for EUREMA (*e. g.*, graph transformations), we will investigate assurances techniques for EUREMA.

Variants of Self-Adaptive Software. So far, EUREMA targets self-adaptive software that employs a single and centralized adaptation engine. Consequently, we did not address the distribution of feedback loops. Thus, future work should tackle distributed self-adaptive software systems with decentralized control [436]. This requires identifying how EUREMA must be extended to cover the distribution and decentralization. For instance, the distribution could be specified with a deployment model. A challenge here is to maintain the ease of use of EUREMA while adding more concerns to the language.

One promising direction is to use open and adaptive collaborations to specify the interaction/coordination among decentralized feedback loops including the exchange of runtime models. We already started this direction with an idealized formalization in the context of systems of systems [13] but we have not realized this idea yet in practice with a prototype. For this purpose, mechanisms to exchange runtime models become relevant (*e. g.*, [195]).

As another variant of self-adaptive software, the idea of self-aware computing systems has recently emerged, which are systems that are self-reflective, self-predictive, and self-adaptive [255]. Developing such systems with different levels of self-awareness such as time, goal, stimulus, and interaction awareness is challenging [277]. We started to discuss

architectures and designs for such systems at the conceptual level while using a notation inspired by EUREMA [10–12]. Extending and using EUREMA to address the engineering of such systems is an interesting area for future work.

Finally, EUREMA targets the engineering of feedback loops for the adaptation engine while rigorously using models throughout the software's life cycle. The rigor use of models for the design, execution, and adaptation of feedback loops enables flexible solutions. As future work, the same rigor of using models can also be applied to the adaptable software to improve flexibility. For instance, Derakhshanmanesh et al. [134] propose model-integrating components in which code and executable models co-exist. Using such components, the models embedded in the components provide the flexibility for runtime adaptation. If we consider such components that contain only models (*cf.* model-only in [134]), the whole adaptable software is specified, executed, and adapted with models. In this case, these models and the reflection models of the adaptable software could collapse, which avoids the need for maintaining the causal connection (*cf.* procedural reflection in Section 2.1.5 and [103]). Therefore, the same benefits of using executable EUREMA models at runtime could be leveraged for the adaptable software.

Summary. Finally, we want to point out that addressing all these aspects for future work is challenging as they together aim for EUREMA models that are compositional, executable, and intuitive while allowing their formal analysis and verification. Models with such characteristics are quite visionary (*cf.* [205]) such that many interesting research problems will appear when tackling the future work discussed here.

BIBLIOGRAPHY

OWN PUBLICATIONS

[1] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério de Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. "Software Engineering Processes for Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw. Vol. 7475. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 51–75. DOI: 10.1007/978-3-642-35813-5_3.

[2] Amel Bennaceur, Robert B. France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Mosterman, Walter Cazzola, Fábio M. Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Aksit, Pär Emmanuelson, Huang Gang, Nikolaos Georgantas, and David Redlich. "Mechanisms for Leveraging Models at Runtime in Self-adaptive Software." In: *Models@run.time*. Ed. by Nelly Bencomo, Robert B. France, Betty H. Cheng, and Uwe Assmann. Vol. 8378. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 19–46. DOI: 10.1007/978-3-319-08915-7_2.

[3] Jens Bruhn, Christian Niklaus, Thomas Vogel, and Guido Wirtz. "Comprehensive support for management of Enterprise Applications." In: *International Conference on Computer Systems and Applications*. AICCSA '08. IEEE, 2008, pp. 755–762. DOI: 10.1109/AICCSA.2008.4493612.

[4] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. "Control Strategies for Self-Adaptive Software Systems." In: *ACM Trans. Auton. Adapt. Syst.* 11.4 (2017), 24:1–24:31. DOI: 10.1145/3024188.

[5] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. "Software Engineering meets Control Theory." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. IEEE, 2015, pp. 71–82. DOI: 10.1109/SEAMS.2015.12.

[6] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures." In: *International Conference on Autonomic Computing*. ICAC '17. Karsten Schwan Best Paper Award. IEEE, 2017, pp. 59–68. DOI: 10.1109/ICAC.2017.35.

[7] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Towards Linking Adaptation Rules to the Utility Function for Dynamic Architectures." In: *International Conference on Self-Adaptive and Self-Organizing Systems*. SASO '16. IEEE, 2016, pp. 142–143. DOI: 10.1109/SASO.2016.21.

[8] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. "Graph Transformations for MDE, Adaptation, and Models at Runtime." In: *Formal Methods for Model-Driven Engineering*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Vol. 7320. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 137–191. DOI: 10.1007/978-3-642-30982-3_5.

[9] Holger Giese, Andreas Seibel, and Thomas Vogel. "A Model-Driven Configuration Management System for Advanced IT Service Management." In: *International Workshop on Models@run.time*. Vol. 509. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 61–70. URL: http://ceur-ws.org/Vol-509/.

[10] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, and Kirstie Bellman. "Generic Architectures for Individual Self-Aware Computing Systems." In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Springer, 2017. Chap. 6, pp. 149–189. DOI: 10.1007/978-3-319-47474-8_6.

[11] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, Nelly Bencomo, Kurt Geihs, Samuel Kounev, and Kirstie Bellman. "State of the Art in Architectures for Self-Aware Computing Systems." In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Springer, 2017. Chap. 8, pp. 237–275. DOI: 10.1007/978-3-319-47474-8_8.

[12] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, and Samuel Kounev. "Architectural Concepts for Self-Aware Computing Systems." In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Springer, 2017. Chap. 5, pp. 109–147. DOI: 10.1007/978-3-319-47474-8_5.

[13] Holger Giese, Thomas Vogel, and Sebastian Wätzoldt. "Towards Smart Systems of Systems." In: *International Conference on Fundamentals of Software Engineering (FSEN '15)*. Vol. 9392. Lecture Notes in Computer Science (LNCS). (invited paper). Springer, 2015, pp. 1–29. DOI: 10.1007/978-3-319-24644-4_1.

[14] Joachim Hänsel, Thomas Vogel, and Holger Giese. "A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models." In: *International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. SASOW '15. IEEE, 2015, pp. 134–139. DOI: 10.1109/SASOW.2015.27.

[15] Rogério de Lemos, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, Yuriy Brun, Javier Camara, Radu Calinescu, Myra B. Cohen, Alessandra Gorla, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jezequel, Sam Malek, Raffaela Mirandola, Marco Mori, Hausi A. Müller, Romain Rouvoy, Cecilia M. F. Rubira, Eric Rutten, Mary Shaw, Giordano Tamburrelli, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, and Franco Zambonelli. "Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances." In: *Software Engineering for Self-Adaptive Systems III. Assurances*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Vol. 9640. Lecture Notes in Computer Science (LNCS). Springer, 2017, pp. 3–30. DOI: 10.1007/978-3-319-74183-3_1.

[16] Rogério de Lemos, Holger Giese, Hausi Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, Joao P. Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. "Software Engineering for Self-Adaptive Systems: A second Research Roadmap." In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw. Vol. 7475. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 1–32. DOI: 10.1007/978-3-642-35813-5_1.

[17] Bradley Schmerl, Jesper Andersson, Thomas Vogel, Myra B. Cohen, Cecilia M. F. Rubira, Yuriy Brun, Alessandra Gorla, Franco Zambonelli, and Luciano Baresi. "Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems III: Assurances*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Vol. 9640. Lecture Notes in Computer Science (LNCS). Springer, 2017, pp. 64–89. DOI: 10.1007/978-3-319-74183-3_3.

[18] Thomas Vogel. *Modular Rice University Bidding System (mRUBiS)*. http://www.mdelab.de [Online; accessed 18-January-2017]. In the meantime after submitting this thesis, mRUBiS has been accepted as an exemplar [19]. 2013.

[19] Thomas Vogel. "mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '18. ACM, 2018. DOI: 10.1145/3194133.3194161.

[20] Thomas Vogel, Jens Bruhn, and Guido Wirtz. "Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications." In: *International Conference on Software Engineering and Knowledge Engineering*. SEKE'08. Knowledge Systems Institute Graduate School, 2008, pp. 48–53.

[21] Thomas Vogel and Holger Giese. "A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE, 2012, pp. 129–138. DOI: 10.1109/SEAMS.2012.6224399.

[22] Thomas Vogel and Holger Giese. "Adaptation and Abstract Runtime Models." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. ACM, 2010, pp. 39–48. DOI: 10.1145/1808984.1808989.

[23] Thomas Vogel and Holger Giese. "Language and Framework Requirements for Adaptation Models." In: *International Workshop on Models@run.time*. Vol. 794. CEUR Workshop Proceedings. (best paper). CEUR-WS.org, 2011, pp. 1–12. URL: http://ceur-ws.org/Vol-794/.

[24] Thomas Vogel and Holger Giese. *Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software: Executable Runtime Megamodels*. Tech. rep. 66. Hasso Plattner Institute at the University of Potsdam, Germany, 2013. URL: http://opus.kobv.de/ubp/volltexte/2013/6382/.

[25] Thomas Vogel and Holger Giese. "Model-Driven Engineering of Self-Adaptive Software with EUREMA." In: *ACM Trans. Auton. Adapt. Syst.* 8.4 (2014), 18:1–18:33. DOI: 10.1145/2555612.

[26] Thomas Vogel and Holger Giese. "On Unifying Development Models and Runtime Models." In: *International Workshop on Models@run.time*. Vol. 1270. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 5–10. URL: http://ceur-ws.org/Vol-1270/.

[27] Thomas Vogel and Holger Giese. "Requirements and Assessment of Languages and Frameworks for Adaptation Models." In: *Models in Software Engineering, Workshops and Symposia at MoDELS 2011, Reports and Revised Selected Papers*. Ed. by Jörg Kienzle. Vol. 7167. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 167–182. DOI: 10.1007/978-3-642-29645-1_18.

[28] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. "Incremental Model Synchronization for Efficient Run-Time Monitoring." In: *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers*. Ed. by Sudipto Ghosh. Vol. 6002. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 124–139. DOI: 10.1007/978-3-642-12261-3_13.

[29] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. "Incremental Model Synchronization for Efficient Run-time Monitoring." In: *International Workshop on Models@run.time*. Vol. 509. CEUR Workshop Proceedings. (best paper). CEUR-WS.org, 2009, pp. 1–10. URL: http://ceur-ws.org/Vol-509/.

[30] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. "Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems." In: *International Conference on Autonomic Computing and Communications*. ICAC '09. ACM, 2009, pp. 67–68. DOI: 10.1145/1555228.1555249.

[31] Thomas Vogel, Andreas Seibel, and Holger Giese. "The Role of Models and Megamodels at Runtime." In: *Models in Software Engineering, Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*. Ed. by Juergen Dingel and Arnor Solberg. Vol. 6627. Lecture Notes in Computer Science (LNCS). Springer, 2011, pp. 224–238. DOI: 10.1007/978-3-642-21210-9_22.

[32] Thomas Vogel, Andreas Seibel, and Holger Giese. "Toward Megamodels at Runtime." In: *International Workshop on Models@run.time*. Vol. 641. CEUR Workshop Proceedings. (best paper). CEUR-WS.org, 2010, pp. 13–24. URL: http://ceur-ws.org/Vol-641/.

[33] Thomas Vogel, Matthias Tichy, and Alessandra Gorla, eds. *Report from the GI Dagstuhl Seminar 14433: Software Engineering for Self-Adaptive Systems*. Research Reports in Software Engineering and Management 2014:02. Department of Computer Science, Engineering, Chalmers University of Technology, and University of Gothenburg, Sweden. 2014. URL: http://hdl.handle.net/2077/37775.

REFERENCES

[34] Dhaminda B. Abeywickrama, Nicklas Hoch, and Franco Zambonelli. "Engineering and implementing software architectural patterns based on feedback loops." In: *Scalable Computing: Practice and Experience* 15.4 (2015), pp. 291–307. DOI: 10.12694/scpe.v15i4.1052.

[35] Dhaminda B. Abeywickrama, Nicklas Hoch, and Franco Zambonelli. "SimSOTA: Engineering and Simulating Feedback Loops for Self-adaptive Systems." In: *International C* Conference on Computer Science and Software Engineering*. C3S2E '13. ACM, 2013, pp. 67–76. DOI: 10.1145/2494444.2494446.

[36] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. "Modeling Context and Dynamic Adaptations with Feature Models." In: *International Workshop on Models@run.time*. Vol. 509. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 89–98. URL: http://ceur-ws.org/Vol-509/.

[37] Jakub Adamczyk, Marcin Chojnacki Rafałand Jarząb, and Krzysztof Zieliński. "Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing." In: *International Conference on Computational Science (ICCS '08)*. Ed. by Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot. Vol. 5101. Lecture Notes in Computer Science (LNCS). Springer, 2008, pp. 355–364. DOI: 10.1007/978-3-540-69384-0_41.

[38] Gul A. Agha. "Adaptive Middleware (Introduction)." In: *Commun. ACM* 45.6 (2002), pp. 30–32. DOI: 10.1145/508448.508469.

[39] Germán H. Alférez and Vicente Pelechano. "Dynamic Evolution of Context-Aware Systems with Models at Runtime." In: *International Conference on Model Driven Engineering Languages and Systems (MODELS '12)*. Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Vol. 7590. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 70–86. DOI: 10.1007/978-3-642-33666-9_6.

[40] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. "Achieving dynamic adaptation via management and interpretation of runtime models." In: *Journal of Systems and Software* 85.12 (2012), pp. 2720–2737. DOI: 10.1016/j.jss.2012.05.033.

[41] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. "Reflecting on self-adaptive software systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 38–47. DOI: 10.1109/SEAMS.2009.5069072.

[42] Suzana Andova, Mark G.J. van den Brand, Luc J.P. Engelen, and Tom Verhoeff. "MDE Basics with a DSL Focus." In: *Formal Methods for Model-Driven Engineering*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Vol. 7320. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 21–57. DOI: 10.1007/978-3-642-30982-3_2.

[43] Konstantinos Angelopoulos, Vítor E. Silva Souza, and João Pimentel. "Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. IEEE, 2013, pp. 23–32. DOI: 10.1109/SEAMS.2013.6595489.

[44] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Formal Design and Verification of Self-Adaptive Systems with Decentralized Control." In: *ACM Trans. Auton. Adapt. Syst.* 11.4 (2017), 25:1–25:35. DOI: 10.1145/3019598.

[45] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Modeling and Analyzing MAPE-K Feedback Loops for Self-adaptation." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. IEEE, 2015, pp. 13–23. DOI: 10.1109/SEAMS.2015.10.

[46] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Modeling and Validating Self-adaptive Service-oriented Applications." In: *SIGAPP Appl. Comput. Rev.* 15.3 (2015), pp. 35–48. DOI: 10.1145/2835260.2835262.

[47] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations." In: *International Conference on Model Driven Engineering Languages and Systems (MODELS '10)*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

[48] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. "StarMX: A framework for developing self-managing Java-based systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 58–67. DOI: 10.1109/SEAMS.2009.5069074.

[49] Uwe Assmann, Nelly Bencomo, Betty H. C. Cheng, and Robert B. France. "Models@run.time (Dagstuhl Seminar 11481)." In: *Dagstuhl Reports*. Vol. 1. 11. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 91–123. DOI: 10.4230/DagRep.1.11.91.

[50] Karl J. Åström and Björn Wittenmark. *Adaptive Control*. 2nd ed. Reprint of the Addison-Wesley, Reading Massachusetts, 1995 second edition. Dover Publications Inc., 2008. ISBN: 0486462781.

[51] Colin Atkinson and Thomas Kühne. "Model-Driven Development: A Metamodeling Foundation." In: *IEEE Software* 20.5 (2003), pp. 36–41. DOI: 10.1109/MS.2003.1231149.

[52] Moritz Balz, Michael Striewe, and Michael Goedicke. "Monitoring Model Specifications in Program Code Patterns." In: *International Workshop on Models@run.time*. Vol. 641. CEUR Workshop Proceedings. CEUR-WS.org, 2010, pp. 60–71. URL: http://ceur-ws.org/Vol-641/.

[53] Mikael Barbero, Marcos Didonet Fabro, and Jean Bézivin. "Traceability and Provenance Issues in Global Model Management." In: *Intnernational Workshop on Traceability*. Ed. by Jon Oldevik, Göran K. Olsen, and Tor Neple. ECMDA-TW '07. SINTEF. 2007, pp. 47–55.

[54] Franck Barbier, Eric Cariou, Olivier Le Goaer, and Samson Pierre. "Software Adaptation: Classification and a Case Study with State Chart XML." In: *IEEE Software* 32.5 (2015), pp. 68–76. DOI: 10.1109/MS.2014.130.

[55] Luciano Baresi and Carlo Ghezzi. "The Disappearing Boundary Between Development-time and Run-time." In: *Workshop on Future of Software Engineering Research*. FoSER '10. ACM, 2010, pp. 17–22. DOI: 10.1145/1882362.1882367.

[56] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma, and Valerio Panzica La Manna. "Efficient Dynamic Updates of Distributed Components through Version Consistency." In: *IEEE Transactions on Software Engineering* 43.4 (2017), pp. 340–358. DOI: 10.1109/TSE.2016.2592913.

[57] Luciano Baresi and Reiko Heckel. "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective." In: *Graph Transformation*. Ed. by Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 2505. Lecture Notes in Computer Science (LNCS). Springer, 2002, pp. 402–429. DOI: 10.1007/3-540-45832-8_30.

[58] Luciano Baresi and Liliana Pasquale. "Live Goals for Adaptive Service Compositions." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. ACM, 2010, pp. 114–123. DOI: 10.1145/1808984.1808997.

[59] Björn Bartels and Moritz Kleine. "A CSP-based Framework for the Specification, Verification, and Implementation of Adaptive Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, 2011, pp. 158–167. DOI: 10.1145/1988008.1988030.

[60] Kent Beck. "The Inevitability of Evolution." In: *IEEE Software* 27.4 (2010), pp. 28–29. DOI: 10.1109/MS.2010.103.

[61] Matthias Becker, Markus Luckey, and Steffen Becker. "Performance Analysis of Self-adaptive Systems for Requirements Validation at Design-time." In: *International Conference on Quality of Software Architectures*. QoSA '13. ACM, 2013, pp. 43–52. DOI: 10.1145/2465478.2465489.

[62] Nelly Bencomo. "On the Use of Software Models During Software Execution." In: *International Workshop on Modeling in Software Engineering*. MISE '09. IEEE, 2009, pp. 62–67. DOI: 10.1109/MISE.2009.5069899.

[63] Nelly Bencomo and Gordon Blair. "Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 183–200. DOI: 10.1007/978-3-642-02161-9_10.

[64] Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, eds. *Models@run.time – Foundations, Applications, and Roadmaps*. Vol. 8378. Lecture Notes in Computer Science (LNCS). Springer, 2014. DOI: 10.1007/978-3-319-08915-7.

[65] Nelly Bencomo, Jon Whittle, Pete Sawyer, Anthony Finkelstein, and Emmanuel Letier. "Requirements Reflection: Requirements As Runtime Entities." In: *International Conference on Software Engineering - Volume 2*. ICSE '10. ACM, 2010, pp. 199–202. DOI: 10.1145/1810295.1810329.

[66] Keith H. Bennett and Václav T. Rajlich. "Software Maintenance and Evolution: A Roadmap." In: *Future of Software Engineering*. FOSE '00. ACM, 2000, pp. 73–87. DOI: 10.1145/336512.336534.

[67] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. "Viatra 3: A Reactive Model Transformation Platform." In: *International Conference on Theory and Practice of Model Transformations*. Ed. by Dimitris Kolovos and Manuel Wimmer. ICMT '15. Springer, 2015, pp. 101–110. DOI: 10.1007/978-3-319-21155-8_8.

[68] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. "Incremental Evaluation of Model Queries over EMF Models." In: *International Conference on Model Driven Engineering Languages and Systems (MODELS '10)*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 76–90. DOI: 10.1007/978-3-642-16145-2_6.

[69] Mohamed Lamine Berkane, Lionel Seinturier, and Mahmoud Boufaida. "Using variability modelling and design patterns for self-adaptive system engineering: application to smart-home." In: *Web Engineering and Technology* 10.1 (2015), pp. 65–93. DOI: 10.1504/IJWET.2015.069359.

[70] Jean Bézivin. "Model Driven Engineering: An Emerging Technical Space." In: *Generative and Transformational Techniques in Software Engineering*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science (LNCS). Springer, 2006, pp. 36–64. DOI: 10.1007/11877028_2.

[71] Jean Bézivin. "On the unification power of models." In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0.

[72] Jean Bézivin, Sebastian Gerard, Pierre-Alain Muller, and Laurent Rioux. "MDA components: Challenges and Opportunities." In: *International Workshop on Metamodelling for MDA*. Ed. by Andy Evans, Paul Sammut, and James S. Willans. 2003, pp. 23–41.

[73] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. "On the Need for Megamodels." In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004. URL: http://atlanmod.emn.fr/www/papers/OOPSLA04/bezivin-megamodel.pdf.

[74] Gordon S. Blair, Fábio Costa, Geoff Coulson, Fabien Delpiano, Hector Duran, Bruno Dumant, François Horn, Nikos Parlavantzas, and Jean-Bernard Stefani. "The Design of a Resource-Aware Reflective Middleware Architecture." In: *Meta-Level Architectures and Reflection*. Ed. by Pierre Cointe. Vol. 1616. Lecture Notes in Computer Science (LNCS). Springer, 1999, pp. 115–134. DOI: 10.1007/3-540-48443-4_9.

[75] Gordon S. Blair and Geoff Coulson. *The case for reflective middleware*. Tech. rep. MPG-98-38. Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1998.

[76] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. "The Design and Implementation of Open ORB 2." In: *IEEE Distributed Systems Online* 2.6 (2001).

[77] Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. "Reflection, Self-awareness and Self-healing in OpenORB." In: *Workshop on Self-healing Systems*. WOSS '02. ACM, 2002, pp. 9–14. DOI: 10.1145/582128.582131.

[78] Gordon S. Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas. "An Architecture for Next Generation Middleware." In: *International Conference on Distributed Systems Platforms and Open Distributed Processing*. Ed. by Nigel Davies, Seitz Jochen, and Kerry Raymond. Middleware '98. Springer, 1998, pp. 191–206. DOI: 10.1007/978-1-4471-1283-9_12.

[79] Gordon Blair, Nelly Bencomo, and Robert B. France. "Models@run.time." In: *IEEE Computer* 42.10 (2009), pp. 22–27. DOI: doi:10.1109/MC.2009.326.

[80] José Bocanegra, Jaime Pavlich-Mariscal, and Angela Carrillo-Ramos. "DMLAS: A Domain-Specific Language for Designing Adaptive Systems." In: *Computing Colombian Conference*. CCC '15. IEEE, 2015, pp. 47–54. DOI: 10.1109/ColumbianCC.2015.7333411.

[81] Barry Boehm. "The Changing Nature of Software Evolution." In: *IEEE Software* 27.4 (2010), pp. 26–28. DOI: 10.1109/MS.2010.103.

[82] Fabienne Boyer, Olivier Gruber, and Damien Pous. "Robust Reconfigurations of Component Assemblies." In: *International Conference on Software Engineering*. ICSE '13. IEEE, 2013, pp. 13–22. DOI: 10.1109/ICSE.2013.6606547.

[83] Víctor A. Braberman, Nicolás D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastián Uchitel. "MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation." In: *CoRR / ArXiv e-prints* abs/1504.08339 (2015). URL: http://arxiv.org/abs/1504.08339.

[84] Víctor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. "MORPH: A Reference Architecture for Configuration and Behaviour Self-adaptation." In: *International Workshop on Control Theory for Software Engineering*. CTSE '15. ACM, 2015, pp. 9–16. DOI: 10.1145/2804337.2804339.

[85] Víctor Braberman, Nicolas D'Ippolito, Nir Piterman, Daniel Sykes, and Sebastian Uchitel. "Controller Synthesis: From Modelling to Enactment." In: *International Conference on Software Engineering*. ICSE '13. IEEE, 2013, pp. 1347–1350. DOI: 10.1109/ICSE.2013.6606714.

[86] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. "A Survey of Self-management in Dynamic Software Architecture Specifications." In: *Workshop on Self-managed Systems*. WOSS '04. ACM, 2004, pp. 28–33. DOI: 10.1145/1075405.1075411.

[87] Frances M.T. Brazier, Jeffrey O. Kephart, H. Van Dyke Parunak, and Michael N. Huhns. "Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda." In: *IEEE Internet Computing* 13.3 (2009), pp. 82–87. DOI: 10.1109/MIC.2009.51.

[88] Frederick P. Brooks Jr. "No Silver Bullet Essence and Accidents of Software Engineering." In: *IEEE Computer* 20.4 (1987), pp. 10–19. DOI: 10.1109/MC.1987.1663532.

[89] Alan W. Brown. "Model driven architecture: Principles and practice." In: *Software and Systems Modeling* 3.4 (2004), pp. 314–327. DOI: 10.1007/s10270-004-0061-2.

[90] Jens Bruhn. "A Realistic Approach for the Autonomic Management of Component-Based Enterprise Systems." PhD thesis. 2009. URL: http://nbn-resolving.de/urn:nbn:de:bvb:473-opus-1986.

[91] Jens Bruhn and Guido Wirtz. "mKernel: A Manageable Kernel for EJB-based Systems." In: *International Conference on Autonomic Computing and Communication Systems*. Autonomics '07. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, 13:1–13:10. URL: http://dl.acm.org/citation.cfm?id=1365562.1365580.

[92] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit. "A Design Space for Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, HausiA. Müller, and Mary Shaw. Vol. 7475. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 33–50. DOI: 10.1007/978-3-642-35813-5_2.

[93] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. "Engineering Self-Adaptive Systems through Feedback Loops." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 48–70. DOI: 10.1007/978-3-642-02161-9_3.

[94] Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter Clarke, Robert France, and Gabor Karsai. "Challenges and Directions in Formalizing the Semantics of Modeling Languages." In: *Computer Science and Information Systems (ComSIS)* 8.2 (2011), pp. 225–253. DOI: 10.2298/CSIS110114012B.

[95] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. "Towards a taxonomy of software change." In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.5 (2005), pp. 309–332. DOI: 10.1002/smr.319.

[96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. John Wiley & Sons Ltd, 1996. ISBN: 0471958697.

[97] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. "Self-adaptive Software Needs Quantitative Verification at Runtime." In: *Commun. ACM* 55.9 (2012), pp. 69–77. DOI: 10.1145/2330667.2330686.

[98] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. "Dynamic QoS Management and Optimization in Service-Based Systems." In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 387–409. DOI: 10.1109/TSE.2010.92.

[99] Javier Cámara and Rogério de Lemos. "Evaluation of Resilience in Self-adaptive Systems Using Probabilistic Model-checking." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE, 2012, pp. 53–62. DOI: 10.1109/SEAMS.2012.6224391.

[100] Matteo Camilli, Angelo Gargantini, and Patrizia Scandurra. "Specifying and verifying real-time self-adaptive systems." In: *International Symposium on Software Reliability Engineering*. ISSRE '15. IEEE, 2015, pp. 303–313. DOI: 10.1109/ISSRE.2015.7381823.

[101] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. "Model-based System Reconfiguration for Dynamic Performance Management." In: *Journal of Systems and Software* 80.4 (2007), pp. 455–473. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.07.039.

[102] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, and Theo DHondt. "Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets." In: *International Symposium on Theoretical Aspects of Software Engineering*. TASE '13. IEEE, 2013, pp. 191–198. DOI: 10.1109/TASE.2013.33.

[103] Eric Cariou, Franck Barbier, and Olivier Le Goaer. "Model execution adaptation?" In: *International Workshop on Models@run.time*. ACM, 2012, pp. 60–65. DOI: 10.1145/2422518.2422528.

[104] Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu. "Diagnosing Architectural Run-time Failures." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. IEEE, 2013, pp. 103–112. DOI: 10.1109/SEAMS.2013.6595497.

[105] Manuel Castells. *The Information Age: Economy, Society and Culture Volume 1: The Rise of the Network Society*. 2nd ed. Wiley-Blackwell, 2010. ISBN: 1405196864.

[106] Luca Cavallaro, Pete Sawyer, Daniel Sykes, Nelly Bencomo, and Valérie Issarny. "Satisfying Requirements for Pervasive Service Compositions." In: *International Workshop on Models@run.time*. ACM, 2012, pp. 17–22. DOI: 10.1145/2422518.2422522.

[107] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes." In: *IEEE Computer* 42.10 (2009), pp. 37–43. DOI: 10.1109/MC.2009.309.

[108] Ned Chapin. "Do We Know What Preventive Maintenance Is?" In: *International Conference on Software Maintenance*. ICSM '00. IEEE, 2000, pp. 15–17. DOI: 10.1109/ICSM.2000.882970.

[109] Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh, and Wenyun Zhao. "Self-adaptation Through Incremental Generative Model Transformations at Runtime." In: *International Conference on Software Engineering*. ICSE '14. ACM, 2014, pp. 676–687. DOI: 10.1145/2568225.2568310.

[110] Huoping Chen and Salim Hariri. "An Evaluation Scheme of Adaptive Configuration Techniques." In: *International Conference on Automated Software Engineering*. ASE '07. ACM, 2007, pp. 493–496. DOI: 10.1145/1321631.1321717.

[111] Betty H. C. Cheng and Joanne M. Atlee. "Research Directions in Requirements Engineering." In: *Future of Software Engineering*. FOSE '07. IEEE, 2007, pp. 285–303. DOI: 10.1109/FOSE.2007.17.

[112] Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. "Software Engineering for Self-Adaptive Systems: A Research Roadmap." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 1–26. DOI: 10.1007/978-3-642-02161-9_1.

[113] Shang-Wen Cheng. "Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation." Institute for Software Research Technical Report CMU-ISR-08-113. PhD thesis. School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 2008. URL: http://reports-archive.adm.cs.cmu.edu/anon/isr2008/abstracts/08-113.html.

[114] Shang-Wen Cheng and David Garlan. "Stitch: A Language for Architecture-Based Self-Adaptation." In: *Journal of Systems and Software* 85.12 (2012), pp. 2860–2875. DOI: 10.1016/j.jss.2012.02.060.

[115] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. "Architecture-based Self-adaptation in the Presence of Multiple Objectives." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '06. ACM, 2006, pp. 2–8. DOI: 10.1145/1137677.1137679.

[116] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. "Evaluating the effectiveness of the Rainbow self-adaptive system." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 132–141. DOI: 10.1109/SEAMS.2009.5069082.

[117] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. "An Architecture for Coordinating Multiple Self-Management Systems." In: *Working Conference on Software Architecture*. WICSA '04. IEEE, 2004, pp. 243–252. DOI: 10.1109/WICSA.2004.1310707.

[118] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development*. 3rd edition. 2015. URL: http://arxiv.org/abs/1505.00149.

[119] Kevin Colson, Robin Dupuis, Lionel Montrieux, Zhenjiang Hu, Sebastián Uchitel, and Pierre-Yves Schobbens. "Reusable Self-adaptation Through Bidirectional Programming." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '16. ACM, 2016, pp. 4–15. DOI: 10.1145/2897053.2897055.

[120] Benoît Combemale, Laurent Broto, Xavier Crégut, Michel Daydé, and Daniel Hagimont. "Autonomic Management Policy Specification: From UML to DSML." In: *International Conference on Model Driven Engineering Languages and Systems (MoDELS '08)*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Vol. 5301. Lecture Notes in Computer Science (LNCS). Springer, 2008, pp. 584–599. DOI: 10.1007/978-3-540-87875-9_41.

[121] Reidar Conradi and Bernhard Westfechtel. "Version Models for Software Configuration Management." In: *ACM Comput. Surv.* 30.2 (1998), pp. 232–282. DOI: 10.1145/280277.280280.

[122] Jonathan E. Cook and Jeffrey A. Dage. "Highly Reliable Upgrading of Components." In: *International Conference on Software Engineering*. ICSE '99. ACM, 1999, pp. 203–212. DOI: 10.1145/302405.302466.

[123] Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. "Evolution in software systems: foundations of the SPE classification scheme." In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.1 (2006), pp. 1–35. DOI: 10.1002/smr.314.

[124] IBM Corporation. *An architectural blueprint for autonomic computing*. White Paper, Fourth Edition. 2006.

[125] Fabio Costa, Lucas Provensi, and Frederico Vaz. "Towards a More Effective Coupling of Reflection and Runtime Metamodels for Middleware." In: *International Workshop on Models@run.time*. 2006. URL: https://st.inf.tu-dresden.de/MRT06/accepted.html.

[126] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. 5th. Addison-Wesley, 2011. URL: http://www.cdk5.net.

[127] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. "Towards Architecture-based Self-healing Systems." In: *Workshop on Self-healing Systems*. WOSS '02. ACM, 2002, pp. 21–26. DOI: 10.1145/582128.582133.

[128] Dylan Dawson, Ron Desmarais, Holger M. Kienle, and Hausi A. Müller. "Monitoring in Adaptive Systems Using Reflection." In: *International Workshop on Software Engineering for Adaptive and Self-managing Systems*. SEAMS '08. ACM, 2008, pp. 81–88. DOI: 10.1145/1370018.1370033.

[129] Frederico Alvares De Oliveira, Remi Sharrock, and Thomas Ledoux. "Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing." In: *International Conference on Coordination Models and Languages (COORDINATION '12)*. Ed. by Marjan Sirjani. Vol. 7274. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 29–43. DOI: 10.1007/978-3-642-30829-1_3.

[130] Tom De Wolf and Tom Holvoet. "Designing Self-Organising Emergent Systems based on Information Flows and Feedback-loops." In: *International Conference on Self-Adaptive and Self-Organizing Systems*. SASO '07. IEEE, 2007, pp. 295–298. DOI: 10.1109/SASO.2007.16.

[131] Tom De Wolf and Tom Holvoet. "Using UML 2 activity diagrams to design information flows and feedback-loops in self-organising emergent systems." In: *International Workshop on Engineering Emergence in Decentralised Autonomic Systems*. Ed. by Tom De Wolf, Fabrice Saffre, and Richard J. Anthony. EEDAS '07. 2007, pp. 52–61. URL: https://distrinet.cs.kuleuven.be/events/eedas/2007/papers.php.

[132] Linda DeMichiel and Michael Keith. *JSR 220: Enterprise JavaBeans™, Version 3.0, EJB Core Contracts and Requirements*. JSR-220. 2006. URL: https://jcp.org/en/jsr/detail?id=220.

[133] Mahdi Derakhshanmanesh, Mehdi Amoui, Greg O'Grady, Jürgen Ebert, and Ladan Tahvildari. "GRAF: Graph-based Runtime Adaptation Framework." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, 2011, pp. 128–137. DOI: 10.1145/1988008.1988026.

[134] Mahdi Derakhshanmanesh, Jürgen Ebert, Thomas Iguchi, and Gregor Engels. "Model-Integrating Software Components." In: *International Conference on Model-Driven Engineering Languages and Systems (MODELS '14)*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Vol. 8767. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 386–402. DOI: 10.1007/978-3-319-11653-2_24.

[135] Mahdi Derakhshanmanesh and Marvin Grieger. "On Enabling Technologies for Longevity in Software." In: *Software Engineering Workshops 2015 (Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems (EMLS'15))*. Vol. 1337. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 112–114. URL: http://ceur-ws.org/Vol-1337/.

[136] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific Languages: An Annotated Bibliography." In: *SIGPLAN Not.* 35.6 (2000), pp. 26–36. DOI: 10.1145/352029.352035.

[137] Zuohua Ding, Yuan Zhou, and MengChu Zhou. "Modeling Self-adaptive Software Systems with Learning Petri Nets." In: *Companion of International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: ACM, 2014, pp. 464–467. DOI: 10.1145/2591062.2591113.

[138] Nicolas D'Ippolito, Víctor Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastian Uchitel. "Hope for the Best, Prepare for the Worst: Multi-tier Control for Adaptive Systems." In: *International Conference on Software Engineering*. ICSE '14. ACM, 2014, pp. 688–699. DOI: 10.1145/2568225.2568264.

[139] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaiti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. "A Survey of Autonomic Communications." In: *ACM Trans. Auton. Adapt. Syst.* 1.2 (2006), pp. 223–259. DOI: 10.1145/1186778.1186782.

[140] Jérémy Dubus and Philippe Merle. "Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-Based Systems." In: *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Reports and Revised Selected Papers*. Ed. by Thomas Kühne. Vol. 4364. Lecture Notes in Computer Science (LNCS). Springer, 2007, pp. 242–251. DOI: 10.1007/978-3-540-69489-2_30.

[141] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Sukhat Gaurav, and Brad Petrus. "Architecture-driven self-adaptation and self-management in robotics systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 142–151. DOI: 10.1109/SEAMS.2009.5069083.

[142] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. "Design of embedded systems: formal models, validation, and synthesis." In: *Proceedings of the IEEE* 85.3 (1997), pp. 366–390. DOI: 10.1109/5.558710.

[143] Jens Ehlers and Wilhelm Hasselbring. "A Self-adaptive Monitoring Framework for Component-Based Software Systems." In: *Software Architecture*. Ed. by Ivica Crnkovic, Volker Gruhn, and Matthias Book. Vol. 6903. Lecture Notes in Computer Science (LNCS). Springer, 2011, pp. 278–286. DOI: 10.1007/978-3-642-23798-0_30.

[144] Jens Ehlers, Andre van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Self-adaptive Software System Monitoring for Performance Anomaly Localization." In: *International Conference on Autonomic Computing*. ICAC '11. ACM, 2011, pp. 197–200. DOI: 10.1145/1998582.1998628.

[145] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. "FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems." In: *International Symposium on Foundations of Software Engineering*. FSE '10. ACM, 2010, pp. 7–16. DOI: 10.1145/1882291.1882296.

[146] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. "Model Evolution by Run-time Parameter Adaptation." In: *International Conference on Software Engineering*. ICSE '09. IEEE, 2009, pp. 111–121. DOI: 10.1109/ICSE.2009.5070513.

[147] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. "A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems." In: *IEEE Transactions on Software Engineering* 39.11 (2013), pp. 1467–1493. DOI: 10.1109/TSE.2013.37.

[148] Naeem Esfahani and Sam Malek. "Uncertainty in Self-Adaptive Software Systems." In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Vol. 7475. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 214–238. DOI: 10.1007/978-3-642-35813-5_9.

[149] Jean-Marie Favre. "Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus." In: *Language Engineering for Model-Driven Software Development*. Ed. by Jean Bézivin and Reiko Heckel. Dagstuhl Seminar Proceedings 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: http://drops.dagstuhl.de/opus/volltexte/2005/13.

[150] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. "Run-time Efficient Probabilistic Model Checking." In: *International Conference on Software Engineering*. ICSE '11. ACM, 2011, pp. 341–350. DOI: 10.1145/1985793.1985840.

[151] Antonio Filieri, Lars Grunske, and Alberto Leva. "Lightweight Adaptive Filtering for Efficient Learning and Updating of Probabilistic Models." In: *International Conference on Software Engineering*. ICSE '15. IEEE, 2015, pp. 200–211. DOI: 10.1109/ICSE.2015.41.

[152] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees." In: *International Conference on Software Engineering*. ICSE '14. ACM, 2014, pp. 299–310. DOI: 10.1145/2568225.2568272.

[153] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. "Supporting Self-adaptation via Quantitative Verification and Sensitivity Analysis at Run Time." In: *IEEE Transactions on Software Engineering* 42.1 (2016), pp. 75–99. DOI: 10.1109/TSE.2015.2421318.

[154] Wladyslaw Findeisen, Fred N. Bailey, Mieczyslaw Brdys, Krzysztof Malinowski, Piotr Tatjewski, and Adam Wozniak. *Control and Coordination in Hierarchical Systems*. International series on applied systems analysis. J. Wiley, 1980. URL: http://www.iiasa.ac.at/publication/more_XB-80-109.php.

[155] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. "Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java." In: *Theory and Application of Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 1764. Lecture Notes in Computer Science (LNCS). Springer, 2000, pp. 296–309. DOI: 10.1007/978-3-540-46464-8_21.

[156] Peter C. Fishburn. "Utility Theory." In: *Management Science* 14.5 (1968), pp. 335–378. DOI: 10.1287/mnsc.14.5.335.

[157] Camilo Fitzgerald, Benjamin Klöpper, and Shinichi Honiden. "Utility-Based Self-Adaption with Environment Specific Quality Models." In: *International Conference on Adaptive and Intelligent Systems (ICAIS '11)*. Ed. by Abdelhamid Bouchachia. Vol. 6943. Lecture Notes in Computer Science (LNCS). Springer, 2011, pp. 107–118. DOI: 10.1007/978-3-642-23857-4_14.

[158] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. "Modeling and Validating Dynamic Adaptation." In: *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Reports and Revised Selected Papers*. Ed. by Michel R.V. Chaudron. Vol. 5421. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 97–108. DOI: 10.1007/978-3-642-01648-6_11.

[159] Franck Fleurey and Arnor Solberg. "A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems." In: *International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*. Ed. by Andy Schürr and Bran Selic. Vol. 5795. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 606–621. DOI: 10.1007/978-3-642-04425-0_47.

[160] Jacqueline Floch, Cristina Frà, Rolf Fricke, Kurt Geihs, Michael Wagner, Jorge Lorenzo, Eduardo Soladana, Stephan Mehlhase, Nearchos Paspallis, Hossein Rahnama, Pedro Antonio Ruiz Ruiz, and Ulrich Scholz. "Playing MUSIC – building context-aware and self-adaptive mobile applications." In: *Software: Practice and Experience* 43.3 (2013), pp. 359–388. DOI: 10.1002/spe.2116.

[161] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. "Using Architecture Models for Runtime Adaptability." In: *IEEE Software* 23.2 (2006), pp. 62–70. DOI: 10.1109/MS.2006.61.

[162] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010. ISBN: 0321712943. URL: http://martinfowler.com/books/dsl.html.

[163] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. "Model-Driven Development Using UML 2.0: Promises and Pitfalls." In: *IEEE Computer* 39.2 (2006), pp. 59–66. DOI: 10.1109/MC.2006.65.

[164] Robert France and Bernhard Rumpe. "Model-driven Development of Complex Software: A Research Roadmap." In: *Future of Software Engineering*. FOSE '07. IEEE, 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[165] Sylvain Frey, Ada Diaconescu, and Isabelle M. Demeure. "Architectural Integration Patterns for Autonomic Management Systems." In: *International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems*. EASe '12. 2012.

[166] Mathias Fritzsche, Hugo Brunelière, Bert Vanhooff, Yolande Berbers, Frédéric Jouault, and Wasif Gilani. "Applying Megamodelling to Model Driven Performance Engineering." In: *International Conference and Workshop on the Engineering of Computer Based Systems*. ECBS '09. IEEE, 2009, pp. 244–253. DOI: 10.1109/ECBS.2009.33.

[167] Cristina Gacek, Holger Giese, and Ethan Hadar. "Friends or Foes?: A Conceptual Analysis of Self-adaptation and It Change Management." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '08. ACM, 2008, pp. 121–128. DOI: 10.1145/1370018.1370040.

[168] Alan G. Ganek and Thomas A. Corbi. "The dawning of the autonomic computing era." In: *IBM Systems Journal* 42.1 (2003), pp. 5–18. DOI: 10.1147/sj.421.0005.

[169] David Garlan. "Software Architecture: A Travelogue." In: *Future of Software Engineering*. FOSE '14. ACM, 2014, pp. 29–39. DOI: 10.1145/2593882.2593886.

[170] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure." In: *IEEE Computer* 37.10 (2004), pp. 46–54. DOI: 10.1109/MC.2004.175.

[171] David Garlan and Bradley Schmerl. "Model-based Adaptation for Self-healing Systems." In: *Workshop on Self-healing Systems*. WOSS '02. ACM, 2002, pp. 27–32. DOI: 10.1145/582128.582134.

[172] David Garlan and Bradley Schmerl. "Using Architectural Models at Runtime: Research Challenges." In: *Software Architecture*. Ed. by Flavio Oquendo, Brian C. Warboys, and Ron Morrison. Vol. 3047. Lecture Notes in Computer Science (LNCS). Springer, 2004, pp. 200–205. DOI: 10.1007/978-3-540-24769-2_15.

[173] David Garlan, Bradley Schmerl, and Jichuan Chang. "Using Gauges for Architecture-Based Monitoring and Adaptation." In: *Working Conference on Complex and Dynamic Systems Architecture*. 2001. URL: http://repository.cmu.edu/compsci/690/.

[174] Erann Gat. "On Three-Layer Architectures." In: *Artificial Intelligence and Mobile Robots*. Ed. by David Kortenkamp, R. Peter Bonasso, and Robin Murphy. MIT/AAAI Press, 1997.

[175] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, Rolf Fricke, Eli Gjørven, Svein O. Hallsteinsen, Geir Horn, Owais Mohammad Khan, Alessandro Mamelli, George A. Papadopoulos, Nearchos Paspallis, Roland Reichle, and Erlend Stav. "A comprehensive solution for application-level adaptation." In: *Software: Practice and Experience* 39.4 (2009), pp. 385–422. DOI: 10.1002/spe.900.

[176] John C. Georgas, André van der Hoek, and Richard N. Taylor. "Using Architectural Models to Manage and Visualize Runtime Adaptation." In: *IEEE Computer* 42.10 (2009), pp. 52–60. DOI: 10.1109/MC.2009.335.

[177] John C. Georgas and Richard N. Taylor. "Policy-based Self-adaptive Architectures: A Feasibility Study in the Robotics Domain." In: *International Workshop on Software Engineering for Adaptive and Self-managing Systems*. SEAMS '08. ACM, 2008, pp. 105–112. DOI: 10.1145/1370018.1370038.

[178] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. "Self-organising Software Architectures for Distributed Systems." In: *Workshop on Self-healing Systems*. WOSS '02. ACM, 2002, pp. 33–38. DOI: 10.1145/582128.582135.

[179] Fatih Gey, Dimitri Van Landuyt, Stefan Walraven, and Wouter Joosen. "Feature Models at Run Time: Feature Middleware for Multi-tenant SaaS applications." In: *International Workshop on Models@run.time*. Vol. 1270. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 21–30. URL: http://ceur-ws.org/Vol-1270/.

[180] Carlo Ghezzi. "Evolution, Adaptation, and the Quest for Incrementality." In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Ed. by Radu Calinescu and David Garlan. Vol. 7539. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 369–379. DOI: 10.1007/978-3-642-34059-8_19.

[181] Carlo Ghezzi, Joel Greenyer, and Valerio Panzica La Manna. "Synthesizing Dynamically Updating Controllers from Changes in Scenario-based Specifications." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE, 2012, pp. 145–154. DOI: 10.1109/SEAMS.2012.6224401.

[182] Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio. "Runtime Monitoring of Functional Component Changes with Behavior Models." In: *International Workshop on Models@run.time*. Vol. 794. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 37–48. URL: http://ceur-ws.org/Vol-794/.

[183] Carlo Ghezzi, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. "Managing Non-functional Uncertainty via Model-driven Adaptivity." In: *International Conference on Software Engineering*. ICSE '13. IEEE, 2013, pp. 33–42. DOI: 10.1109/ICSE.2013.6606549.

[184] Debanjan Ghosh, Raj Shaman, H. Raghav Rao, and Shambhu Upadhyaya. "Self-healing systems – survey and synthesis." In: *Decision Support Systems in Emerging Economies* 42.4 (2007), pp. 2164–2185. DOI: 10.1016/j.dss.2006.06.011.

[185] Holger Giese and Stephan Hildebrandt. *Efficient Model Synchronization of Large-Scale Models*. Tech. rep. 28. Hasso Plattner Institute at the University of Potsdam, Germany, 2009. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:kobv:517-opus-29281.

[186] Holger Giese and Stephan Hildebrandt. "Incremental Model Synchronization for Multiple Updates." In: *International Workshop on Graph and Model Transformations*. GRaMoT '08. ACM, 2008, pp. 1–8. DOI: 10.1145/1402947.1402949.

[187] Holger Giese and Wilhelm Schäfer. "Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML." In: *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*. Ed. by Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes. Vol. 7740. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 152–186. DOI: 10.1007/978-3-642-36249-1_6.

[188] Holger Giese and Robert Wagner. "From model transformation to incremental bidirectional model synchronization." In: *Software and Systems Modeling* 8.1 (2009), pp. 21–43. DOI: 10.1007/s10270-008-0089-9.

[189] Tony Gjerlufsen, Mads Ingstrup, and Jesper Wolff Olsen. "Mirrors of meaning: supporting inspectable runtime models." In: *IEEE Computer* 42.10 (2009), pp. 61–68. DOI: 10.1109/MC.2009.325.

[190] Michael W. Godfrey and Daniel M. German. "The past, present, and future of software evolution." In: *Frontiers of Software Maintenance*. FoSM '08. IEEE, 2008, pp. 129–138. DOI: 10.1109/FOSM.2008.4659256.

[191] Michael W. Godfrey and Qiang Tu. "Evolution in Open Source Software: A Case Study." In: *International Conference on Software Maintenance*. ICSM '00. IEEE, 2000, pp. 131–142. DOI: 10.1109/ICSM.2000.883030.

[192] Heather J. Goldsby, Betty H.C. Cheng, and Ji Zhang. "AMOEBA-RT: Run-Time Verification of Adaptive Software." In: *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Reports and Revised Selected Papers*. Ed. by Holger Giese. Vol. 5002. Lecture Notes in Computer Science (LNCS). Springer, 2008, pp. 212–224. DOI: 10.1007/978-3-540-69073-3_23.

[193] Ian Gorton, Yan Liu, and Nihar Trivedi. "An Extensible, Lightweight Architecture for Adaptive J2EE Applications." In: *International Workshop on Software Engineering and Middleware*. SEM '06. ACM, 2006, pp. 47–54. DOI: 10.1145/1210525.1210537.

[194] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification, Java SE 8 Edition*. JSR-337, Version: 8. 2015. URL: https://jcp.org/en/jsr/detail?id=337.

[195] Sebastian Götz, Ilias Gerostathopoulos, Filip Krikava, Adnan Shahzada, and Romina Spalazzese. "Adaptive Exchange of Distributed Partial Models@Run.Time for Highly Dynamic Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. IEEE, 2015, pp. 64–70. DOI: 10.1109/SEAMS.2015.25.

[196] Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Assmann. "Runtime Variability Management for Energy-efficient Software by Contract Negotiation." In: *International Workshop on Models@run.time*. Vol. 794. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 61–72. URL: http://ceur-ws.org/Vol-794/.

[197] Thomas R.G. Green and Marian Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. DOI: 10.1006/jvlc.1996.0009.

[198] Jack Greenfield and Keith Short. "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools." In: *Companion of the International Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '03. ACM, 2003, pp. 16–27. DOI: 10.1145/949344.949348.

[199] Matthias Güdemann, Florian Nafz, Frank Ortmeier, Hella Seebach, and Wolfgang Reif. "A Specification and Construction Paradigm for Organic Computing Systems." In: *International Conference on Self-Adaptive and Self-Organizing Systems*. SASO '08. 2008, pp. 233–242. DOI: 10.1109/SASO.2008.66.

[200] Soguy Mak Karé Gueye, Noel De Palma, and Eric Rutten. "Coordinating Energy-aware Administration Loops Using Discrete Control." In: *International Conference on Autonomic and Autonomous Systems*. ICAS '12. IARIA, 2012, pp. 99–106. URL: http://www.thinkmind.org/index.php?view=article&articleid=icas_2012_5_10_20027.

[201] Daniel Hagimont, Estella Annoni, Jean-Paul Bahsoun, Benoit Combemale, and Laurent Broto. "Towards a Model Driven Autonomic Management System." In: *International Conference on Information Technology: New Generation*. ITNG '08. IEEE, 2008, pp. 63–69. DOI: 10.1109/ITNG.2008.247.

[202] Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George A. Papadopoulos. "A development framework and methodology for self-adapting applications in ubiquitous computing environments." In: *Journal of Systems and Software* 85.12 (2012), pp. 2840–2859. DOI: 10.1016/j.jss.2012.07.052.

[203] Lars Hamann, Martin Gogolla, and Daniel Honsel. "Towards Supporting Multiple Execution Environments for UML/OCL Models at Runtime." In: *International Workshop on Models@run.time*. ACM, 2012, pp. 46–51. DOI: 10.1145/2422518.2422526.

[204] Deshuai Han, Qiliang Yang, Jianchun Xing, Juelong Li, and Hongda Wang. "FAME: A UML-based framework for modeling fuzzy self-adaptive software." In: *Information and Software Technology* 76.C (2016), pp. 118–134. DOI: 10.1016/j.infsof.2016.04.014.

[205] David Harel and Assaf Marron. "The quest for runware: on compositional, executable and intuitive models." In: *Software and Systems Modeling* 11.4 (2012), pp. 599–608. DOI: 10.1007/s10270-012-0258-8.

[206] David Harel and Amnon Naamad. "The STATEMATE Semantics of Statecharts." In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (1996), pp. 293–333. DOI: 10.1145/235321.235322.

[207] David Harel and Bernhard Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *IEEE Computer* 37.10 (2004), pp. 64–72. DOI: 10.1109/MC.2004.172.

[208] Wilhelm Hasselbring, Robert Heinrich, Reiner Jung, Andreas Metzger, Klaus Pohl, Ralf Reussner, and Eric Schmieders. *iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems*. Tech. rep. 1309. Institut für Informatik, Universität Kiel, Deutschland, 2013. URL: http://www.uni-kiel.de/journals/receive/jportal_jparticle_00000031.

[209] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. "A Case Study in Goal-Driven Architectural Adaptation." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 109–127. DOI: 10.1007/978-3-642-02161-9_6.

[210] Regina Hebig, Holger Giese, and Basil Becker. "Making Control Loops Explicit when Architecting Self-adaptive Systems." In: *International Workshop on Self-organizing Architectures*. SOAR '10. ACM, 2010, pp. 21–28. DOI: 10.1145/1809036.1809042.

[211] Regina Hebig, Andreas Seibel, and Holger Giese. "On the Unification of Megamodels." In: *International Workshop on Multi-Paradigm Modeling (MPM '10)*. Ed. by Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Laszlo Lengyel, Tiziana Magaria, Julia Padberg, and Gabriele Taentzer. Vol. 42. Electronic Communications of the EASST. 2011. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/704/713.

[212] Jan Heering and Paul Klint. "Semantics of Programming Languages: A Tool-oriented Approach." In: *SIGPLAN Not.* 35.3 (2000), pp. 39–48. DOI: 10.1145/351159.351173.

[213] Christian Hein, Tom Ritter, and Michael Wagner. "System Monitoring using Constraint Checking as part of Model Based System Management." In: *International Workshop on Models@run.time*. 2007. URL: https://st.inf.tu-dresden.de/MRT07/accepted.html.

[214] Robert Heinrich, Eric Schmieders, Reiner Jung, Kiana Rostami, Andreas Metzger, Willhelm Hasselbring, Ralf Reussner, and Klaus Pohl. "Integrating Run-time Observations and Design Component Models for Cloud System Analysis." In: *International Workshop on Models@run.time*. Vol. 1270. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 41–46. URL: http://ceur-ws.org/Vol-1270/.

[215] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN: 0471266372.

[216] Brian Henderson-Sellers. "UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts." In: *Software and Systems Modeling* 4.1 (2005), pp. 4–13. DOI: 10.1007/s10270-004-0076-8.

[217] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. "The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review." In: *ACM Comput. Surv.* 46.2 (2013), 28:1–28:28. DOI: 10.1145/2543581.2543595.

[218] Thorsten Hestermeyer, Oliver Oberschelp, and Holger Giese. "Structured Information Processing For Self-optimizing Mechatronic Systems." In: *International Conference on Informatics in Control, Automation and Robotics*. Ed. by Helder Araujo, Alves Vieira, Jose Braz, Bruno Encarnacao, and Marina Carvalho. ICINCO '04. INSTICC Press, 2004, pp. 230–237.

[219] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. "Design Science in Information Systems Research." In: *MIS Quarterly* 28.1 (2004), pp. 75–105. URL: http://www.jstor.org/stable/25148625.

[220] Stephan Hildebrandt, Leen Lambers, Giese Holger, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. "A Survey of Triple Graph Grammar Tools." In: *International Workshop on Bidirectional Transformations*. Vol. 57. BX '13. EC-EASST, 2013, pp. 1–18. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/865/858.

[221] Michael G. Hinchey and Roy Sterritt. "Self-Managing Software." In: *IEEE Computer* 39.2 (2006), pp. 107–109. DOI: 10.1109/MC.2006.69.

[222] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. *SEEC: A Framework for Self-aware Computing*. Tech. rep. MIT-CSAIL-TR-2010-049. MIT CSAIL Technical Report, 2010. URL: http://hdl.handle.net/1721.1/59519.

[223] Edzard Höfig, Peter H. Deussen, and Hakan Coskun. "Statechart Interpretation on Resource Constrained Platforms: a Performance Analysis." In: *International Workshop on Models@run.time*. Vol. 509. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 99–108. URL: http://ceur-ws.org/Vol-509/.

[224] Gerard J. Holzmann. "Code Inflation." In: *IEEE Software* 32.2 (2015), pp. 10–13. DOI: 10.1109/MS.2015.40.

[225] Paul Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. 2001.

[226] Hans Hrasna. *Java™ 2 Platform, Enterprise Edition Management Specification*. JSR-77, Maintenance Release v1.1. 2006. URL: https://jcp.org/en/jsr/detail?id=77.

[227] Gang Huang, Hong Mei, and Qian-xiang Wang. "Towards Software Architecture at Runtime." In: *SIGSOFT Softw. Eng. Notes* 28.2 (2003), pp. 8–13. DOI: 10.1145/638750.638780.

[228] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bahr. "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language." In: *IEEE Transactions on Software Engineering* 43.5 (2017), pp. 432–452. DOI: 10.1109/TSE.2016.2613863.

[229] Nikolaus Huber, André van Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments." In: *Service Oriented Computing and Applications* 8.1 (2014), pp. 73–89. DOI: 10.1007/s11761-013-0144-4.

[230] Markus C. Huebscher and Julie A. McCann. "A Survey of Autonomic Computing—Degrees, Models, and Applications." In: *ACM Comput. Surv.* 40.3 (2008), 7:1–7:28. DOI: 10.1145/1380584.1380585.

[231] M. Usman Iftikhar, Jonas Lundberg, and Danny Weyns. "A Model Interpreter for Timed Automata." In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA '16)*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9952. Lecture Notes in Computer Science (LNCS). Springer, 2016, pp. 243–258. DOI: 10.1007/978-3-319-47166-2_17.

[232] M. Usman Iftikhar and Danny Weyns. "ActivFORMS: Active Formal Models for Self-adaptation." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '14. ACM, 2014, pp. 125–134. DOI: 10.1145/2593929.2593944.

[233] Didac Gil De La Iglesia and Danny Weyns. "MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems." In: *ACM Trans. Auton. Adapt. Syst.* 10.3 (2015), 15:1–15:31. DOI: 10.1145/2724719.

[234] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. "A Perspective on the Future of Middleware-based Software Engineering." In: *Future of Software Engineering*. FOSE '07. IEEE, 2007, pp. 244–258. DOI: 10.1109/FOSE.2007.2.

[235] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K. Nadgir, and Amr F. Yassin. *A Practical Guide to the IBM Autonomic Computing Toolkit*. 1st ed. IBM Corporation Redbooks, 2004. URL: http://www.redbooks.ibm.com/abstracts/sg246635.html.

[236] Thomas Johanndeiter, Anat Goldstein, and Ulrich Frank. "Towards Business Process Models at Runtime." In: *International Workshop on Models@run.time*. Vol. 1079. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 13–25. URL: http://ceur-ws.org/Vol-1079/.

[237] T. C. Jones. "Measuring Programming Quality and Productivity." In: *IBM Syst. J.* 17.1 (1978), pp. 39–63. DOI: 10.1147/sj.171.0039.

[238] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. "ATL: A Model Transformation Tool." In: *Sci. Comput. Program.* 72.1–2 (2008), pp. 31–39. DOI: 10.1016/j.scico.2007.08.002.

[239] Frédéric Jouault, Jean Bézivin, Regis Chevrel, and Jeff Gray. "Experiments in Run-Time Model Extraction." In: *International Workshop on Models@run.time*. 2006. URL: https://st.inf.tu-dresden.de/MRT06/accepted.html.

[240] Elsy Kaddoum, Claudia Raibulet, Jean-Pierre Georgé, Gauthier Picard, and Marie-Pierre Gleizes. "Criteria for the Evaluation of Self-* Systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. ACM, 2010, pp. 29–38. DOI: 10.1145/1808984.1808988.

[241] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Volkel. "Design Guidelines for Domain Specific Languages." In: *OOPSLA Workshop on Domain-Specific Modeling*. Ed. by Matti Rossi, Jonathan Sprinkle, Jeff Gray, and Juha-Pekka Tolvanen. DSM '09. Helsinki School of Economics. TR no B-108, 2009, pp. 7–13. URL: http://urn.fi/URN:ISBN:978-952-488-372-6.

[242] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008. ISBN: 0470036662. URL: http://dsmbook.com/.

[243] Stuart Kent. "Model Driven Engineering." In: *Integrated Formal Methods*. Ed. by Michael Butler, Luigia Petre, and Kaisa Sere. Vol. 2335. Lecture Notes in Computer Science (LNCS). Springer, 2002, pp. 286–298. DOI: 10.1007/3-540-47884-1_16.

[244] Jeffrey O. Kephart. "Research Challenges of Autonomic Computing." In: *International Conference on Software Engineering*. ICSE '05. ACM, 2005, pp. 15–22. DOI: 10.1145/1062455.1062464.

[245] Jeffrey O. Kephart, Hoi Chan, Rajarshi Das, David W. Levine, Gerald Tesauro, Freeman Rawson, and Charles Lefurgy. "Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs." In: *International Conference on Autonomic Computing*. ICAC '07. IEEE, 2007, pp. 24–33. DOI: 10.1109/ICAC.2007.12.

[246] Jeffrey O. Kephart and David Chess. "The Vision of Autonomic Computing." In: *IEEE Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[247] Jeffrey O. Kephart and William E. Walsh. "An Artificial Intelligence Perspective on Autonomic Computing Policies." In: *International Workshop on Policies for Distributed Systems and Networks*. POLICY '04. IEEE, 2004, pp. 3–12. DOI: 10.1109/POLICY.2004.1309145.

[248] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN: 0262610744.

[249] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. "Towards an ontology of software maintenance." In: *Journal of Software Maintenance: Research and Practice* 11.6 (1999), pp. 365–389. DOI: 10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W.

[250] Rob Kitchin and Martin Dodge. *Code/Space: Software and Everyday Life*. MIT Press, 2011. ISBN: 0262042482.

[251] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar. "Control Theory-Based Foundations of Self-Controlling Software." In: *IEEE Intelligent Systems* 14.3 (1999), pp. 37–45. DOI: 10.1109/5254.769883.

[252] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. "The Case for Reflective Middleware." In: *Commun. ACM* 45.6 (2002), pp. 33–38. DOI: 10.1145/508448.508470.

[253] Hermann Kopetz. "Event-Triggered Versus Time-Triggered Real-Time Systems." In: *International Workshop on Operating Systems of the 90s and Beyond*. Ed. by Arthur Karshmer and Jürgen Nehmer. Vol. 563. Lecture Notes in Computer Science (LNCS). Springer, 1991, pp. 86–101. DOI: 10.1007/BFb0024530.

[254] Samuel Kounev, Nikolaus Huber, Fabian Brosig, Xiaoyun Zhu, undefined, undefined, undefined, and undefined. "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures." In: *IEEE Computer* 49.7 (2016), pp. 53–61. DOI: 10.1109/MC.2016.198.

[255] Samuel Kounev, Xiaoyun Zhu, Jeffrey O. Kephart, and Marta Kwiatkowska. "Model-driven Algorithms and Architectures for Self-Aware Computing Systems." In: *Dagstuhl Reports* 5.1 (2015), pp. 164–196. DOI: 10.4230/DagRep.5.1.164.

[256] Jeff Kramer. "Adventures in Adaptation: a software engineering playground!" In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. Keynote. IEEE, 2015.

[257] Jeff Kramer. "Is Abstraction the Key to Computing?" In: *Commun. ACM* 50.4 (2007), pp. 36–42. DOI: 10.1145/1232743.1232745.

[258] Jeff Kramer and Jeff Magee. "Self-Managed Systems: An Architectural Challenge." In: *Future of Software Engineering*. FOSE '07. IEEE, 2007, pp. 259–268. DOI: 10.1109/FOSE.2007.19.

[259] Jeff Kramer and Jeff Magee. "The Evolving Philosophers Problem: Dynamic Change Management." In: *IEEE Transactions on Software Engineering* 16.11 (1990), pp. 1293–1306. DOI: 10.1109/32.60317.

[260] Filip Krikava. "Domain-specific modeling language for self-adaptive software system architectures." PhD thesis. Université Nice Sophia Antipolis, 2013. URL: https://tel.archives-ouvertes.fr/tel-00935083.

[261] Filip Krikava, Philippe Collet, and Robert B. France. "Actor-based Runtime Model of Adaptable Feedback Control Loops." In: *International Workshop on Models@run.time*. ACM, 2012, pp. 39–44. DOI: 10.1145/2422518.2422525.

[262] Filip Krikava, Philippe Collet, and Robert B. France. "ACTRESS: Domain-specific Modeling of Self-adaptive Software Architectures." In: *Symposium on Applied Computing*. SAC '14. ACM, 2014, pp. 391–398. DOI: 10.1145/2554850.2555020.

[263] Christian Krupitzer, Felix Maximilian Roth, Christian Becker, Markus Weckesser, Malte Lochau, and Andy Schürr. "FESAS IDE: An Integrated Development Environment for Autonomic Computing." In: *International Conference on Autonomic Computing*. ICAC '16. IEEE, 2016, pp. 15–24. DOI: 10.1109/ICAC.2016.49.

[264] Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, and Christian Becker. "Towards Reusability in Autonomic Computing." In: *International Conference on Autonomic Computing*. ICAC '15. IEEE, 2015, pp. 115–120. DOI: 10.1109/ICAC.2015.21.

[265] Adrian Kuhn and Toon Verwaest. "FAME - A Polyglot Library for Metamodeling at Runtime." In: *International Workshop on Models@run.time*. Technical Report COMP COMP-005-2008, Lancaster University, 2008, pp. 57–66. URL: http://eprints.lancs.ac.uk/id/eprint/42322.

[266] Robert Laddaga. "Creating Robust Software through Self-Adaptation." In: *IEEE Intelligent Systems* 14.3 (1999), pp. 26–29. DOI: 10.1109/MIS.1999.769879.

[267] Leen Lambers, Stephan Hildebrandt, Holger Giese, and Fernando Orejas. "Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case." In: *International Workshop on Bidirectional Transformations*. Ed. by Frank Hermann and Janis Voigtländer. Vol. 49. BX '12. EC-EASST, 2012, pp. 1–16. DOI: 10.14279/tuj.eceasst.49.706.714.

[268] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. "EMF Profiles: A Lightweight Extension Approach for EMF Models." In: *Journal of Object Technology* 11.1 (2012), 8:1–29. DOI: 10.5381/jot.2012.11.1.a8.

[269] Craig Larman and Victor R. Basili. "Iterative and Incremental Development: A Brief History." In: *IEEE Computer* 36.6 (2003), pp. 47–56. DOI: 10.1109/MC.2003.1204375.

[270] Meir M. Lehman. "Laws of software evolution revisited." In: *Software Process Technology*. Ed. by Carlo Montangero. Vol. 1149. Lecture Notes in Computer Science (LNCS). Springer, 1996, pp. 108–124. DOI: 10.1007/BFb0017737.

[271] Meir M. Lehman. "Programs, life cycles, and laws of software evolution." In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[272] Meir M. Lehman. "Software's Future: Managing Evolution." In: *IEEE Software* 15.1 (1998), pp. 40–44. DOI: 10.1109/MS.1998.646878.

[273] Meir M. Lehman and Laszlo A. Belady, eds. *Program evolution: processes of software change*. San Diego, CA, USA: Academic Press Professional, Inc., 1985. ISBN: 0124424406.

[274] Meir M. Lehman and Juan F. Ramil. "Rules and Tools for Software Evolution Planning and Management." In: *Ann. Softw. Eng.* 11.1 (2001), pp. 15–44. DOI: 10.1023/A:1012535017876.

[275] Meir M. Lehman, Juan F. Ramil, Paul D. Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. "Metrics and Laws of Software Evolution - The Nineties View." In: *International Symposium on Software Metrics*. METRICS '97. IEEE, 1997, pp. 20–32. DOI: 10.1109/METRIC.1997.637156.

[276] Rogerio de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. "Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511)." In: *Dagstuhl Reports* 3.12 (2014). Ed. by Rogerio de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, pp. 67–96. DOI: 10.4230/DagRep.3.12.67.

[277] Peter R. Lewis, Arjun Chandra, Funmilade Faniyi, Kyrre Glette, Tao Chen, Rami Bahsoon, Jim Torresen, and Xin Yao. "Architectural Aspects of Self-Aware and Self-Expressive Computing Systems: From Psychology to Engineering." In: *IEEE Computer* 48.8 (2015), pp. 62–70. DOI: 10.1109/MC.2015.235.

[278] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification, Java SE 8 Edition*. JSR-337. 2015. URL: https://jcp.org/en/jsr/detail?id=337.

[279] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. "A Policy Description Language." In: *National Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference*. AAAI '99/IAAI '99. American Association for Artificial Intelligence, 1999, pp. 291–298. URL: http://www.aaai.org/Library/AAAI/1999/aaai99-043.php.

[280] Markus Luckey and Gregor Engels. "High-quality Specification of Self-adaptive Software Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. IEEE, 2013, pp. 143–152. DOI: 10.1109/SEAMS.2013.6595501.

[281] Markus Luckey, Benjamin Nagel, Christian Gerth, and Gregor Engels. "Adapt Cases: Extending Use Cases for Adaptive Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, 2011, pp. 30–39. DOI: 10.1145/1988008.1988014.

[282] Jochen Ludewig. "Models in software engineering – an introduction." In: *Software and Systems Modeling* 2.1 (2003), pp. 5–14. DOI: 10.1007/s10270-003-0020-3.

[283] Maksym Lushpenko, Nicolas Ferry, Hui Song, Franck Chauvel, and Arnor Solberg. "Using Adaptation Plans to Control the Behavior of Models@runtime." In: *International Workshop on Models@run.time*. Vol. 1474. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 11–20. URL: http://ceur-ws.org/Vol-1474/.

[284] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. "Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems." In: *SIGSOFT Symposium and European Conference on Foundations of Software Engineering*. ESEC/FSE '11. ACM, 2011, pp. 245–255. DOI: 10.1145/2025113.2025148.

[285] Pattie Maes. "Concepts and Experiments in Computational Reflection." In: *International Conference on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '87. ACM, 1987, pp. 147–155. DOI: 10.1145/38765.38821.

[286] Jeff Magee and Jeff Kramer. "Dynamic Structure in Software Architectures." In: *Symposium on Foundations of Software Engineering*. SIGSOFT '96. ACM, 1996, pp. 3–14. DOI: 10.1145/239098.239104.

[287] Edson Manoel, Morten Jul Nielsen, Abdi Salahshour, Sai Sampath K.V.L., and Sanjeev Sudarshanan. *Problem Determination Using Self-Managing Autonomic Technology*. 1st ed. IBM Corporation Redbooks, 2005. URL: http://www.redbooks.ibm.com/abstracts/sg246665.html.

[288] Shahar Maoz. "Using Model-Based Traces as Runtime Models." In: *IEEE Computer* 42.10 (2009), pp. 28–36. DOI: 10.1109/MC.2009.336.

[289] Jim A. McCall, Paul K. Richards, and Gene F. Walters. *Factors in Software Quality: Concepts and Definitions of Software Quality*. Tech. rep. RADC-TR-77-369. Rome Air Development Center, 1977. URL: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA049014.

[290] Julie A. McCann and Markus C. Huebscher. "Evaluation Issues in Autonomic Computing." In: *International Conference on Grid and Cooperative Computing - GCC 2004 Workshops*. Ed. by Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun. Vol. 3252. Lecture Notes in Computer Science (LNCS). Springer, 2004, pp. 597–608. DOI: 10.1007/978-3-540-30207-0_74.

[291] Dennis McCarthy and Umeshwar Dayal. "The Architecture of an Active Database Management System." In: *International Conference on Management of Data*. SIGMOD '89. ACM, 1989, pp. 215–224. DOI: 10.1145/67544.66946.

[292] Philip McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. Cheng. "Composing Adaptive Software." In: *IEEE Computer* 37.7 (2004), pp. 56–64. DOI: 10.1109/MC.2004.48.

[293] Nenad Medvidovic and Richard N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages." In: *IEEE Transactions on Software Engineering* 26.1 (2000), pp. 70–93. DOI: 10.1109/32.825767.

[294] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. "Model-driven development - Guest editor's introduction." In: *IEEE Software* 20.5 (2003), pp. 14–18. DOI: 10.1109/MS.2003.1231145.

[295] Daniel Menasce, Hassan Gomaa, sam Malek, and Joao Sousa. "SASSY: A Framework for Self-Architecting Service-Oriented Systems." In: *IEEE Software* 28.6 (Nov. 2011), pp. 78–85. ISSN: 0740-7459. DOI: 10.1109/MS.2011.22.

[296] Josh Mengerink, Ramon R.H. Schiffelers, Alexander Serebrenik, and Mark van den Brand. "DSL/Model Co-Evolution in Industrial EMF-Based MDSE Ecosystems." In: *Workshop on Models and Evolution*. Vol. 1706. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 2–7. URL: http://ceur-ws.org/Vol-1706/.

[297] Tom Mens and Serge Demeyer, eds. *Software Evolution*. Springer, 2008. DOI: 10.1007/978-3-540-76440-3.

[298] Tom Mens, Yann-Gael Gueheneuc, Juan Fernandez-Ramil, and Maja D'Hondt. "Guest Editors' Introduction: Software Evolution." In: *IEEE Software* 27.4 (2010), pp. 22–25. DOI: 10.1109/MS.2010.100.

[299] Tom Mens, Alexander Serebrenik, and Anthony Cleve, eds. *Evolving Software Systems*. Springer, 2014. DOI: 10.1007/978-3-642-45398-4.

[300] Tom Mens and Pieter Van Gorp. "A Taxonomy of Model Transformation." In: *Electron. Notes Theor. Comput. Sci.* 152 (2006), pp. 125–142. DOI: 10.1016/j.entcs.2005.10.021.

[301] Tom Mens, Michel Wermelinger, Stephane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. "Challenges in Software Evolution." In: *International Workshop on Principles of Software Evolution*. IEEE, 2005, pp. 13–22. DOI: 10.1109/IWPSE.2005.7.

[302] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages." In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.

[303] Thomas O. Meservy and Kurt D. Fenstermacher. "Transforming Software Development: An MDA Road Map." In: *IEEE Computer* 38.9 (2005), pp. 52–58. DOI: 10.1109/MC.2005.316.

[304] Daniel Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering." In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779. DOI: 10.1109/TSE.2009.67.

[305] Brice Morin. "Leveraging Models from Design-time to Runtime to Support Dynamic Variability." PhD thesis. University of Rennes 1, France, 2010. URL: https://hal.inria.fr/tel-00538548/en.

[306] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. "Models@ Run.time to Support Dynamic Adaptation." In: *IEEE Computer* 42.10 (2009), pp. 44–51. DOI: 10.1109/MC.2009.327.

[307] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. "Taming Dynamically Adaptive Systems Using Models and Aspects." In: *International Conference on Software Engineering*. ICSE '09. IEEE, 2009, pp. 122–132. DOI: 10.1109/ICSE.2009.5070514.

[308] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. "An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability." In: *International Conference on Model Driven Engineering Languages and Systems (MoDELS '08)*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Vol. 5301. Lecture Notes in Computer Science (LNCS). Springer, 2008, pp. 782–796. DOI: 10.1007/978-3-540-87875-9_54.

[309] Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais, and Jean-Marc Jézéquel. "Unifying Runtime Adaptation and Design Evolution." In: *International Conference on Computer and Information Technology*. CIT '09. IEEE, 2009, pp. 104–109. DOI: 10.1109/CIT.2009.94.

[310] Hausi A. Müller, Holger M. Kienle, and Ulrike Stege. *Autonomic Computing Now You See It, Now You Don't*. Ed. by Andrea De Lucia and Filomena Ferrucci. 2009. DOI: 10.1007/978-3-540-95888-8_2.

[311] Hausi Müller, Mauro Pezzè, and Mary Shaw. "Visibility of Control in Adaptive Systems." In: *International Workshop on Ultra-large-scale Software-intensive Systems*. ULSSIS '08. ACM, 2008, pp. 23–26. DOI: 10.1145/1370700.1370707.

[312] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H.C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. "The Relevance of Model-Driven Engineering Thirty Years from Now." In: *Model-Driven Engineering Languages and Systems (MoDELS '14)*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahao, and Emilio Insfran. Vol. 8767. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 183–200. DOI: 10.1007/978-3-319-11653-2_12.

[313] Florian Nafz, Frank Ortmeier, Hella Seebach, Jan-Philipp Steghofer, and Wolfgang Reif. "A generic software framework for role-based Organic Computing systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 96–105. DOI: 10.1109/SEAMS.2009.5069078.

[314] Benjamin Nagel, Christian Gerth, Enes Yigitbas, Fabian Christ, and Gregor Engels. "Model-driven Specification of Adaptive Cloud-based Systems." In: *International Workshop on Model-Driven Engineering for High Performance and CLoud Computing*. MDHPCL '12. ACM, 2012, 4:1–4:6. DOI: 10.1145/2446224.2446228.

[315] Leandro Nahabedian, Víctor Braberman, Nicolas D'Ippolito, Shinichi Honiden, Jeff Kramer, Kenji Tei, and Sebastian Uchitel. "Assured and Correct Dynamic Update of Controllers." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '16. ACM, 2016, pp. 96–107. DOI: 10.1145/2897053.2897056.

[316] Hiroyuki Nakagawa, Akihiko Ohsuga, and Shinichi Honiden. "Towards Dynamic Evolution of Self-Adaptive Systems Based on Dynamic Updating of Control Loops." In: *International Conference on Self-Adaptive and Self-Organizing Systems*. SASO '12. IEEE, 2012, pp. 59–68. DOI: 10.1109/SASO.2012.17.

[317] Sangeeta Neti and Hausi A. Muller. "Quality Criteria and an Analysis Framework for Self-Healing Systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '07. IEEE, 2007, 6:1–6:10. DOI: 10.1109/SEAMS.2007.15.

[318] Viet Hoa Nguyen, Francois Fouquet, Noël Plouzeau, and Olivier Barais. "A Process for Continuous Validation of Self-adapting Component Based Systems." In: *International Workshop on Models@run.time*. ACM, 2012, pp. 32–37. DOI: 10.1145/2422518.2422524.

[319] Oscar Nierstrasz, Marcus Denker, and Lukas Renggli. "Model-Centric, Context-Aware Software Adaptation." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 128–145. DOI: 10.1007/978-3-642-02161-9_7.

[320] Oliver Niggemann, Anne Geburzi, and Joachim Stroop. "Benefits of System Simulation for Automotive Applications." In: *Model-Based Engineering of Embedded Real-Time Systems*. Ed. by Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz. Vol. 6100. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 329–336. DOI: 10.1007/978-3-642-16277-0_15.

[321] Linda Northrop, Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

[322] Object Management Group. *Common Warehouse Metamodel (CWM) Specification*. Version 1.1, Volume 1, formal/03-03-02. 2003. URL: http://www.omg.org/spec/CWM/1.1/.

[323] Object Management Group. *Diagram Definition (DD)*. Version 1.1, formal/2015-06-01. 2015. URL: http://www.omg.org/spec/DD/1.1/.

[324] Object Management Group. *MDA Guide rev. 2.0*. Version 2.0, OMG Document ormsc/2014-06-01. 2014. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf.

[325] Object Management Group. *MDA Guide Version 1.0.1*. Version 1.0.1, omg/2003-06-01. 2003. URL: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

[326] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Version 1.2, formal/15-02-01. 2014. URL: http://www.omg.org/spec/QVT/1.2/.

[327] Object Management Group. *Object Constraint Language*. Version 2.4, formal/2014-02-03. 2014. URL: http://www.omg.org/spec/OCL/2.4.

[328] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification*. Version 2.5, formal/2015-06-05. 2015. URL: http://www.omg.org/spec/MOF/2.5.

[329] Object Management Group. *OMG Unified Modeling Language™ (OMG UML)*. Version 2.5, formal/2015-03-01. 2015. URL: http://www.omg.org/spec/UML/2.5.

[330] Object Management Group. *Software & Systems Process Engineering Meta-Model Specification*. Version 2.0, formal/2008-04-01. 2008. URL: http://www.omg.org/spec/SPEM/2.0/.

[331] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework." In: *International Workshop on Reflection and Meta-level Architecture*. 1992, pp. 36–47.

[332] Raphael Pereira de Oliveira and Eduardo Santana de Almeida. "Evaluating Lehman's Laws of Software Evolution for Software Product Lines." In: *IEEE Software* 33.3 (2016), pp. 90–93. DOI: 10.1109/MS.2016.78.

[333] Raphael Pereira de Oliveira, Eduardo Santana de Almeida, and Gecynalda Soares da Silva Gomes. "Evaluating Lehman's Laws of Software Evolution within Software Product Lines: A Preliminary Empirical Study." In: *International Conference on Software Reuse (ICSR '15)*. Ed. by Ina Schaefer and Ioannis Stamelos. Vol. 8919. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 42–57. DOI: 10.1007/978-3-319-14130-5_4.

[334] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. "An Architecture-Based Approach to Self-Adaptive Software." In: *IEEE Intelligent Systems* 14.3 (1999), pp. 54–62. DOI: 10.1109/5254.769885.

[335] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. "Architecture-based Runtime Software Evolution." In: *International Conference on Software Engineering*. ICSE '98. IEEE, 1998, pp. 177–186. DOI: 10.1109/ICSE.1998.671114.

[336] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. "Runtime Software Adaptation: Framework, Approaches, and Styles." In: *Companion of the International Conference on Software Engineering*. ICSE Companion '08. ACM, 2008, pp. 899–910. DOI: 10.1145/1370175.1370181.

[337] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. "Metamodelling for Grammarware Researchers." In: *Software Language Engineering*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 64–82. DOI: 10.1007/978-3-642-36089-3_5.

[338] Valerio Panzica La Manna, Joel Greenyer, Carlo Ghezzi, and Christian Brenner. "Formalizing Correctness Criteria of Dynamic Updates Derived from Specification Changes." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. IEEE, 2013, pp. 63–72. DOI: 10.1109/SEAMS.2013.6595493.

[339] David Lorge Parnas. "Software Aging." In: *International Conference on Software Engineering*. ICSE '94. IEEE, 1994, pp. 279–287. DOI: 10.1109/ICSE.1994.296790.

[340] *Participants' documents and presentations from the Dagstuhl Seminar on Models@run.time*. 2011. URL: http://www.dagstuhl.de/mat/index.en.phtml?11481.

[341] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. "Self-adaptation of mobile systems driven by the Common Variability Language." In: *Future Generation Computer Systems* 47 (2015). Special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems, pp. 127–144. DOI: 10.1016/j.future.2014.08.015.

[342] Liliana Pasquale, Luciano Baresi, and Bashar Nuseibeh. "Towards Adaptive Systems through Requirements@Runtime." In: *International Workshop on Models@run.time*. Vol. 794. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 13–24. URL: http://ceur-ws.org/Vol-794/.

[343] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. "A Systematic Survey on the Design of Self-adaptive Software Systems Using Control Engineering Approaches." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE, 2012, pp. 33–42. DOI: 10.1109/SEAMS.2012.6224389.

[344] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. "A Design Science Research Methodology for Information Systems Research." In: *J. Manage. Inf. Syst.* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

[345] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. "Context Aware Computing for The Internet of Things: A Survey." In: *IEEE Communications Surveys & Tutorials* 16.1 (2014), pp. 414–454. DOI: 10.1109/SURV.2013.042313.00197.

[346] Daniel Perovich, MaríaCecilia Bastarrica, and Cristian Rojas. "Model-Driven approach to Software Architecture design." In: *International Workshop on Sharing and Reusing Architectural Knowledge*. SHARK '09. IEEE, 2009, pp. 1–8.

[347] Gilles Perrouin, Brice Morin, Franck Chauvel, Franck Fleurey, Jacques Klein, Yves Le Traon, Olivier Barais, and Jean-Marc Jézéquel. "Towards Flexible Evolution of Dynamically Adaptive Systems." In: *International Conference on Software Engineering*. ICSE '12. IEEE, 2012, pp. 1353–1356. DOI: 10.1109/ICSE.2012.6227081.

[348] Mauro Pezzè. "From off-Line to continuous on-line maintenance." In: *International Conference on Software Maintenance*. ICSM '12. IEEE, 2012, pp. 2–3. DOI: 10.1109/ICSM.2012.6405244.

[349] Goran Piskachev. *LAFORE: A Domain Specific Language for Reconfiguration*. Universität Paderborn, Germany. Master's thesis. 2015.

[350] Harald Psaier and Schahram Dustdar. "A survey on self-healing systems: approaches and systems." In: *Computing* 91.1 (2011), pp. 43–73. DOI: 10.1007/s00607-010-0107-y.

[351] Nauman A. Qureshi and Anna Perini. "Engineering adaptive requirements." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 126–131. DOI: 10.1109/SEAMS.2009.5069081.

[352] Nauman A. Qureshi and Anna Perini. "Requirements Engineering for Adaptive Service Based Applications." In: *International Requirements Engineering Conference*. RE '10. IEEE, 2010, pp. 108–111. DOI: 10.1109/RE.2010.23.

[353] Václav Rajlich. "Software Evolution and Maintenance." In: *Future of Software Engineering*. FOSE '14. ACM, 2014, pp. 133–144. DOI: 10.1145/2593882.2593893.

[354] Vaclav T. Rajlich and Keith H. Bennett. "A Staged Model for the Software Life Cycle." In: *IEEE Computer* 33.7 (2000), pp. 66–71. DOI: 10.1109/2.869374.

[355] Andres J. Ramirez and Betty H. C. Cheng. "Design Patterns for Developing Dynamically Adaptive Systems." In: *Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. ACM, 2010, pp. 49–58. DOI: 10.1145/1808984.1808990.

[356] Andres J. Ramirez, Betty H.C. Cheng, and Philip K. McKinley. "Adaptive Monitoring of Software Requirements." In: *International Workshop on Requirements@Run.Time*. RE@RunTime '10. 2010, pp. 41–50. DOI: 10.1109/RERUNTIME.2010.5628549.

[357] Andres J. Ramirez, Betty H.C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. "Automatically Generating Adaptive Logic to Balance Non-functional Tradeoffs During Reconfiguration." In: *International Conference on Autonomic Computing*. ICAC '10. ACM, 2010, pp. 225–234. DOI: 10.1145/1809049.1809080.

[358] Andres J. Ramirez, Adam C. Jensen, and Betty H. C. Cheng. "A Taxonomy of Uncertainty for Dynamically Adaptive Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE, 2012, pp. 99–108. DOI: 10.1109/SEAMS.2012.6224396.

[359] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. "Applying Genetic Algorithms to Decision Making in Autonomic Computing Systems." In: *International Conference on Autonomic Computing*. ICAC '09. ACM, 2009, pp. 97–106. DOI: 10.1145/1555228.1555258.

[360] Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin, eds. *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Springer, 2013. DOI: 10.1007/978-3-642-36654-3.

[361] Jorge Ressia, Lukas Renggli, Tudor Girba, and Oscar Nierstrasz. "Run-Time Evolution through Explicit Meta-Objects." In: *International Workshop on Models@run.time*. Vol. 641. CEUR Workshop Proceedings. CEUR-WS.org, 2010, pp. 37–48. URL: http://ceur-ws.org/Vol-641/.

[362] Jim des Rivières and Brian Cantwell Smith. "The Implementation of Procedurally Reflective Languages." In: *Symposium on LISP and Functional Programming*. LFP '84. ACM, 1984, pp. 331–347. DOI: 10.1145/800055.802050.

[363] Liliana Rosa, Luis Rodrigues, Antonia Lopes, Matti Hiltunen, and Richard Schlichting. "Self-Management of Adaptable Component-Based Applications." In: *IEEE Transactions on Software Engineering* 39.3 (2013), pp. 403–421. DOI: 10.1109/TSE.2012.29.

[364] Jarrett Rosenberg. "Some Misconceptions About Lines of Code." In: *4th International Symposium on Software Metrics*. METRICS '97. IEEE, 1997, pp. 137–142. DOI: 10.1109/METRIC.1997.637174.

[365] Felix Maximilian Roth, Christian Krupitzer, and Christian Becker. "Runtime Evolution of the Adaptation Logic in Self-Adaptive Systems." In: *International Conference on Autonomic Computing*. ICAC '15. IEEE, 2015, pp. 141–142. DOI: 10.1109/ICAC.2015.20.

[366] Jeff Rothenberg. "The Nature of Modeling." In: *Artificial Intelligence, Simulation & Modeling*. Ed. by Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. John Wiley & Sons, Inc., 1989, pp. 75–92. URL: http://dl.acm.org/citation.cfm?id=73119.73122.

[367] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 164–182. DOI: 10.1007/978-3-642-02161-9_9.

[368] Rick Salay, John Mylopoulos, and Steve Easterbrook. "Using Macromodels to Manage Collections of Related Models." In: *Advanced Information Systems Engineering*. Ed. by Pascal van Eck, Jaap Gordijn, and Roel Wieringa. Vol. 5565. Lecture Notes in Computer Science (LNCS). Springer, 2009, pp. 141–155. DOI: 10.1007/978-3-642-02144-2_15.

[369] Mazeiar Salehie and Ladan Tahvildari. "A Weighted Voting Mechanism for Action Selection Problem in Self-Adaptive Software." In: *International Conference on Self-Adaptive and Self-Organizing Systems*. SASO '07. IEEE, 2007, pp. 328–331. DOI: 10.1109/SASO.2007.4.

[370] Mazeiar Salehie and Ladan Tahvildari. "Self-adaptive software: Landscape and research challenges." In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (2009), pp. 1–42. DOI: 10.1145/1516533.1516538.

[371] Mario Sanchez, Ivan Barrero, Jorge Villalobos, and Dirk Deridder. "An Execution Platform for Extensible Runtime Models." In: *International Workshop on Models@run.time*. Technical Report COMP COMP-005-2008, Lancaster University, 2008, pp. 107–116. URL: http://eprints.lancs.ac.uk/id/eprint/42322.

[372] Mahadev Satyanarayanan. "Pervasive Computing: Vision and Challenges." In: *IEEE Personal Communications* 8.4 (2001), pp. 10–17. DOI: 10.1109/98.943998.

[373] Bernhard Schätz and Katharina Spies. "Model-based software engineering: Linking more and less formal product-models to a structured process." In: *International Conference Software and Systems Engineering and their Applications*. ICSSEA '02. 2002. URL: http://www4.in.tum.de/~schaetz/papers/SchaetzSpies_ICSSEA_02.pdf.

[374] Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. "Adaptivity and Self-organization in Organic Computing Systems." In: *ACM Trans. Auton. Adapt. Syst.* 5.3 (2010), 10:1–10:32. DOI: 10.1145/1837909.1837911.

[375] Douglas C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering." In: *IEEE Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

[376] Daniel Schneider and Martin Becker. "Runtime Models for Self-Adaptation in the Ambient Assisted Living Domain." In: *International Workshop on Models@run.time*. Technical Report COMP COMP-005-2008, Lancaster University, 2008, pp. 47–56. URL: http://eprints.lancs.ac.uk/id/eprint/42322.

[377] Andy Schürr. "Specification of graph translators with triple graph grammars." In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Vol. 903. Lecture Notes in Computer Science (LNCS). Springer, 1995, pp. 151–163. DOI: 10.1007/3-540-59071-4_45.

[378] Tobias Schwalb, Graf Philipp, and Klaus D. Müller-Glase. "Monitoring Executions on Reconfigurable Hardware at Model Level." In: *International Workshop on Models@run.time*. Vol. 641. CEUR Workshop Proceedings. CEUR-WS.org, 2010, pp. 96–107. URL: http://ceur-ws.org/Vol-641/.

[379] Ed Seidewitz. "What Models Mean." In: *IEEE Software* 20.5 (2003), pp. 26–32. DOI: 10.1109/MS.2003.1231147.

[380] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. "A component-based middleware platform for reconfigurable service-oriented architectures." In: *Software: Practice and Experience* 42.5 (2012), pp. 559–583. DOI: 10.1002/spe.1077.

[381] Bran Selic. "Personal reflections on automation, programming culture, and model-based software engineering." In: *Automated Software Engineering* 15.3–4 (2008), pp. 379–391. DOI: 10.1007/s10515-008-0035-7.

[382] Bran Selic. "The Pragmatics of Model-Driven Development." In: *IEEE Software* 20.5 (2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146.

[383] Bran Selic. "The Theory and Practice of Modeling Language Design for Model-Based Software Engineering – A Personal Perspective." In: *Generative and Transformational Techniques in Software Engineering III*. Ed. by João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 6491. Lecture Notes in Computer Science (LNCS). Springer, 2011, pp. 290–321. DOI: 10.1007/978-3-642-18023-1_7.

[384] Shane Sendall and Wojtek Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development." In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[385] Bill Shannon. *Java™ Platform, Enterprise Edition (Java EE) Specification, v5*. JSR-244. 2006. URL: https://www.jcp.org/en/jsr/detail?id=244.

[386] Mary Shaw. "Beyond objects: A software design paradigm based on process control." In: *SIGSOFT Softw. Eng. Notes* 20.1 (1995), pp. 27–38. DOI: 10.1145/225907.225911.

[387] Mary Shaw. "Everyday Dependability for Everyday Needs." In: *International Symposium on Software Reliability Engineering*. ISSRE '02. (keynote). IEEE, 2002, pp. 7–11.

[388] Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. "Awareness Requirements for Adaptive Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, 2011, pp. 60–69. DOI: 10.1145/1988008.1988018.

[389] Carlos Eduardo da Silva and Rogerio de Lemos. "Using Dynamic Workflows for Coordinating Self-adaptation of Software Systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '09. IEEE, 2009, pp. 86–95. DOI: 10.1109/SEAMS.2009.5069077.

[390] Brian Cantwell Smith. "Procedural Reflection in Programming Languages." PhD thesis. Massachusetts Institute of Technology, 1982. URL: http://hdl.handle.net/1721.1/15961.

[391] Brian Cantwell Smith. "Reflection and Semantics in LISP." In: *Symposium on Principles of Programming Languages*. POPL '84. ACM, 1984, pp. 23–35. DOI: 10.1145/800017.800513.

[392] *Software Engineering – Software Life Cycle Processes – Maintenance*. ISO/IEC 14764, IEEE Std 14764-2006 (Revision of IEEE Std 1219-1998). 2006. DOI: 10.1109/IEEESTD.2006.235774.

[393] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu. "Model-Driven Engineering for Autonomic Provisioned Systems." In: *International Computer Software and Applications Conference*. COMPSAC '08. IEEE, 2008, pp. 1110–1115. DOI: 10.1109/COMPSAC.2008.221.

[394] Ian Sommerville. *Software Engineering*. 9th ed. Addison-Wesley, 2010. ISBN: 0137035152.

[395] Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu, Yanchun Sun, and Hong Mei. "Supporting runtime software architecture: A bidirectional-transformation-based approach." In: *Journal of Systems and Software* 84.5 (2011), pp. 711–723. DOI: 10.1016/j.jss.2010.12.009.

[396] Hui Song, Gang Huang, Yingfei Xiong, Franck Chauvel, Yanchun Sun, and Hong Mei. "Inferring Meta-models for Runtime System Data from the Clients of Management APIs." In: *International Conference on Model Driven Engineering Languages and Systems (MODELS '10)*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Oystein Haugen. Vol. 6395. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 168–182. DOI: 10.1007/978-3-642-16129-2_13.

[397] Hui Song, Yingfei Xiong, Franck Chauvel, Gang Huang, Zhenjiang Hu, and Hong Mei. "Generating Synchronization Engines between Running Systems and Their Model-Based Views." In: *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers*. Ed. by Sudipto Ghosh. Vol. 6002. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 140–154. DOI: 10.1007/978-3-642-12261-3_14.

[398] Hui Song, Xiaodong Zhang, Nicolas Ferry, Franck Chauvel, Arnor Solberg, and Gang Huang. "Modelling Adaptation Policies as Domain-Specific Constraints." In: *International Conference on Model-Driven Engineering Languages and Systems (MODELS '14)*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahao, and Emilio Insfran. Vol. 8767. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 269–285. DOI: 10.1007/978-3-319-11653-2_17.

[399] Diomidis Spinellis. "Notable design patterns for domain-specific languages." In: *Journal of Systems and Software* 56.1 (2001), pp. 91–99. DOI: 10.1016/S0164-1212(00)00089-3.

[400] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien: Springer, 1973. ISBN: 3211811060.

[401] Michael Stein, Alexander Frömmgen, Roland Kluge, Frank Löffler, Andy Schürr, Alejandro Buchmann, and Max Mühlhäuser. "TARL: Modeling Topology Adaptations for Networking Applications." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '16. ACM, 2016, pp. 57–63. DOI: 10.1145/2897053.2897061.

[402] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. 2nd ed. Eclipse Series. Addison-Wesley, 2008. ISBN: 0321331885.

[403] Perdita Stevens. "A Landscape of Bidirectional Model Transformations." In: *Generative and Transformational Techniques in Software Engineering II*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 5235. Lecture Notes in Computer Science (LNCS). Springer, 2008, pp. 408–424. DOI: 10.1007/978-3-540-88643-3_10.

[404] Perdita Stevens. "Bidirectional model transformations in QVT: semantic issues and open questions." In: *Software and Systems Modeling* 9.1 (2010), pp. 7–20. DOI: 10.1007/s10270-008-0109-9.

[405] E. Burton Swanson. "The Dimensions of Maintenance." In: *International Conference on Software Engineering*. ICSE '76. IEEE, 1976, pp. 492–497. URL: http://dl.acm.org/citation.cfm?id=800253.807723.

[406] Daniel Sykes, Domenico Corapi, Jeff Magee, Jeff Kramer, Alessandra Russo, and Katsumi Inoue. "Learning Revised Models for Planning in Adaptive Systems." In: *International Conference on Software Engineering*. ICSE '13. IEEE, 2013, pp. 63–71. DOI: 10.1109/ICSE.2013.6606552.

[407] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. "From Goals to Components: A Combined Approach to Self-management." In: *International Workshop on Software Engineering for Adaptive and Self-managing Systems*. SEAMS '08. ACM, 2008, pp. 1–8. DOI: 10.1145/1370018.1370020.

[408] Michael Szvetits and Uwe Zdun. "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime." In: *Software and Systems Modeling* (2013), pp. 1–39. DOI: 10.1007/s10270-013-0394-9.

[409] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

[410] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. "PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation." In: *International Conference on Automated Software Engineering*. ASE '10. ACM, 2010, pp. 467–476. DOI: 10.1145/1858996.1859092.

[411] Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, Laurence Duchien, and Lionel Seinturier. "Improving Context-awareness in Self-adaptation Using the DYNAMICO Reference Model." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. IEEE, 2013, pp. 153–162. DOI: 10.1109/SEAMS.2013.6595502.

[412] Matthias Tichy and Benjamin Klöpper. "Planning Self-adaption with Graph Transformations." In: *International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE '11)*. Ed. by Andy Schürr, Dániel Varró, and Gergely Varró. Vol. 7233. Lecture Notes in Computer Science (LNCS). Springer, 2012, pp. 137–152. DOI: 10.1007/978-3-642-34176-2_13.

[413] Sven Tomforde, Jörg Hähner, Sebastian von Mammen, Christian Gruhl, Bernhard Sick, and Kurt Geihs. "Know Thyself – Computational Self-Reflection in Intelligent Technical Systems." In: *International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. SASOW '14. IEEE, 2014, pp. 150–159. DOI: 10.1109/SASOW.2014.25.

[414] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. "EMF-IncQuery: An integrated development environment for live model queries." In: *Science of Computer Programming* 98, Part 1 (2015), pp. 80–99. DOI: 10.1016/j.scico.2014.01.004.

[415] Ricardo Valerdi, Elliot Axelband, Thomas Baehren, Barry Boehm, Dave Dorenbos, Scott Jackson, Azad Madni, Gerald Nadler, Paul Robitaille, and Stan Settles. "A research agenda for systems of systems architecting." In: *International Journal of System of Systems Engineering* 1.1–2 (2008), pp. 171–188. DOI: 10.1504/IJSSE.2008.018137.

[416] Giuseppe Valetto and Gail Kaiser. "Using Process Technology to Control and Coordinate Software Adaptation." In: *International Conference on Software Engineering*. ICSE '03. IEEE, 2003, pp. 262–272. DOI: 10.1109/ICSE.2003.1201206.

[417] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates." In: *IEEE Trans. Softw. Eng.* 33.12 (2007), pp. 856–868. DOI: 10.1109/TSE.2007.70733.

[418] Dániel Varró and András Balogh. "The model transformation language of the VIATRA2 framework." In: *Science of Computer Programming* 68.3 (2007), pp. 214–234. DOI: 10.1016/j.scico.2007.05.004.

[419] Emil Vassev and Mike Hinchey. "ASSL: A Software Engineering Approach to Autonomic Computing." In: *IEEE Computer* 42.6 (2009), pp. 90–93. DOI: 10.1109/MC.2009.174.

[420] Emil Vassev and Mike Hinchey. "The ASSL approach to specifying self-managing embedded systems." In: *Concurrency and Computation: Practice and Experience* 24.16 (2012), pp. 1860–1878. DOI: 10.1002/cpe.1758.

[421] Emil Vassev and Mike Hinchey. "The KnowLang Approach to Self-adaptation." In: *Software, Services, and Systems*. Ed. by Rocco De Nicola and Rolf Hennicker. Vol. 8950. Lecture Notes in Computer Science (LNCS). Springer, 2015, pp. 676–692. DOI: 10.1007/978-3-319-15545-6_38.

[422] Norha M. Villegas and Hausi A. Müller. "Managing Dynamic Context to Optimize Smart Interactions and Services." In: *The Smart Internet*. Ed. by Mark Chignell, James Cordy, Joanna Ng, and Yelena Yesha. Vol. 6400. Lecture Notes in Computer Science (LNCS). Springer, 2010, pp. 289–318. DOI: 10.1007/978-3-642-16599-3_18.

[423] Norha M. Villegas, Hausi A. Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. "A Framework for Evaluating Quality-driven Self-adaptive Software Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, 2011, pp. 80–89. DOI: 10.1145/1988008.1988020.

[424] Norha M. Villegas, Gabriel Tamura, Hausi A. Müller, Laurence Duchien, and Rubby Casallas. "DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems." In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Vol. 7475. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 265–293. DOI: 10.1007/978-3-642-35813-5_11.

[425] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013. ISBN: 1481218581. URL: http://dslbook.org/.

[426] Markus Völter. "From Programming to Modeling - and Back Again." In: *IEEE Software* 28.6 (2011), pp. 20–25. DOI: 10.1109/MS.2011.139.

[427] Mira Vrbaski, Gunter Mussbacher, Dorina Petriu, and Daniel Amyot. "Goal Models As Run-time Entities in Context-aware Systems." In: *International Workshop on Models@run.time*. ACM, 2012, pp. 3–8. DOI: 10.1145/2422518.2422520.

[428] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. "On Interacting Control Loops in Self-adaptive Systems." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, 2011, pp. 202–207. DOI: 10.1145/1988008.1988037.

[429] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. "Utility functions in autonomic systems." In: *International Conference on Autonomic Computing*. ICAC '04. IEEE, 2004, pp. 70–77. DOI: 10.1109/ICAC.2004.1301349.

[430] Yiqiao Wang and John Mylopoulos. "Self-Repair Through Reconfiguration: A Requirements Engineering Approach." In: *International Conference on Automated Software Engineering*. ASE '09. IEEE, 2009, pp. 257–268. DOI: 10.1109/ASE.2009.66.

[431] Mark Weiser. "The Computer for the 21st Century." In: *Scientific American* 265.3 (1991), pp. 94–104.

[432] Michel Wermelinger and José Luiz Fiadeiro. "A graph transformation approach to software architecture reconfiguration." In: *Science of Computer Programming* 44.2 (2002). Special Issue on Applications of Graph Transformations (GRATRA '00), pp. 133–155. DOI: 10.1016/S0167-6423(02)00036-9.

[433] Danny Weyns, Mauro Caporuscio, Bahtijar Vogel, and Arianit Kurti. "Design for Sustainability = Runtime Adaptation ∪ Evolution." In: *European Conference on Software Architecture Workshops*. ECSAW '15. ACM, 2015, 62:1–62:7. DOI: 10.1145/2797433.2797497.

[434] Danny Weyns, M. Usman Iftikhar, Sam Malek, and Jesper Andersson. "Claims and supporting evidence for self-adaptive systems: A literature study." In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE, 2012, pp. 89–98. DOI: 10.1109/SEAMS.2012.6224395.

[435] Danny Weyns, Sam Malek, and Jesper Andersson. "FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems." In: *ACM Trans. Auton. Adapt. Syst.* 7.1 (2012), 8:1–8:61. DOI: 10.1145/2168260.2168268.

[436] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl Goeschka. "On Patterns for Decentralized Control in Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw. Vol. 7475. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 76–107. DOI: 10.1007/978-3-642-35813-5_4.

[437] Danny Weyns, Stepan Shevtsov, and Sabri Pllana. "Providing Assurances for Self-Adaptation in a Mobile Digital Storytelling Application Using ActivFORMS." In: *International Conference on Self-Adaptive and Self-Organizing Systems*. SASO '14. IEEE, 2014, pp. 110–119. DOI: 10.1109/SASO.2014.23.

[438] Jules White, Douglas Schmidt, and Aniruddha Gokhale. "Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation." In: *Software and Systems Modeling* 7.1 (2008), pp. 3–23. DOI: 10.1007/s10270-007-0057-9.

[439] Jon Whittle, John Hutchinson, and Mark Rouncefield. "The State of Practice in Model-Driven Engineering." In: *IEEE Software* 31.3 (2014), pp. 79–85. DOI: 10.1109/MS.2013.65.

[440] Jon Whittle, Pete Sawyer, Betty H.C. Bencomo Nelly and Cheng, and Jean-Michel Bruel. "RELAX: a language to address uncertainty in self-adaptive systems requirement." In: *Requirements Engineering* 15.2 (2010), pp. 177–196. DOI: 10.1007/s00766-010-0101-0.

[441] David Wile. "Lessons learned from real DSL experiments." In: *Science of Computer Programming* 51.3 (2004), pp. 265–290. DOI: 10.1016/j.scico.2003.12.006.

[442] Martin Wirsing, ed. *Report on the EU/NSF Strategic Workshop on Engineering Software-Intensive Systems*. ERCIM, 2004. URL: http://www.ercim.eu/EU-NSF/sis.pdf.

[443] Murray Woodside, Greg Franks, and Dorina C. Petriu. "The Future of Software Performance Engineering." In: *Future of Software Engineering*. FOSE '07. IEEE, 2007, pp. 171–187. DOI: 10.1109/FOSE.2007.32.

[444] Yihan Wu, Ying Zhang, Yingfei Xiong, Xiaodong Zhang, and Gang Huang. "Towards RSA-based HA configuration in Cloud." In: *International Workshop on Models@run.time*. Vol. 1079. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 96–102. URL: http://ceur-ws.org/Vol-1079/.

[445] Eric Yuan, Naeem Esfahani, and Sam Malek. "A Systematic Survey of Self-Protecting Software Systems." In: *ACM Trans. Auton. Adapt. Syst.* 8.4 (2014), 17:1–17:41. DOI: 10.1145/2555611.

[446] Ji Zhang and Betty H. C. Cheng. "Model-based Development of Dynamically Adaptive Software." In: *International Conference on Software Engineering*. ICSE '06. ACM, 2006, pp. 371–380. DOI: 10.1145/1134285.1134337.

[447] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. "Modular Verification of Dynamically Adaptive Systems." In: *International Conference on Aspect-oriented Software Development*. AOSD '09. ACM, 2009, pp. 161–172. DOI: 10.1145/1509239.1509262.

All links were last followed on January 29, 2017.

Part IV

APPENDIX

A EUREMA LANGUAGE

In this appendix, we discuss in detail and at a more technical level the EUREMA language. Particularly, we discuss the metamodel of the language including well-formedness in Section A.1, the mapping of abstract syntax elements to elements of the concrete syntax in Section A.2, the grammar for expressing conditions in FLDs to branch the control flow in Section A.3, and finally, the grammar for expressing triggering conditions of feedback loops in LDs in Section A.4.

A.1 METAMODEL AND WELL-FORMEDNESS

As discussed in Section 2.1, a metamodel defines a modeling language in terms of its abstract syntax while for visual modeling a concrete syntax based on the metamodel is employed. The FLDs and LDs shown in this thesis use the concrete syntax. In the following, we discuss the abstract syntax of EUREMA together with the well-formedness rules constraining and asserting the use of the constructs of the abstract syntax.

The EUREMA metamodel conceptually consists of three parts that address (1) the specification of feedback loops as visualized by FLDs, (2) the specification of layered architectures as visualized by LDs, and (3) that finally support the EUREMA interpreter in executing EUREMA models. These three aspects can be identified in the EUREMA metamodel depicted in Figure 99 on the next page: the elements shaded light-gray are concerned with (1), the hollow elements with (2), and the elements shaded dark-gray are relevant for (3). In the following, we discuss each of this aspect in detail together with the corresponding well-formedness rules.

(1) Specification of Feedback Loops

The core concept of this aspect is the Megamodel that describes a feedback loop and that is modeled by an FLD. It has a mandatory identifier (uid) and name as well as an optional description. It further contains different kinds of Operations that are linked with each other by Transitions. A transition links exactly two operations by connecting an Exit[1] of the operation to an Entry of the other operation. This defines the control flow between operations and makes the operations and transitions the Executable elements of all MegamodelElements. In general, an operation may have arbitrary many entries and exits while an entry has at least one incoming and an exit exactly one outgoing transition. In this context, both operations that are connected by a transition must be contained in the same megamodel. This constraint is defined by the invariant shown in Listing 3 on Page 307. The invariant makes use of a helper feature to navigate from an Operation to the Megamodel containing the operation, which is not directly supported by the metamodel.

1 The isActivated attribute of an Exit reflects which exit of an operation is activated after an execution of the operation. The attribute is set to true for the activated exit and to false for all the other exits of the operation. The activated exit determines the branch for continuing the control flow after executing the operation. Therefore, the isActivated attribute is required for executing (*cf.* Chapter 6) but not relevant for modeling (*cf.* Chapter 5) feedback loops.

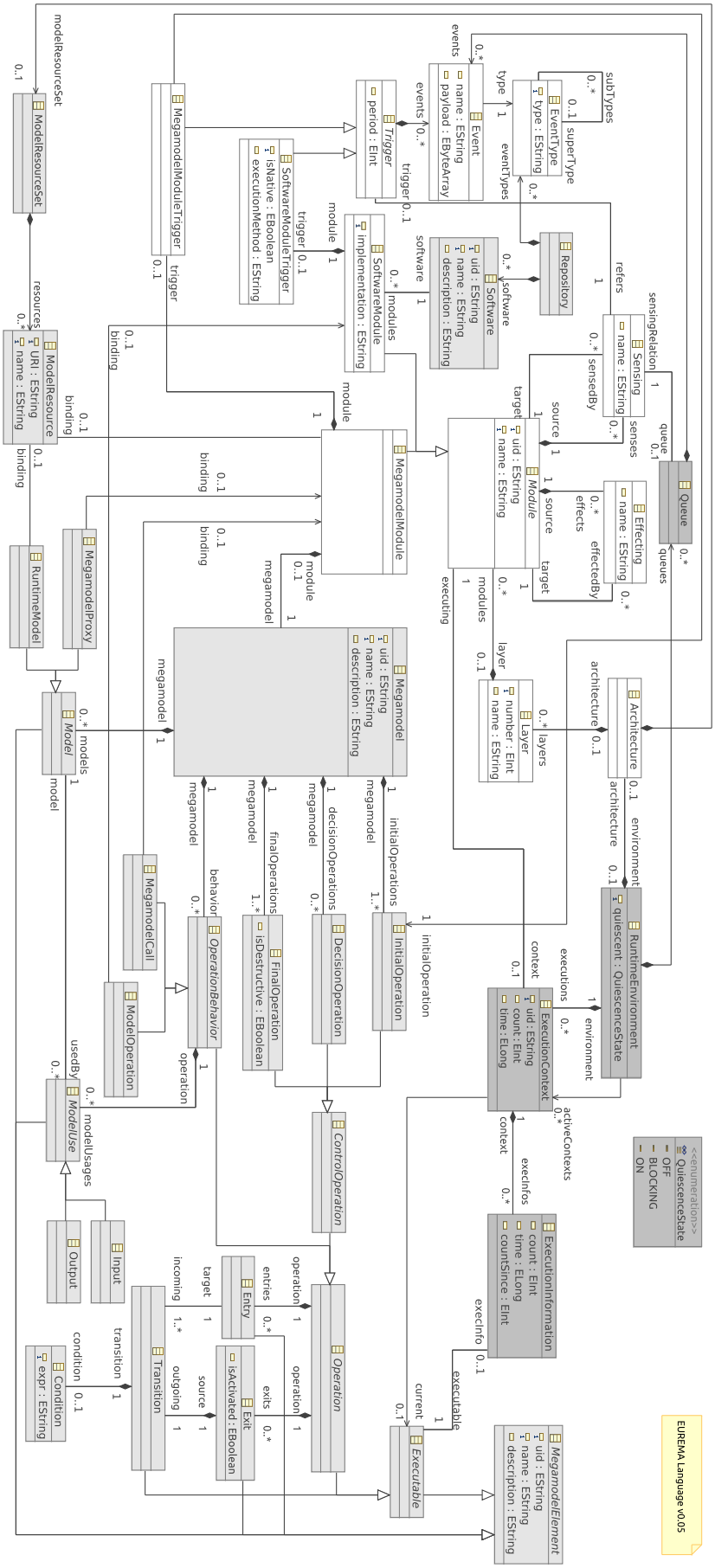


Figure 99: Metamodel of the EUREMA Language Including Runtime Concepts.

Listing 3: Well-formedness Rule of Transitions.

```
context Transition
/* A Transition connects two Operations that must be part of the same Megamodel. */
inv transitionWithinMegamodel:
self.source.operation.megamodel = self.target.operation.megamodel

context Operation
/* Helper feature to navigate from an Operation to its Megamodel. */
def getMegamodelHelper: megamodel : Megamodel =
if self.oclIsTypeOf(InitialOperation)
then self.oclAsType(InitialOperation).megamodel
else if self.oclIsTypeOf(DecisionOperation)
    then self.oclAsType(DecisionOperation).megamodel
    else if self.oclIsTypeOf(FinalOperation)
        then self.oclAsType(FinalOperation).megamodel
        else if self.oclIsKindOf(OperationBehavior)
            then self.oclAsType(OperationBehavior).megamodel
            else null
            endif
        endif
    endif
endif
```

The different types of operation in a megamodel are the following ones: At least one InitialOperation and one FinalOperation are required, which are the megamodel's initial and final states, respectively. A FinalOperation can be *destructive* defining that an instance of the megamodel is executed exactly once and destroyed afterwards (see attribute isDestructive of a final operation). A megamodel may contain DecisionOperations. Such an operation branches the control flow in a feedback loop based on Conditions that are annotated to the DecisionOperation's (through Exits) outgoing Transitions. A concrete condition is stored by the expr attribute of a Condition. The language to construct such conditions is discussed in Appendix A.3. The well-formedness rule shown in Listing 4 assures that conditions can only be annotated to outgoing transitions of decision operations but not of the other types of operations. InitialOperations, FinalOperations, and DecisionOperations are ControlOperations as they determine the control flow in a feedback loop in terms of where the control flow starts, how it branches, and where it terminates.

Listing 4: Well-formedness Rule of Operations with respect to Conditions.

```
context Operation
/* A Transition must have a Condition if the source Operation is a
 * DecisionOperation, otherwise not. */
inv operationCondition: if self.oclIsTypeOf(DecisionOperation)
  then self.exits->forAll(e | e.outgoing.condition <> null)
  else self.exits->forAll(e | e.outgoing.condition = null)
endif
```

The actual behavior of a megamodel respectively feedback loop is defined by arbitrary many OperationBehaviors that may use Models as Input or Output (see the ModelUse element connecting OperationBehaviors and Models). An OperationBehavior is also an Operation and refined either to a ModelOperation as an atomic computation unit or to a MegamodelCall as a *complex model operation* invoking another megamodel.

A Model is contained in a Megamodel and it can be used as an input or output by arbitrary many OperationBehaviors, which enables the sharing of a model among different operations. A model is either a RuntimeModel, which is an arbitrary model as an instance of an arbitrary metamodel, or a MegamodelProxy, which refers to a concrete megamodel used as a reflection model in layered architectures.

The relationships of an Operation to an Entry and Exit are constrained by invariants depending on the type of operation in order to achieve well-formedness. Multiple entries (exits) of an operation are different entry (exit) points of the operation and therefore correspond to different ways to initiate (continue the control flow after) the execution of the operation. These constraints are shown in Listing 5 and briefly described in the following.

Listing 5: Well-formedness Rules of Operations with respect to Entries and Exits.

```
context InitialOperation
/* An InitialOperation must have no Entry. */
inv initialOperationEntry: self.entries->size() = 0
/* An InitialOperation must have exactly one Exit. */
inv initialOperationExit: self.exits->size() = 1

context FinalOperation
/* A FinalOperation must have no Exit. */
inv finalOperationEntry: self.exits->size() = 0
/* A FinalOperation must have exactly one Entry. */
inv finalOperationExit: self.entries->size() = 1

context DecisionOperation
/* A DecisionOperation must have exactly one Entry. */
inv decisionOperationEntry: self.entries->size() = 1
/* A DecisionOperation must have at least two Exits. */
inv decisionOperationExit: self.exits->size() >= 2

context OperationBehavior
/* An OperationBehavior must have at least one Entry. */
inv operationBehaviorEntry: self.entries->size() >= 1
/* An OperationBehavior must have at least one Exit. */
inv operationBehaviorExit: self.exits->size() >= 1
```

An initial operation does not have any entries and therefore any incoming transitions since it is the initial state of a feedback loop. Moreover, it has exactly one default exit to deterministically start the execution of the feedback loop. A final operation has exactly one default entry and thus no choices of initiating it. Being the final state of a feedback loop, it does not have any exits and therefore any outgoing transitions. A decision operation has exactly one default entry (alternative entries are not required since the purpose of the operation is just the evaluation of the conditions for branching the control flow) and at least two exits to actually branch the control flow with individual outgoing transitions each of which is annotated with a condition (see the invariant in Listing 4 on the previous page). Thus, the exits of a decision operation depend on the number of branches (*i. e.*, the outgoing transitions of the operation) each of which must have a condition. A model operation or a megamodel call (*i. e.*, an OperationBehavior) has at least one entry and one exit since it exhibits behavior that requires means to enter and return from executing this behavior.

Finally, all elements contained in a megamodel have the common super class, called MegamodelElement, with a mandatory identifier (uid) and name as well as an optional description. The names of elements are constrained by invariants as defined in Listing 6.

Listing 6: Well-formedness Rules of Element Names in a Megamodel.

```
context Megamodel
/* The names of all Operations and Models contained in one Megamodel must be unique. */
inv uniqueOperationModelNames:
  self.initialOperations->selectByKind(MegamodelElement)
    ->union(self.decisionOperations->selectByKind(MegamodelElement))
    ->union(self.finalOperations->selectByKind(MegamodelElement))
    ->union(self.behavior->selectByKind(MegamodelElement))
    ->union(self.models->selectByKind(MegamodelElement))
    ->forAll(me1, me2 | me1 <> me2 implies me1.name <> me2.name)

context Operation
/* The names of Entries of an Operation must be unique. */
inv uniqueEntryNames:
  self.entries->forAll(en1, en2 | en1 <> en2 implies en1.name <> en2.name)
/* The names of Exits of an Operation must be unique. */
inv uniqueExitNames:
  self.exits->forAll(ex1, ex2 | ex1 <> ex2 implies ex1.name <> ex2.name)
/* The names of all outgoing transitions of an Operation must be unique. */
inv uniqueOutgoingTransitionNames:
  self.exits.outgoing->forAll(t1, t2 | t1 <> t2 implies t1.name <> t2.name)
```

The names of all Operations and Models contained within the same Megamodel must be unique in the scope of this Megamodel (*cf.* first invariant in Listing 6). In contrast, the names of Entries (Exits) must be unique in the context of the Operation that contains the entries (exits), which is defined by the second (third) invariant in Listing 6. Finally, the names of transitions that have—through different Exits—the same source operation must be unique in the scope of this operation (see last invariant in Listing 6). These constraints are required to uniquely identify by a name the operation or model within the megamodel, or the entry, exit, and outgoing transition in the context of an operation. The reasons for these constraints will be discussed in the remaining part of this appendix.

(2) Layered Architectures

This aspect is concerned with specifying a layered architecture of a self-adaptive software as visualized by an LD. Such an architecture is represented by the Architecture element that consists of arbitrary many Layers that are numbered bottom-up starting with 0 (see the attribute called number) and that have an optional name. Each layer contains arbitrary many Modules each of which has an identifier (uid) and a name. A module is exclusively located in one layer. Modules may monitor and adapt other modules, which is represented by Sensing and Effecting relationships. Such a sensing or effecting relationship has a name and connects exactly two modules, either the sensing and the sensed module or the effecting and the effected module. In general, a module may sense or effect arbitrary many other modules and it might be sensed or effected by arbitrary many other modules.

A module is either a SoftwareModule or a MegamodelModule. Software modules represent the adaptable software, legacy adaptation components, or implementations of model operations, which are all considered as black boxes. The implementation attribute defines how to initiate the execution of a legacy component or an operation implementation. A software module is an instance of a Software defined by a mandatory identifier (uid) and name as well as an optional description. A Repository keeps such specifications of Software together.

A megamodel module encapsulates a runtime instance of a Megamodel. The specification of a feedback loop (Megamodel/FLD) is directly used as the runtime instance of the loop (instance of the FLD). The specification and the runtime instance collapse because instances can be individually adapted at runtime, which changes their individual specifications. Technically, if a feedback loop should be instantiated multiple times, copies of the original Megamodel are created for each instance and encapsulated in MegamodelModules to provide identifiable contexts for each instance.

When encapsulating a Megamodel in a MegamodelModule, dependencies to other modules and to materialized models have to be resolved by binding elements of the megamodel: ModelOperations must be bound to SoftwareModules implementing these operations, MegamodelCalls (complex model operations) must be bound to MegamodelModules as targets of the invocations, and MegamodelProxies must be bound to other MegamodelModules that are used as runtime models within the newly instantiated module. Such a binding is defined in LDs by a *use* relationship labeled with the variable that has been declared when naming the element to be bound. This calls for having unique names for such elements, particularly for Operations and Models, in the scope of the containing Megamodel (*cf.* Listing 6 on the previous page). Moreover, the RuntimeModels contained in the megamodel must be bound to materialized models described by ModelResources. Such a resource is characterized by a name and a URI, that is, a Uniform Resource Identifier to identify, locate, and load the materialized model to be used in a feedback loop. Model resources are part of a set (see element ModelResourceSet) as part of the overall Architecture. For each of these four kinds of bindings, an invariant assures that corresponding bindings exist when a megamodel is encapsulated in a megamodel module, that is, when the megamodel is going to be used at runtime (see Listing 7). These bindings satisfy the dependencies of a megamodel module encapsulating a megamodel to other modules as well as materialized models.

Listing 7: Well-formedness Rules concerning Bindings of Megamodel Elements.

```
context ModelOperation
/* A ModelOperation must be bound if it is part of a Megamodel used
 * within a MegamodelModule. */
inv bindingExistsForModelOperation:
  self.megamodel.module <> null implies self.binding <> null

context MegamodelCall
/* A MegamodelCall must be bound if it is part of a Megamodel used
 * within a MegamodelModule. */
inv bindingExistsForMegamodelCall:
  self.megamodel.module <> null implies self.binding <> null

context MegamodelProxy
/* A MegamodelProxy must be bound if it is part of a Megamodel used
 * within a MegamodelModule. */
inv bindingExistsForMegamodelProxy:
  self.megamodel.module <> null implies self.binding <> null

context RuntimeModel
/* A RuntimeModel must be bound if it is part of a Megamodel used
 * within a MegamodelModule. */
inv bindingExistsForRuntimeModel:
  self.megamodel.module <> null implies self.binding <> null
```

Concerning such bindings, the binding of a MegamodelModule to a ModelResource need not to be specified by the engineer. It is established by the EUREMA interpreter to handle megamodel modules and the encapsulated megamodels as materialized models in a unified way with the runtime models.

The constructs of Layers, Modules (MegamodelModules and SoftwareModules) and Sensing/Effecting relationships allows engineers to specify architectures of self-adaptive software adopting the external approach and a layered architectural style (*cf.* Section 2.2.3).

To assure that an architecture conforms to the external approach, the architecture has to consist of at least two layers (*i.e.*, the adaptable software and the adaptation engine), which is defined by the first invariant Listing 8 on the next page. Thereby, the lowest layer represents the adaptable software and therefore contains only software modules but not any megamodel modules, that is, feedback loops specified by EUREMA (see second invariant in Listing 8). Consequently, megamodel modules are located at higher layers that represent the adaptation engine of the self-adaptive software.

Considering the layered architectural style, we require that modules at a certain layer can only sense or effect modules located in the adjacently underlying layer (see the invariants in Lines 13-25 of Listing 8). The last two of these four invariants explicitly forbid that a module senses or effects itself, which would otherwise establish a sense or effect relationship within one layer—although this aspect is already covered by the first two of these four invariants. Thus, these invariants ensure that the layering of modules is determined by the sense and effect relationships that have to cross neighboring layers. Consequently, the EUREMA language enforces the strict layering of megamodel and software modules based on their monitoring and adaptation relationships.

The remaining invariants of Listing 8 assure that bindings of a ModelOperation, MegamodelCall, and MegamodelProxy conforms to the principles of layered architectures. Particularly, a ModelOperation can only be bound to a SoftwareModule, providing the implementation of the operation, that is located in the same layer as the operation, more precisely, the same layer as the megamodel module encapsulating the megamodel containing the operation. Similarly, a MegamodelCall can only be bound to a MegamodelModule that is located in the same layer as the call, more precisely, the same layer as the megamodel module encapsulating the megamodel containing the call. These constraints assure that a megamodel module and its used modules are located at the same layer. A megamodel module uses other modules in terms of invoking software modules providing the implementations of its model operations or of invoking other megamodel modules (*i.e.*, feedback loop fragments) via its complex model operations. The motivation is that such modules together form a feedback loop and a feedback loop is located at exactly one layer and not spread across layers. Thus, the bindings of model operations and megamodel calls to software modules respectively megamodel modules must not cross layers.

In contrast, a MegamodelProxy must be bound to a MegamodelModule that is located at the adjacently lower layer than the proxy, more precisely, the adjacently lower layer than the megamodel module encapsulating the megamodel containing the proxy. A megamodel module contains and uses such a proxy to reflect upon another megamodel module, the one to which the proxy is bound. Since in layered architectures, modules can only reflect upon other modules that are located at the adjacently lower layer, the proxy can therefore only be bound to modules at this adjacently lower layer. Thus, the binding of a MegamodelProxy crosses neighboring layers and usually accompanies the sense and effect relationships—also crossing these neighboring layers—between the corresponding reflecting and reflected megamodel modules.

Listing 8: Well-formedness Rules concerning Layered Architectures.

```
context Architecture
/* An architecture should have at least two Layers to conform with
 * the external approach. */
inv architectureExternalApproach: self.layers->size() >= 2

context Layer
/* The lowest layer being the adaptable software only contains SoftwareModules
 * and hence no MegamodelModules (feedback loops). Consequently, MegamodelModules
 * are located at layers 1..n, the adaptation engine. */
inv adaptableSoftwareLayer: self.number = 0 implies
  self.modules->forAll(m | m.oclIsTypeOf(SoftwareModule))

context Module
/* A Module sensing another Module must be located at the adjacently
 * higher Layer than the sensed Module. */
inv layeringSenses:
  self.senses->forAll(s | self.layer.number = s.target.layer.number + 1)
/* A Module effecting another Module must be located at the adjacently
 * higher Layer than the effected Module. */
inv layeringEffects:
  self.effects->forAll(e | self.layer.number = e.target.layer.number + 1)
/* A Module must not sense itself. */
inv noSelfLoopSense: self.senses->forAll(s | s.target <> self)
/* A Module must not effect itself. */
inv noSelfLoopEffect: self.effects->forAll(e | e.target <> self)

/* Bindings */

context ModelOperation
/* A ModelOperation must be bound to a SoftwareModule within the same layer. */
inv bindingOpWithinLayer: self.megamodel.module.layer = self.binding.layer

context MegamodelCall
/* A MegamodelCall must be bound to a MegamodelModule within the same layer. */
inv bindingCallWithinLayer: self.megamodel.module.layer = self.binding.layer

context MegamodelProxy
/* A MegamodelProxy must be bound to a MegamodelModule at the adjacently
 * lower layer. Thus, modules using the proxy can only reflect upon modules
 * at the adjacently lower layer. */
inv bindingProxyAcrossLayer:
  self.megamodel.module.layer.number = self.binding.layer.number + 1
```

Finally, to initiate the execution of a module, triggering conditions are required. The language to specify textually such conditions (*i. e.*, the concrete syntax of the conditions) is discussed in Section A.4 while we discuss here its abstract syntax.

In general, a Trigger may have a period to periodically execute a module. Such a period defines the time in seconds that must elapse after one run of a module before the next run starts. Moreover, it may define a list of Events with a name. The attribute payload (*i. e.*, the content of an event) is only used for representing events that actually occur at runtime. For instance, the payload of an exception event can be the stack trace. Each event in the list of the trigger conforms to a user-defined EventType. Such an event type has a name

(see attribute type) and can be structured in a hierarchy of event types based on single inheritance. All event types are kept together in the Repository. Finally, a trigger refers to a Sensing relationship such that the sensing module is triggered if the sensed module actually emits one of the events specified by the trigger. Thus, the module, for which the trigger is specified, senses another module that emits the corresponding events.

A valid trigger must define at least the period or the list of events (see first invariant in Listing 9) which results in timed- or event-triggered modules, respectively. A trigger that defines both aspects will be activated when at least one of the events defined by the trigger matches an occurred event and when the period between consecutive runs of the module is satisfied. In this context, an event specified in a triggering condition matches an event occurred at runtime if the latter is of the same type or of a subtype of the former event and if both have the same name. If an event in a triggering condition is anonymous (*i. e.*, it has no name), then only the event types are used to identify a match.

Listing 9: Well-formedness Rules concerning Triggers.

```
context Trigger
/* If a Trigger has no Events, it must have a Period. If a Trigger has no
 * Period, it must have at least one Event. Thus, it is not allowed that
 * a Trigger has no period and no events. */
inv validTrigger: not (self.period <= 0 and self.events->size() = 0)

/* Helper feature to obtain the Module from the Trigger. */
def getModuleHelper: module : Module =
  if self.oclIsTypeOf(MegamodelModuleTrigger)
  then self.oclAsType(MegamodelModuleTrigger).module
  else if self.oclIsTypeOf(SoftwareModuleTrigger)
       then self.oclAsType(SoftwareModuleTrigger).module
       else null
       endif
  endif
/* A Trigger must not have a period but at least one event if the related
 * Module is located at Layer 2..n and if it senses (intercepts) a
 * MegamodelModule while the trigger is attached to this sense relationship. */
inv higherLayerModuleTrigger: self.module.layer.number >= 2 and
  self.refers.target.oclIsTypeOf(MegamodelModule) implies
   self.period <= 0 and self.events->size() >= 1

context SoftwareModuleTrigger
/* A native SoftwareModuleTrigger does not require an executionMethod */
inv nativeTrigger: self.isNative implies
  (self.executionMethod = '' or self.executionMethod = null)
/* A non-native SoftwareModuleTrigger does require an executionMethod */
inv nativeTrigger: not self.isNative implies
  (self.executionMethod <> '' and self.executionMethod <> null)

context MegamodelModuleTrigger
/* The InitialOperation of a MegamodelModuleTrigger must be part of the Megamodel
 * encapsulated in the MegamodelModule for which the trigger is specified. */
inv triggerInitialOp: self.initialOperation.megamodel.module = self.module
```

A valid trigger is further constrained for higher-layer modules that sense lower-layer megamodel modules. As we have discussed in Section 5.3, triggering conditions of higher-layer modules define events as interception points of the lower-layer megamodel modules and they must not use the period concept. Hence, triggers for higher-layer modules sensing

and intercepting megamodel modules are purely event-triggered, which is ensured by the constraint in Lines 7-21 of Listing 9. This constraint ensures that no period but at least one event is specified for a trigger if the trigger belongs to a module that is located at the second or a higher layer (layers are numbered bottom-up starting with 0 being the adaptable software) and if this trigger is attached to a sensing relationship pointing to a megamodel module. Thus, the module for which the trigger is specified senses and intercepts a megamodel module.

Triggers are either SoftwareModuleTriggers for SoftwareModules that implement legacy adaptation components or MegamodelModuleTriggers for MegamodelModules that are EUREMA feedback loops. A SoftwareModuleTrigger defines an executionMethod of how to trigger the corresponding software module. In this case, the triggering is performed by the EUREMA interpreter such that we call the trigger to be non-native (see attribute isNative). If the triggering cannot be performed by the EUREMA interpreter but by some glue code, the trigger is native such that no executionMethod needs to be defined (see the two invariants in Lines 23-29 of Listing 9). A MegamodelModuleTrigger additionally points to an InitialOperation that is used as the initial state to start executing the corresponding megamodel module. Consequently, this InitialOperation must be part of the megamodel that is encapsulated by the megamodel module for which the trigger has been defined (see the last invariant in Listing 9).

Finally, the relationship between a module and its trigger is constrained by the following rules that are shown in Listing 10. The first rule defines that a megamodel module must not have a trigger while potentially being invoked by another megamodel module through a megamodel call. Otherwise, conflicts may arise when the module is triggered and invoked concurrently. Likewise, the second rule defines that a software module must not have a trigger while potentially being invoked by a model operation to avoid conflicts. Finally, software modules as part of the adaptable software must not have a trigger since the scheduling and execution of the adaptable software is independent of the adaptation engine. Thus, triggers are only specified for modules being part of the adaptation engine. In this context, a trigger does not necessarily have to be associated to such modules of the adaptation engine since they can be either invoked by other modules or they might be currently inactive and remain as standby variants of active modules in the engine.

Listing 10: Well-formedness Rules concerning Modules and Triggers.

```
1  context MegamodelModule
2  /* A MegamodelModule must not have a trigger and be invoked by any
3   * MegamodelCall at the same time. */
4  inv megamodelModuleTrigger:
5    not (self.trigger <> null and
6    MegamodelCall.allInstances()->exists(call | call.binding = self))
7
8  context SoftwareModule
9  /* A SoftwareModule must not have a trigger and be invoked by any
10  * ModelOperation at the same time. */
11 inv softwareModuleTrigger:
12   not (self.trigger <> null and
13   ModelOperation.allInstances()->exists(op | op.binding = self))
14
15 /* A SoftwareModule part of the adaptable software (i.e., it
16  * is located at Layer 0) must not have a trigger. */
17 inv adaptableSoftwareModuleTrigger:
18   self.layer.number = 0 implies self.trigger = null
```

(3) Execution Support

The final aspect of the metamodel supports the execution of EUREMA models by the interpreter. The RuntimeEnvironment manages the execution of the self-adaptive software, particularly, of the adaptation engine as specified by the Architecture in an LD. The architecture defines the modules to be executed, the dependencies between the modules in terms of sensing, effecting, and using relationships, and the triggers of the modules. This information is used by the interpreter to trigger the modules and especially to execute the MegamodelModules, that is, the feedback loops specified in FLDs. Relying directly on the architectural part (*i. e.*, LD) of an EUREMA model capturing this information, the interpreter uses the latest information to control the execution since the model can be dynamically modified through off-line adaptation (*cf.* Section 5.4).

Therefore, the RuntimeEnvironment maintains an ExecutionContext for each Module that is part of the architecture and that realizes a feedback loop (see the executions relationship). Such contexts support executing feedback loops that are either legacy components and thus SoftwareModules, or feedback loops specified with EUREMA and thus MegamodelModules. Each context maintains runtime information about the execution of its module in terms of count and time. The former describes how often the module has been executed and the latter shows the timestamp when the last execution has finished. When a module is running, its execution context is active. The RuntimeEnvironment maintains a list of active contexts to describe which modules are currently running (see the activeContexts relationship).

Additionally, when executing a MegamodelModule, such a context particularly supports executing the Megamodel that is encapsulated in the module and visualized by an FLD. Knowing the internals (*i. e.*, the megamodel) of such a module, further runtime information are reflected in the metamodel. On the one hand, the ExecutionContext points to the currently executed Executable element of the megamodel, which is either an Operation or a Transition. Thus, the metamodel reflects where the interpreter is in its sequence of executing the megamodel, which is similar to a program counter. In this context, the isActivated attribute of an Exit reflects which exit among all exits of an operation is activated after executing the operation, particularly, the implementation of the operation. The activated exit then determines the branch for continuing the control flow such that the interpreter can properly progress execution.

On the other hand, the execution context maintains ExecutionInformation for each Executable in a Megamodel, namely, count, time, and countSince. The count attribute reflects how often the operation or transition has been executed. The time attribute reflects the timestamp when the last execution of the operation or transition has finished. The countSince attribute only applies for transitions such that it holds the value of 0 for each operation. For a transition, it reflects the number of how often its source operation has been consecutively executed without taking this transition but another outgoing transition. If a transition is taken, the countSince value of this transition is reset to 0. Thus, the execution information maintains data about how often (count) and when (time) an operation or transition has been executed while countSince combines counter and timer data since it reflects how often a transition has not been taken consecutively but another outgoing transition of the same operation.

This ExecutionInformation is used in Conditions for exclusively branching the control flow in a Megamodel. That is, conditions may refer to the count, time, and countSince attributes for each executable element of a megamodel by using specific constructs and may apply arith-

metic and boolean operations on the corresponding values. The language for expressing conditions is discussed in detail in Section A.3.

Moreover, the RuntimeEnvironment is responsible for realizing quiescence, which is required for safely adapting the layer architecture in the context of off-line adaptation (*cf.* Section 5.4). Therefore, the RuntimeEnvironment has the attribute quiescent that reflects the state of quiescence (see the QuiescenceState enumeration). This state can be either (1) QuiescenceState::OFF such that the modules may run without any restrictions, (2) QuiescenceState::BLOCKING such that currently running modules should terminate execution while the initiations of new executions are blocked unless they are required to let already running modules terminate, and finally, (3) QuiescenceState::ON such that no modules are currently running and initiations of new executions are blocked. In this state, an adaptation of the layered architecture can be safely performed. Thus, the state OFF means that quiescence is not required at the moment, BLOCKING means that quiescence should be achieved, and finally, ON means that quiescence has been achieved. Maintaining the quiescence state, the currently running modules (see activeContexts relationship pointing from the RuntimeEnvironment to the ExecutionContext), and the dependencies among modules in terms of sensing, effecting, and using relationships in the metamodel, all the required information are available for the EUREMA interpreter to manage quiescence. For instance, if the quiescence state is BLOCKING, the EUREMA interpreter blocks new initiations of modules and waits until the currently running modules terminate. If no module is running any more, the activeContexts relationship is empty, which is the criterion for the interpreter to switch from BLOCKING to ON. Further details on quiescence in the context of EUREMA are discussed in Section 6.6.

Finally, the RuntimeEnvironment manages Queues for events that are emitted by modules and used for triggering other modules. Therefore, a Queue refers to a Sensing relationship between two modules, along which events can be observed. These events are emitted by the sensed module and used for triggering the sensing module according to the Trigger attached to the same Sensing relationship. A Queue consequently contains the emitted Events.

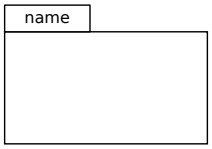
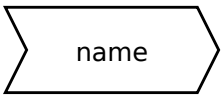
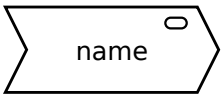
We have focused the discussion of this section on the abstract syntax in terms of the metamodel of EUREMA. Details on the execution semantics of the corresponding EUREMA models and especially of the layered architecture defined by the LD as well as the feedback loops specified by FLDs are presented in Chapter 6.

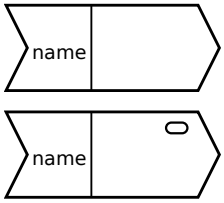
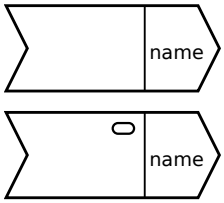
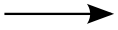
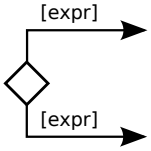
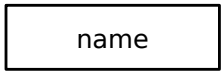
A.2 MAPPING OF THE ABSTRACT AND CONCRETE SYNTAXES

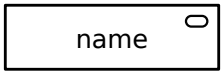
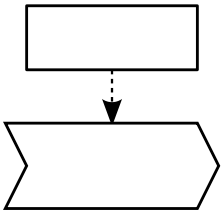
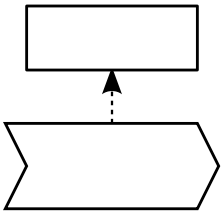
In this section, we map each language concepts as defined in the EUREMA metamodel to its notational element in the Feedback Loop Diagram (FLD) and Layer Diagram (LD). In other words, we map the abstract syntax to the concrete syntax of the EUREMA language. This mapping is shown in Table 13 contrasting the language concepts and the diagram notations while giving a brief description. The first part of the table discusses the concepts and notation with respect to FLDs, the second part with respect to LDs.

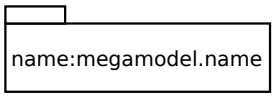
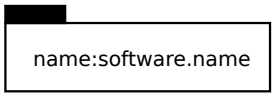
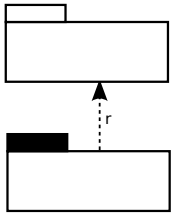
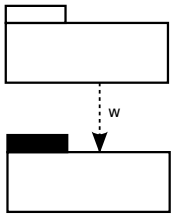
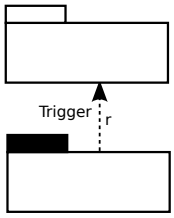
Based on this table, we see that all language concepts with a different concrete syntax are distinguished in the abstract syntax. Moreover, elements of the concrete syntax are basically mapped one-to-one to elements of the abstract syntax. Both of these aspects promote the alignment of the concrete syntax and the abstract syntax of EUREMA.

Table 13: Mapping of the Abstract and Concrete Syntaxes of EUREMA.

Language Concept (Abstract Syntax)	**Diagram Notation** (Concrete Syntax)	**Description**
Feedback Loop Diagram (FLD)		
Megamodel		A Megamodel is visualized by an FLD, particularly by the frame of an FLD that is labeled with the name of the Megamodel.
InitialOperation	 name	An InitialOperation is the initial state of a Megamodel and denoted by a black circle labeled with the name of the InitialOperation.
FinalOperation	 name  name	A FinalOperation is a final state of a Megamodel and denoted by an encircled black circle labeled with its name. A *destructive* FinalOperation, denoted by a cross labeled with its name, additionally defines that an instance of the Megamodel is destroyed after the execution and thus not executed again.
DecisionOperation		A DecisionOperation branches the control flow in a Megamodel and is denoted by a diamond.
ControlOperation	N/A	The ControlOperation is the super class of InitialOperation, FinalOperation, and DecisionOperation. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
ModelOperation		A ModelOperation is an atomic operation that realizes some feedback loop behavior in a Megamodel. It is represented by a hexagon block arrow labeled with its name.
MegamodelCall		A MegamodelCall is a complex operation in a Megamodel that invokes another Megamodel. It is represented by a hexagon block arrow labeled with its name and an icon to distinguish it from a ModelOperation.
OperationBehavior	N/A	The OperationBehavior is the super class of ModelOperation and MegamodelCall. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.

Operation	N/A	The Operation is the super class of ControlOperation and OperationBehavior. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
Entry		An Entry defines the entry point of an Operation. Since each ControlOperation has either no or a default entry (*cf.* Appendix A.1), they are visualized only for ModelOperations and MegamodelCalls as entry compartments of the block arrow, which are labeled with the Entry's name. If there exists exactly one Entry for the ModelOperation or MegamodelCall, it can be omitted in the diagram.
Exit		An Exit defines the exit point of an Operation. Since each ControlOperation has either no or default exits (*cf.* Appendix A.1), they are visualized only for ModelOperations and MegamodelCalls as exit compartments of the block arrow, which are labeled with the Exit's name. If there exists exactly one Exit for the ModelOperation or MegamodelCall, it can be omitted in the diagram.
Transition		A Transition defines the control flow in a Megamodel by connecting an Exit to an Entry of a operation. It is represented by a solid arrow. For ControlOperations, the entries and exits are not visualized such that the Transition is directly attached to the graphical element either representing an InitialOperation, FinalOperation, or DecisionOperation.
Executable	N/A	The Executable is the super class of Operation and Transition. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
Condition		A Condition, more specifically, its attribute expr is annotated to an outgoing Transition of a DecisionOperation to branch the control flow in the Megamodel.
RuntimeModel		A RuntimeModel represents a user-defined model used within a feedback loop / Megamodel. It is represented by a rectangle labeled with its name.

MegamodelProxy		A MegamodelProxy represents a model used within a feedback loop / Megamodel, which is itself a Megamodel. It is represented by a rectangle labeled with its name and an icon to distinguish it from the RuntimeModel.
Model	N/A	The Model is the super class of RuntimeModel and MegamodelProxy. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
Input		An Input describes that a RuntimeModel or MegamodelProxy is consumed by a ModelOperation or MegamodelCall. It is represented by a dotted arrow connecting the former to the latter.
Output		An Output describes that a RuntimeModel or MegamodelProxy is produced or updated by a ModelOperation or MegamodelCall. It is represented by a dotted arrow connecting the latter to the former.
ModelUse	N/A	The ModelUse is the super class of Input and Output. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
MegamodelElement	N/A	The MegamodelElement is the super class of all elements contained in a Megamodel. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
ModelResource	N/A	A ModelResource represents a materialized model. It is not represented in the concrete syntax since it is specified in the context of a RuntimeModel that must be bound to such a resource.
ModelResourceSet	N/A	A ModelResourceSet just collects the ModelResource elements in the model and is not represented in the concrete syntax.
Layer Diagram (LD)		
Architecture		An Architecture is visualized by an LD, particularly, the frame of an LD that is partitioned by Layers.

Layer		Layers partition an Architecture vertically. Each Layer is labeled with its name while its number is mapped bottom-up to its position in the stack of layers. The counting of the position starts with 0.
MegamodelModule		A MegamodelModule encapsulates a Megamodel instance in the Architecture. A module has a name and it is typed by the encapsulated Megamodel.
SoftwareModule		A SoftwareModule encapsulates an instance of a Software in the Architecture. A module has a name and it is typed by the encapsulated Software.
Module	N/A	The Module is the super class of MegamodelModule and SoftwareModule. It is an abstract (not instantiable) class such that is not represented in the concrete syntax.
Software	N/A	A Software specifies type information of SoftwareModules that have been extracted from the class SoftwareModule for reuse. It is not represented in the concrete syntax but specified in the context of SoftwareModules.
Repository	N/A	A Repository just collects the Software elements in the model and is not represented in the concrete syntax.
Sensing		A Sensing relationship describes that one module observes or generally reads another one. It is represented by a dotted arrow labeled with r (for reading) and whose direction indicates the data flow.
Effecting		An Effecting relationship describes that one module adapts or generally writes another one. It is represented by a dotted arrow labeled with w (for writing) and whose direction indicates the data flow.
MegamodelModule-Trigger, SoftwareModuleTrigger, *Trigger*, Event EventType		A Trigger, either a MegamodelModule- or a SoftwareModuleTrigger, is specified textually using the language discussed in Appendix A.4. It is annotated to a Sensing relationship and defined for the sensing module.

binding	name →	MegamodelCalls and MegamodelProxies are bound to MegamodelModules, and ModelOperations to SoftwareModules. A binding is represented by an arrow labeled with the name of the element to be bound and connecting two modules. The module at the source end of the binding relationship contains via a Megamodel the element that is bound to the module at the target end of the binding relationship.[2]

The remaining language concepts of the abstract syntax are elements that are only used internally by the interpreter but not by the engineer for modeling. Hence, such concepts do not have representations in the concrete syntax. These elements are the RuntimeEnvironment, ExecutionContext, and ExecutionInformation.

A.3 CONDITION EXPRESSION LANGUAGE

In this section, we discuss the grammar that defines the language to express conditions for branching the control flow between operations in FLDs. When branching the control flow in FLDs, a decision operation is used that has as many outgoing transitions as the number of required branches. Each outgoing transition is annotated with a condition that either evaluates to true or false. At runtime, the branch whose condition evaluates to true is taken. As a guideline, the conditions of the same decision operation should be disjoint, that is, at most one condition evaluates to true. This allows the EUREMA interpreter to deterministically choose a branch. If more than one condition evaluates to true, the interpreter will raise a runtime failure. Moreover, the conditions of the same decision operation should be complete, that is, exactly one of the conditions must evaluate to true. If an engineer cannot assure completeness (*i. e.*, none of the condition might evaluate to true), a default branch labeled with ELSE should be used. This ELSE branch is taken if all other conditions of the decision operation evaluate to false.

The following grammar shown in Listing 11 defines a textual language to express such conditions for branching the control flow in FLDs. A condition expressed with this grammar is captured in the expr attribute of a Condition in the EUREMA model (see Appendix A.1). Consequently, the EUREMA interpreter may access the condition in the model and use the generated parser to evaluate the condition.

Following the guideline of disjoint and complete conditions, a condition <COND> is either ELSE or an expression <EXP> (Line 1). Such an <EXP> is a boolean expression <BOOL_EXP> or multiple boolean expressions combined by AND or OR that evaluate either to true or false (Line 2). A boolean expression is either a bracketed expression (<EXP>), a negated bracketed expression NOT (<EXP>), or the comparison of two numerical expressions <SUM_EXP> (Lines 3-5). Such a numerical expression is a numerical term <TERM_EXP> or the summation/difference of multiple numerical terms (Line 6). A

2 To uniquely identify the element to be bound based on the binding relationship in the LD, the name of the elements, particularly of Operations and Models contained in a Megamodel are constrained as defined by invariants shown in Listing 6 on Page 309. Particularly, the names have to be unique within the megamodel since assigning a name to an operation or model declares a variable that is used later to label the binding relationship, which realizes the binding of the corresponding operation or model.

numerical term is a unary element <UNARY> or the product/division of multiple unary elements (Line 7) while a single unary element is either a numerical element <ELEMENT> or a negative numerical element (Line 8). A numerical element is either a bracketed numerical expression (<SUM_EXP>) for nesting arithmetical operations (Line 9), a natural number (Lines 10 and 15), or the following numerical values provided by the EUREMA interpreter: CURRENT_TIME results in the current timestamp when evaluating this construct (Line 11). C_OF, C_SINCE, and T_WHEN (Lines 12-14) have an argument <ARG> (Line 16) referring to a name of an Operation or Transition in the Megamodel, in which the condition under evaluation is defined, and they respectively return the count, countSince, and time values of the referred Operation or Transition. These values are maintained by the interpreter in the EUREMA model for each operation and transition. The language elements Operation, Transition, and Megamodel as well as the count, countSince, and time values have been discussed in the context of the EUREMA metamodel in Appendix A.1 respectively A.1.

Listing 11: Grammar of the EUREMA Condition Expression Language.

```
1  <COND>      ::= <EXP> | 'ELSE'
2  <EXP>       ::= <BOOL_EXP> (('AND' | 'OR' ) <BOOL_EXP>)*
3  <BOOL_EXP>  ::= '(' <EXP> ')'
4        | 'NOT' '(' <EXP> ')'
5        | <SUM_EXP> ('=' | '<>' | '>' | '<' | '>=' | '<=') <SUM_EXP>
6  <SUM_EXP>   ::= <TERM_EXP> (('+' | '-') <TERM_EXP>)*
7  <TERM_EXP>  ::= <UNARY> (('*' | '/') <UNARY>)*
8  <UNARY>     ::=  '-' <ELEMENT> | <ELEMENT>
9  <ELEMENT>   ::=  '(' <SUM_EXP> ')'
10       | <NUMBER>
11       | 'CURRENT_TIME'
12       | 'C_OF' '(' <ARG> ')'
13       | 'C_SINCE' '(' <ARG> ')'
14       | 'T_WHEN' '(' <ARG> ')'
15 <NUMBER>    ::= ([ "0"–"9" ])+
16 <ARG>       ::= <ME_NAME> (<SEPARATOR> <ME_NAME>)?
17 // alpha–numeric model element name starting with a letter
18 // space character is allowed but not at the first or last position
19 <ME_NAME>   ::= [ "A"–"Z", "a"–"z" ]
20               (([ " " ])? [ "A"–"Z", "a"–"z", "0"–"9", "_", "–", "." ])*
21 <SEPARATOR> ::= '::'
```

Names of operations and transitions are covered by <ME_NAME> standing for model element name (Lines 19 and 20). To uniquely identify an operation and transition by its name within a Megamodel/FLD, which is the scope for evaluating a condition <COND>, we require that operation names are unique within the Megamodel/FLD while transitions names just have to be unique for all transitions that share the same source operation (*cf.* corresponding invariants described in Listing 6 on Page 309). Thus, a condition refers to an operation by the name of the operation. On the other hand, a condition refers to a transition relatively by using the corresponding source operation to construct a pattern consisting of the operation name, a separator (Line 21), and the transition name. However, if the transition name is unique within the Megamodel/FLD, it is sufficient to use only the transition name without the operation name and separator as a prefix. Nevertheless, a transition has to be uniquely identifiable in a Megamodel/FLD, which might require using the pattern.

From a user's point of view, the names of transitions are not represented and hence hidden in the concrete syntax. In FLDs, a Transition is denoted just by an arrow (*cf.* Ap-

pendix A.2). To ease expressing conditions that use transition names to refer to transitions, we allow the user to use instead the names of the Exits being the exclusive source ends of the transitions. That is, the one-to-one relationship between Exits and Transitions (*cf.* EUREMA metamodel in Appendix A.1) enables the unique identification of the Exit and Transition using one of their names. Thus, for ease of use, engineers may employ the exit names that are represented in the concrete syntax and therefore visualized in FLDs to unambiguously refer to transitions. To assure this ease of use, we require that the names of an Exit and its (single, mandatory) outgoing Transition are the same (see Listing 12).

Listing 12: Invariant Assuring Equal Names of an Exit and its Outgoing Transition.

```
context Exit
/* The name of an Exit should equal the name of the outgoing Transition. */
inv equalExitTransitionNames: self.name = self.outgoing.name
```

Thus, the language allows engineers to define conditions for branching the control flow within a megamodel based on arithmetical and boolean expressions over information such as how often and when operations and transitions of the same megamodel have been executed. This supports basic conditions that may depend on the execution history of the megamodel. For instance, a delay can be specified that has to elapse before the same branch in a megamodel is executed again by subsequent runs of the megamodel. During the delay another branch of the megamodel would be executed.

Concerning the design of the language to express conditions for branching the control flow between operations we have decided in favor of a *generic* language. That is, the language respects the clear separation of the abstraction levels between a megamodel and the operations/models contained in the megamodel. The EUREMA interpreter works at the level of megamodels and considers the (implementations of) operations and models as black boxes. Since conditions are evaluated by the interpreter, they must not utilize concepts that are internal to the operations or models. Otherwise, it couples the interpreter to the individual implementations of the operations or models, which prevents reuse of the interpreter across applications.

However, if application-specific conditions are needed, they can be modeled by appropriate Exits and outgoing Transitions of operations, especially of ModelOperations. Different exits represent different return states of model operations that are determined by the implementations at runtime. That is, the implementation of an operation can be aware of internal aspects of runtime models and it decides which exit/transition will be taken at runtime, for instance, based on these internal aspects. The EUREMA interpreter then uses the exit/transition selected by the implementation to continue the control flow after executing the operation. Moreover, using the generic execution information such as count, countSince, and time for the transitions—which are maintained by the EUREMA interpreter—in conditions, the control flow can be generically branched in addition to such application-specific exits/transitions of operations.

A.4 TRIGGERING CONDITION LANGUAGE

Triggering conditions have been introduced for megamodel modules (*i. e.*, feedback loops specified with EUREMA) that directly control the adaptable software in Section 5.1.3. They have been extended at first for higher-layer megamodel modules that control lower-layer megamodel modules in Section 5.3, and finally for software modules implementing legacy

feedback loops (*i.e.*, feedback loops that have not been developed with EUREMA) in Section 5.4.3. In this section, we discuss the grammar defining the textual language to express such triggering conditions as well as the mapping of the grammar to the EUREMA language. The grammar is shown in Listing 13.

Listing 13: Grammar of the EUREMA Triggering Condition Language.

```
1  <EXP> ::= // Triggers for higher-layer MegamodelModules + SoftwareModules
2            // sensing MegamodelModules at the adjacently lower layer.
3            ( <EUREMA_EVENTS> ';' <INIT> ';' )
4            // Triggers for MegamodelModules and SoftwareModules directly
5            // controlling the adaptable software (other SoftwareModules)
6            // as well as for higher-layer SoftwareModules managing other
7            // SoftwareModules at the adjacently lower layer.
8          | ( <EVENTS> ';' (<PERIOD>)? ';' <INIT> ';' )
9          | ( ';' <PERIOD> ';' <INIT> ';' )
10           // "native" triggering of SoftwareModules at any layer of the
11           // adaptation engine.
12         | 'native'
13 <EUREMA_EVENTS> ::= <EUREMA_EVENT> (',' <EUREMA_EVENTS>)?
14 <EUREMA_EVENT>  ::= ( 'Before' | 'After' | 'OnTransition' )
15                      '[' (<ME_NAME> <SEPARATOR>)? <ME_NAME> ']'
16 <EVENTS>        ::= <EVENT> (',' <EVENTS>)?
17 <EVENT>         ::= <EVENT_TYPE> ('[' <EVENT_NAME> ']')?
18 <EVENT_TYPE>    ::= <EVENT_NAME> (<SEPARATOR> <EVENT_TYPE>)?
19 <SEPARATOR>     ::= '::'
20 <PERIOD>        ::= <NUMBER> 's'
21 <NUMBER>        ::= ["0"-"9"]+
22 <INIT>          ::= ('script:' <EXEC_METHOD>) | <ME_NAME>
23 <EVENT_NAME>    ::= [ "A"-"Z", "a"-"z" ]
24                     ([ "A"-"Z", "a"-"z", "0"-"9", "_" ])*
25 <ME_NAME>       ::= [ "A"-"Z", "a"-"z" ] (([ " " ])?
26                     [ "A"-"Z", "a"-"z", "0"-"9", "_", "-", "." ])*
27 <EXEC_METHOD>   ::= [ "A"-"Z", "a"-"z" ] (([ " " ])?
28                     [ "A"-"Z", "a"-"z", "0"-"9", "_", "-", ".", "(", ")",
29                     "/", "\" ])*
```

A triggering condition is an expression <EXP> with four basic variants that characterize event-based or timed triggering and that depend on the type and layer of the module for which the triggering condition is specified.

At first, we discuss the triggering conditions for a megamodel or software module that is located at a higher layer and that senses megamodel modules at the adjacently lower layer (*cf.* Section 5.3). Such conditions consist of two parts each followed by a semicolon: <EUREMA_EVENTS> and <INIT> (Line 3 in Listing 13). The first part is a comma-separated list of at least one <EUREMA_EVENT> (Line 13). As shown in Line 14f., such an EUREMA event is of a predefined type that is either Before, After, or OnTransition, and it has a name that is parenthesized by square brackets. The name of an event references an operation or transition[3] by name that is contained in the megamodel encapsulated by the sensed mega-

3 For ease of use, engineers may also specify events referencing the Exit that is the source end of a transition to actually reference the transition. This is possible due to the one-to-one relationship between Exits and Transitions (*cf.* Appendix A.1) and our requirement that both have the same name (*cf.* Listing 12 on the previous page). The motivation for supporting this ease of use is that transition names are not represented in the concrete syntax and therefore not visualized in FLDs (*cf.* Appendix A.2) while the exit names are. The same ease of use is also supported in the context of specifying conditions for branching the control flow in FLDs (*cf.* Appendix A.3).

model module. Names of operations and transitions are covered by <ME_NAME> standing for model element name (Line 25f.). To uniquely reference an operation or transition by its name within a megamodel, we require that operation names are unique within the megamodel while transitions names just have to be unique for all transitions that share the same source operation (*cf.* corresponding invariants described in Listing 6 on Page 309).[4] Thus, an event references an operation by just using the name of the operation as its name (see mandatory <ME_NAME> element in Line 15). In contrast, a transition is referenced relatively by using the corresponding source operation to construct a pattern consisting of the operation name, the separator :: (Line 19), and the transition name (see all of the elements <ME_NAME> <SEPARATOR> <ME_NAME> in Line 15). However, if the transition name is unique within the megamodel, it is sufficient to use only the transition name without the operation name and separator as the optional prefix. Thus, engineers specify events of the predefined types that reference operations or transitions of the (megamodel encapsulated in the) sensed megamodel module.

When executing the sensed megamodel module, the EUREMA interpreter synchronously emits corresponding events, that is, it emits a Before event before an operation, an After event after an operation, and an OnTransition event when a transition is executed. Thereby, the name of an emitted event references the corresponding operation or transition by name. If at least one of the events listed in the triggering condition matches an emitted event (*i. e.*, the emitted event is of the same type or of a subtype of the listed event and both events have the same name), the sensed megamodel module is intercepted and blocked at this point by the interpreter to initiate the execution of the higher-layer module. Consequently, the events are similar to interception points of megamodel modules. A higher-layer module that senses a lower-layer megamodel module can therefore intercept this lower-layer module to initiate its own execution by means of the triggering condition.

Here, the second part of the triggering condition, namely <INIT>, becomes relevant as it defines how the execution of the higher-layer module should be initiated. As shown in Line 22 of Listing 13, there are two ways of how to initiate the execution. The first way applies to software modules such as legacy feedback loops and it defines a method that is executed to run the module. Such a method is prefixed by script: and follows the String pattern <EXEC_METHOD> (Lines 27-29). For instance, the method may define some code fragment or also point to some script to be executed. The second way to initiate an execution targets megamodel modules. It references an initial operation by name that should be used to start the execution of a module. Thus, this initialization is the name of an initial operation that conforms to the general String pattern of model elements <ME_NAME> (Line 25f.). Hence, the megamodel module whose execution should be initiated must encapsulate a megamodel that contains an initial operation with the specified name (see last invariant of Listing 9 on Page 313).

The second and third variants of a triggering condition target megamodel modules (*i. e.*, EUREMA feedback loops) and software modules (*i. e.*, legacy feedback loops) that are located at the lowest layer of the adaptation engine and that directly control the adaptable software. Remember that the adaptable software is also represented by software modules. Additionally, these variants target legacy feedback loops at higher layers that manage other

4 We already require that a name of an Exit is unique for all Exits of the same operation to distinguish the different return states of the operation (*cf.* Listing 6 on Page 309). This uniqueness of Exit names together with the uniqueness of a Transition name for all outgoing Transitions of an operation as well as the one-to-one relationship between Exits and Transitions enable the ease of use discussed in the previous footnote.

legacy feedback loops at the adjacently lower layers. Hence, these two variants target modules that control software modules, either the adaptable software or legacy feedback loops.

In general, these two variants of a triggering condition consist of three parts: <EVENTS>, <PERIOD>, and <INIT>. Each of these parts ends with a semicolon to separate the different parts. As shown in Line 8 of Listing 13, if the expression starts with the events, the period is optional. However, if the events are omitted, the period has to be specified (Line 9). This constitutes the two variants of either event-based (Line 8) or timed (Line 9) triggering of modules. In both cases, the initialization part <INIT> is required.

In detail, the different parts of these two variants are defined as follows. The first part <EVENTS> is a comma-separated list of at least one <EVENT> (Line 16). An <EVENT> defines an event whose occurrence will activate the trigger. Such an <EVENT> consists of an <EVENT_TYPE> and an optional <EVENT_NAME> enclosed in square brackets (Line 17). Thus, an event is defined by its type and it can be further specialized by an instance name. If the instance name is omitted, the event is anonymous and solely characterized by its type. An <EVENT_TYPE> (Line 18) consists of a name <EVENT_NAME> for the type. This name can be optionally refined with a <SEPARATOR>, particularly :: (Line 19), and a further <EVENT_TYPE> to describe type hierarchies, for instance, Exception::RuntimeException::NullPointerException. Similarly to the EUREMA events discussed previously, the events listed in the triggering condition are matched against the events emitted by the sensed module. If there is at least one match, the trigger is activated, that is, the execution is initiated unless the period (see next paragraph) delays the execution. However, in contrast to the EUREMA events, the events emitted by sensed software modules such as the adaptable software are typically emitted asynchronously such that the sensed module is not intercepted and blocked to execute the triggered feedback loop.

The second part <PERIOD> is a positive number followed by an s indicating the time unit of seconds (Lines 20 and 21). The period defines the time that has to elapse after a run of the triggered module before the next run will be initiated.

The third part <INIT> describes the initialization point of the triggered module (Line 22) similar to the first variant of triggering conditions discussed previously. For a software module (*i. e.*, a legacy feedback loop), the initialization point has the prefix script: followed by an execution method <EXEC_METHOD> pointing to some code or script to be executed for actually executing the module. For a megamodel module (*i. e.*, an EUREMA feedback loop), the initialization point is the name of the initial operation in which the execution should start when starting to run the module.

In general, the names of events and event types (*i. e.*, <EVENT_NAME>), the names of model elements, especially of operations and transitions (*i. e.*, <ME_NAME>), and the execution methods (*i. e.*, <EXEC_METHOD>) are constrained by the String formats described in Lines 23-29. Thereby, the names of model elements is kept consistent with the grammar defining the language for expressing branching conditions in FLDs (*cf.* Appendix A.3).

These three variants of triggering conditions are processed by the EUREMA interpreter to trigger a module using the initialization part based on events, a period, or both. In contrast, the fourth variant is the native trigger (Line 12) that is not processed by the EUREMA interpreter. Such a trigger denotes that the triggering of a legacy feedback loop is controlled by some glue code integrating the adapting and the adapted modules (*cf.* Section 5.4.3).

A triggering condition specified with this grammar is parsed and represented in the EUREMA model. Therefore, the EUREMA metamodel provides the relevant elements as well as well-formedness constraints to properly capture triggering conditions (*cf.* Appendix A.1). A triggering condition for a megamodel module (software module) is repre-

sented as a MegamodelModuleTrigger (SoftwareModuleTrigger) by the metamodel. Both types of triggers have the common super class Trigger in the metamodel. The mapping of the individual parts of a triggering condition to elements of the EUREMA metamodel is straightforward and outlined in Table 14. We discussed the EUREMA metamodel with all of its elements in Appendix A.1 and therefore we omit discussing the elements relevant for triggering conditions here.

The remaining elements of the grammar (*i.e.*, <EVENT_NAME>, <ME_NAME>, and <EXEC_METHOD> in Lines 23-29 in Listing 13) define only String formats for other grammar elements such that they are not mapped to metamodel elements.

Table 14: Mapping of Triggering Conditions to the EUREMA Metamodel.

Triggering Condition Element	**EUREMA Metamodel Element**
native (Line 12)	SoftwareModuleTrigger with isNative set to true. Otherwise (*i.e.*, default/non-native triggering), isNative is set to false.
<EUREMA_EVENTS> (Line 13)	Trigger.events, that is, a list of Events attached to a Trigger.
<EUREMA_EVENT> (Line 14, 15 and 19)	Event with Event.name set to (<ME_NAME> <SEPARATOR>)? <ME_NAME> and Event.type.type set to "Before", "After", or "OnTransition".
<EVENTS> (Line 16)	Trigger.events, that is, a list of Events attached to a Trigger.
<EVENT> (Line 17)	Event with Event.name set to <EVENT_NAME>, if the condition specifies the optional event name, and Event.type set to <EVENT_TYPE>.
<EVENT_TYPE> (Line 18 and 19)	EventType with EventType.type set to <EVENT_NAME> and EventType.subTypes set to <EVENT_TYPE> (the latter recursively maps the type hierarchy).
<PERIOD> (Line 20 and 21)	Trigger.period set to <NUMBER>.
<INIT> (Line 22)	Based on the two options, either SoftwareModuleTrigger.executionMethod is set to <EXEC_METHOD> or MegamodelModuleTrigger.initialOperation is set to the InitialOperation with the name <ME_NAME>. This InitialOperation must be part of the Megamodel that is encapsulated by the MegamodelModule for which the triggering condition is defined (*cf.* last invariant of Listing 9 on Page 313).

As outlined by this mapping, an <EUREMA_EVENT> and an <EVENT> are handled similarly. The only difference with respect to the mapping is that for EUREMA events, the event types Before, After, and OnTransition are predefined while for the other events user-defined types must be specified (see <EVENTTYPE>). In this context, events of user-defined event types can be anonymous, that is, they must not have a name (see optional event name in Line 17). In contrast, the event name is not optional if the event type is a predefined type (see mandatory event name in Line 14f.). This requirement that events of type Before, After, and OnTransition must have a name is ensured in the EUREMA model by the following invariant in Listing 14.

Listing 14: Invariant Assuring Existence of Names for Events of Predefined EUREMA Types.

```
1 context Event
2 /* If an Event is of type Before, After, or OnTransition, which are
3  * predefined by EUREMA, then the Event name must not be the empty String. */
4 inv eventNameExists: self.type.type = 'Before' or self.type.type = 'After' or
5   self.type.type = 'OnTransition' implies self.name <> ''
```

Moreover, if a triggering condition specifies events of the predefined types, it must not specify a period (*cf.* Line 3 in Listing 13). In the EUREMA model, this is implicitly ensured by the invariant higherLayerModuleTrigger shown in Listing 9 on Page 313. Similar to the triggering condition scheme shown in Line 3 of Listing 13, this invariant applies to modules that sense megamodel modules and therefore the corresponding triggering condition uses events but not any period.

To map an overall triggering condition (*cf.* Lines 3, 8, and 9 in Listing 13) to the EUREMA metamodel, the individual mappings outlined in Table 14 have to be combined. For instance, if a triggering condition specifies events as well as a period, then in the EUREMA model the corresponding Trigger has events attached to its association Trigger.events as well as the attribute Trigger.period is set. In general, the same invariants as described for triggers in the context of the EUREMA metamodel in Listing 9 on Page 313 and Listing 10 on Page 314 apply as well for triggering conditions expressed in the textual language as defined by the grammar.

This grammar has been specified and implemented with the *Java Compiler Compiler (JavaCC)*[5]. The generated parser processes such triggering conditions and creates appropriate elements in the EUREMA model according to the outlined mapping in Table 14. By representing triggering conditions in the EUREMA model, they can be used by the EUREMA interpreter to schedule the execution of the megamodel modules (*i. e.*, EUREMA feedback loops) and software modules (*i. e.*, legacy feedback loops).

5 Java Compiler Compiler (JavaCC): https://javacc.java.net/

B EXECUTION SEMANTICS

We discussed the execution semantics of EUREMA in detail in Chapter 6. There, we specified the semantics in an "[i]nformal but structured and precise" manner, which is according to Selic [383, p. 316] "best suited for human consumption". The goal of this discussion was that the reader easily becomes familiar with the EUREMA execution semantics. In the following, we present a more formal specification of the execution semantics that can be consumed and executed by a computer. This specification is expressed with graph transformations [57, 8] and in particular *Story Diagrams (SDs)* [155]. It comprises multiple SDs that operate on an EUREMA model to be executed. These SDs describe conditions of the execution as well as how the model changes when executing the model and thus the feedback loops specified in the model. Understanding this specification of the execution semantics requires a basic understanding of the EUREMA metamodel discussed in Appendix A.1.

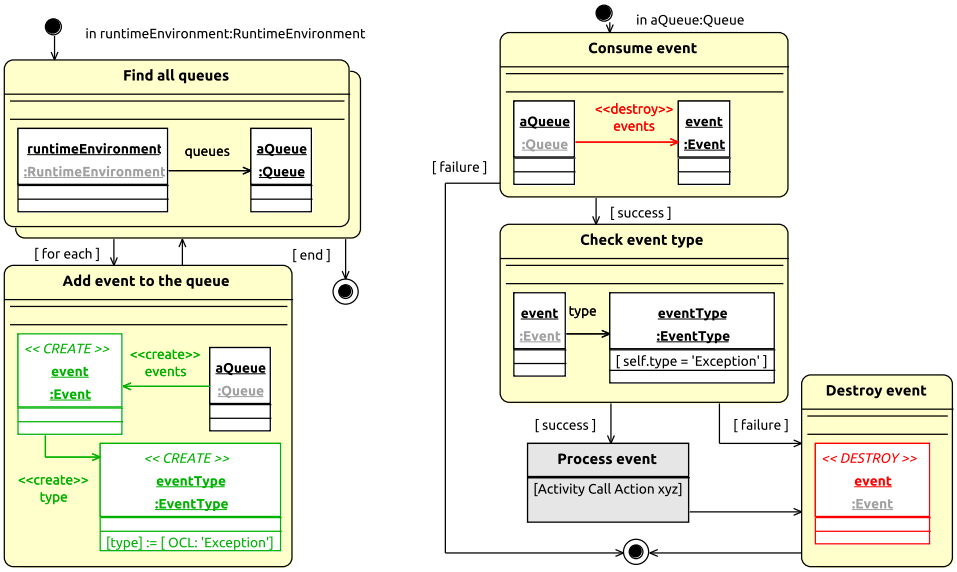
In the following, we introduce the SD formalism in Section B.1 and provide an overview of the SDs defining the EUREMA execution semantics in Section B.2. These SDs are grouped based on their purpose such that we first discuss the initialization of an EUREMA model to be ready for execution in Section B.3. Then, we discuss the triggering and the execution of megamodel modules in Sections B.4 and B.5. Finally, we discuss quiescence of all megamodel modules to enable safe adaptations of the layered architecture in Section B.6.

B.1 STORY DIAGRAMS

Story Diagrams (SDs) have been originally proposed by Fischer et al. [155]. They are a combination of UML Activity Diagrams [329] and graph transformations [57, 8]. That is, each action node of an activity describes and executes a graph transformation. An action node may also invoke another activity or execute some code instead of a graph transformation if code is the more suitable formalism to specify the behavior (*e. g.*, to realize mathematical algorithms that do not work on the graph structure of the model). All action nodes of an activity are structured in a control flow. Consequently, an SD specifies an activity as a flow of action nodes that are usually graph transformations.

To introduce the SD formalism, we use two example activities that are shown in Figure 100. Each of these activities is an SD with one initial and one final node as well as one input parameter. In general, an SD may have multiple final nodes and input parameters as well as output parameters to define return values. The action nodes of an activity are either *Story Patterns (SPs)* (*i. e.*, the yellow nodes) or *Call Actions* (*i. e.*, the gray nodes). An SP describes and executes a graph transformation while a call action invokes either another SD or code. An SD structures the initial, final, and action nodes in a control flow.

A graph transformation expressed in an SP defines a graph rewriting or in other terms an in-place model transformation. Therefore, an SP refers to the elements of a model, which are represented in the abstract syntax of the corresponding language within the SP. An SP further denotes these elements as either black nodes and edges without any annotation, red nodes and edges annotated with ≪DESTROY≫ respectively ≪destroy≫, or green nodes and edges annotated with ≪CREATE≫ respectively ≪create≫. The application of an SP tries to match the pattern consisting of the black and red elements in the model and if it



(a) Add Event. (b) Consume Event.

Figure 100: Two SD Examples for Adding and Consuming Events.

finds a match, the black elements remain unchanged, the red elements are removed, and the green elements are added to the model. Thereby, attribute values of the green and black nodes can be initialized respectively modified. The pattern and the side effect can be extended with constraints expressed in the OCL [327].

Considering the examples in Figure 100, both SDs operate on an EUREMA model and define how events are either added or consumed from a queue. To add events to all queues, the SD shown in Figure 100a uses its input parameter (*i. e.*, the RuntimeEnvironment element) to find all queues and for each queue, it creates an event and adds this event to the queue. To achieve this behavior, the first Find all queues SP is cascaded such that it identifies *all* matches of its pattern, that is, it finds all Queue elements by navigating the queues relationship from the given RuntimeEnvironment element. For each Queue element, the second SP is executed. It creates an Event with its type set to a created EventType element as well as adds this event to the queue by creating the events relationship. Moreover, the type attribute of the EventType is set. After an event has been added to all queues, the SD terminates. This SD particularly illustrates how all matches of a pattern can be identified in the model and how each match can be modified, in this case, by creating elements in the model.

To consume an event of one specific queue, the SD shown in Figure 100b has the specific queue as an input parameter. The first SP tries to consume one event from this queue. Since this SP node is not cascaded, it tries to identify only one match, which is the single event to be consumed. If the queue is empty, there is no match for an event such that the control flow continues along the [failure] edge such that the SD terminates. If there is a match, the SP removes the event from the queue by destroying the events relationship and the control flow continues along the [success] edge. The second SP defines a pattern to check the type of the consumed event. The pattern is extended with an OCL constraint requiring that the type attribute of the associated EventType is set to Exception. If the extended pattern is matched, the control flow continues along the [success] edge, otherwise along the [failure] edge. In the first case, the Process event action node is executed, which invokes another SD to actually process the event, and afterwards, the event is destroyed and thus removed

from the model. In the latter case, the event is not processed but immediately destroyed. This SD particularly illustrates how the control flow is branched depending on the success of identifying a match, how nodes and edges are removed from the model, how constraints extending a pattern are specified, and how call actions are used to invoke other SDs.

To sum up, the SD formalism is a means to specify modifications of a model. Since an SD is executable, we can use SDs to specify the execution semantics if EUREMA in an operational manner, that is, we specify the computations that are performed when executing the individual concepts of the EUREMA language. Therefore, we use a tool suite[1] comprising a modeling editor, interpreter, and debugger for SDs. The resulting SDs defining the EUREMA execution semantics are discussed in the following sections.

B.2 OVERVIEW

In this section, we provide an overview of the 17 SDs that together specify the EUREMA execution semantics. The overview helps in understanding the interplay of these SDs since there are several invocations among them. One of these SD is the main activity as it is the single entry point to the specification of the semantics. This activity is shown in Figure 101.

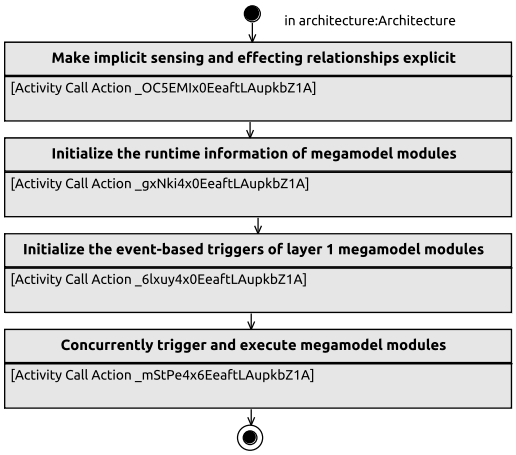


Figure 101: Main SD Defining the EUREMA Execution Semantics.

This SD sequentially invokes other SDs that will be discussed in the following sections. The first three invocations initializes the EUREMA model to be executed. This initialization makes the implicit sensing and effecting relationships explicit and sets up the runtime information of megamodel modules as well as the event-based triggers of megamodel modules at layer 1, that is, the lowest layer of the adaptation engine (see Section B.3). The last invocation defines the execution of the triggers for megamodel modules at layer 1 and the actual execution of the megamodel modules at any layer (see Sections B.4 and B.5).

These four invocations result in sub-invocations of SDs, which is represented by the call graph for the whole SD-based specification of the execution semantics in Figure 102. This graph references the individual SDs by pointing to their sections and figures that discuss and depict them such that it provides an overview of the dependencies between the SDs.

Moreover, the call graph shows the recursive invocations when a megamodel module is executed in the scope of executing another megamodel module (see bended edges). This is either case when an operation and in particular a megamodel call of a module is executed

1 *Story-Driven Modeling (SDM) Tools*: http://www.mdelab.de.

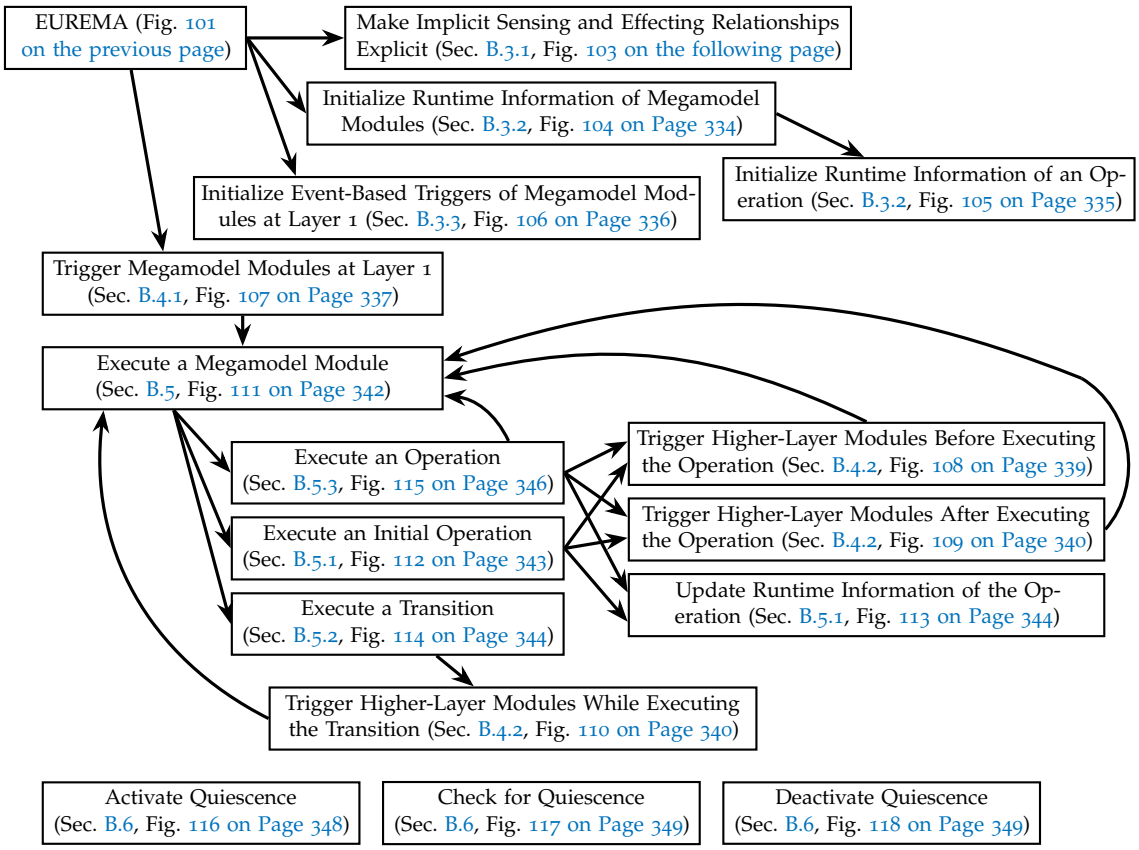


Figure 102: Call Graph of the SD-based Specification.

that invokes another module or when the execution of a module is intercepted—before or after executing an operation or while executing a transition—to execute a higher-layer module. Finally, the call graphs depicts the SDs that define the activation, checking, and deactivation of quiescence. The checking identifies when a quiescent state has been reached when quiescence has been activated before. These SDs are not invoked by any other SD as they can be executed at any time when quiescence of the adaptation is required.

In the following sections, we discuss the individual SDs that constitute the specification of the EUREMA execution semantics.

B.3 INITIALIZATION

After the creation of an EUREMA model (*i. e.*, after an engineer has specified and developed the self-adaptive software with the feedback loops) and before the actual execution of the model, initialization steps establish information in the model that is required for the execution. This information concerns the elements of the language that support the execution and that are not specified by the engineer (*cf.* Appendix A.1). The initialization happens once before the execution of an EUREMA model. Consequently, the following SDs specifying the initialization steps are executed once.

B.3.1 *Explicit Sensing and Effecting Relationships for Megamodel Modules*

The first initialization step makes implicit sensing and effecting relationships between megamodel modules explicit in the EUREMA model. As discussed in Section 5.3, if a megamodel module senses and effects another megamodel module that uses other megamodel modules, then it also senses and effects these other modules—even though this is not explicitly defined in the LD. The initialization step specified by the SD shown in Figure 103 makes these implicit sensing and effecting relationships explicit for the execution.

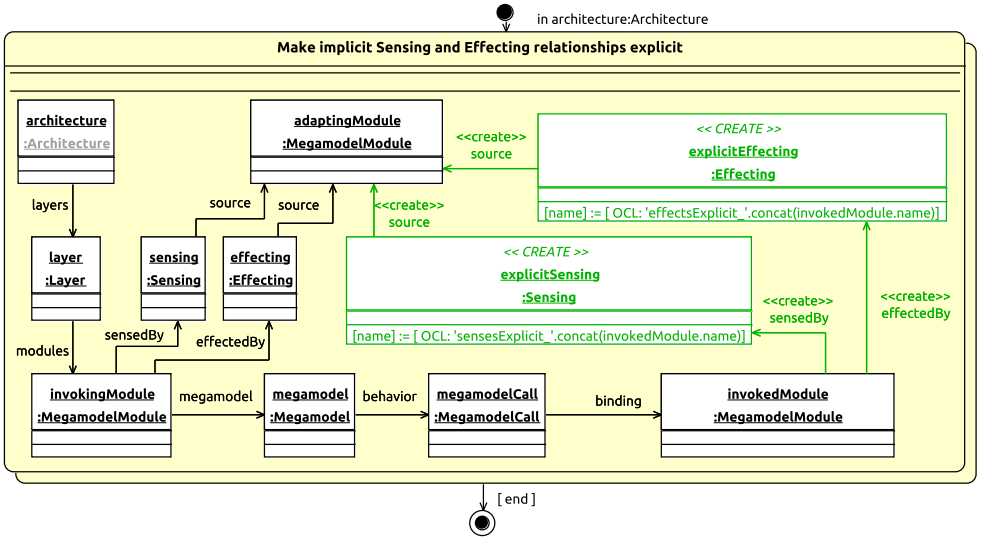


Figure 103: Making Implicit Sensing and Effecting Relationships Explicit.

This SD has the single Architecture element of the EUREMA model as an input parameter. It contains one SP node (*i. e.*, the yellow node) defining the pattern that a megamodel module named invokingModule of any layer of the architecture is sensed and effected by another megamodel module named adaptingModule. Moreover, the invokingModule uses another megamodel module named invokedModule by means of invocations from the megamodelCall (*i. e.*, a complex model operation) within its megamodel. Consequently, the adaptingModule also senses and effects the invokedModule, which is made explicit by creating corresponding sensing and effecting relationships between these two modules. This SP node is cascaded, which indicates that the SP is applied to *all* occurrences of the pattern in the model such that all implicit sensing and effecting relationships are made explicit in the model.

With respect to the execution, explicit sensing and effecting relationships ease the triggering of a higher-layer megamodel module (*cf.* Appendix B.4.2) that senses and effects multiple lower-layer megamodel modules composing a modular feedback loop. Otherwise, the implicit relationships have to be obtained every time when executing an invoked lower-layer megamodel module to check for any triggering of higher-layer megamodel modules.

B.3.2 *Initializing the Runtime Information of Megamodel Modules*

The second initialization step creates and initializes the necessary elements in the model that capture runtime information concerning the execution of megamodel modules. The elements and the captured information have been discussed in Section 6.4.4 and Appendix A.1. This initialization step is specified by the SD shown in Figure 104.

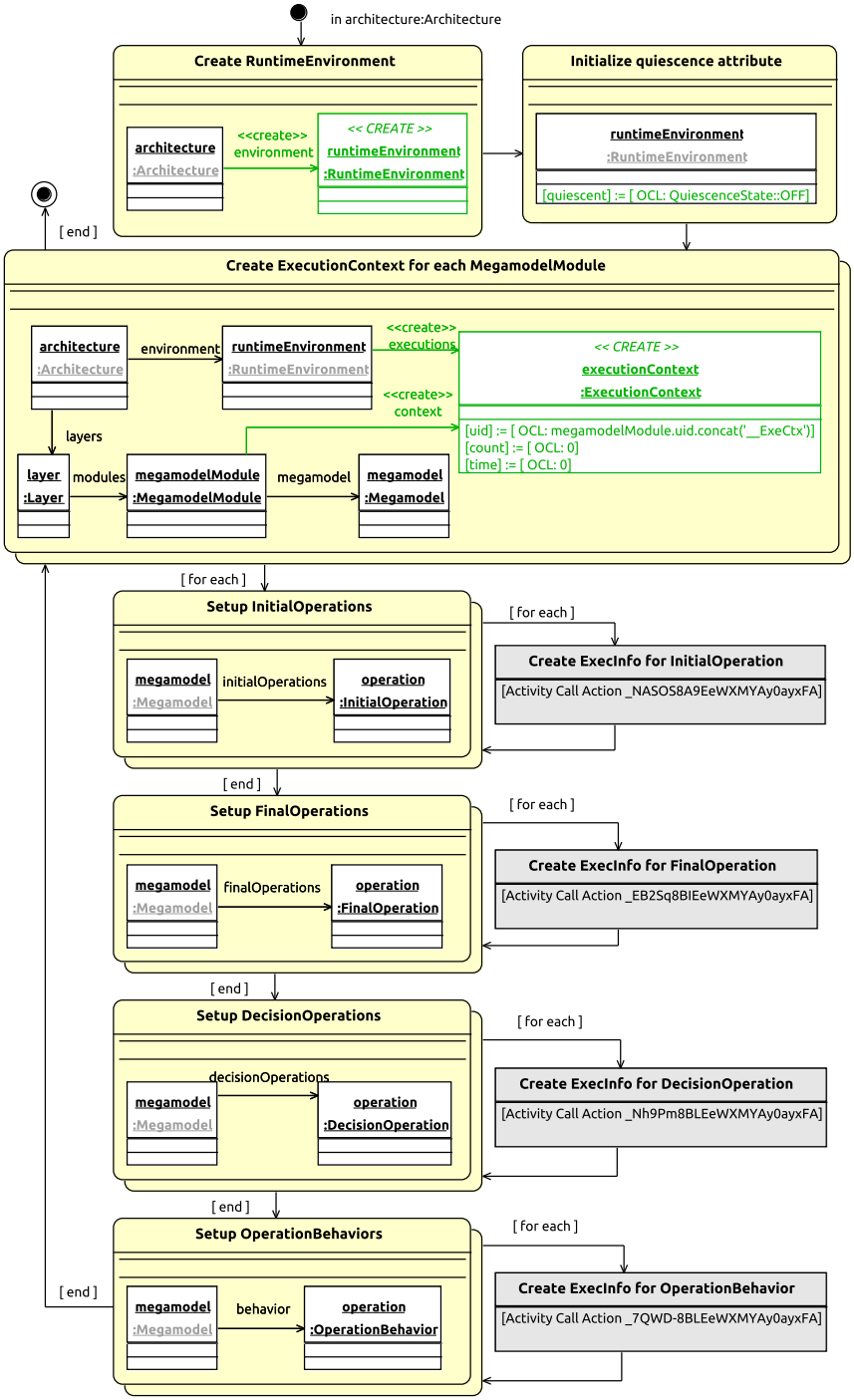


Figure 104: Creating and Initializing the Runtime Information of Megamodel Modules.

For the given Architecture element (see input parameter of the SD), the first SP creates the single RuntimeEnvironment element in the model, which is the root element for all elements that capture runtime information. The second SP initializes the attribute quiescent of the created RuntimeEnvironment element with OFF meaning that the adaptation engine is not in and should not reach a quiescent state. The third SP creates for each MegamodelModule (note the cascaded SP node) of any layer of the architecture an individual ExecutionContext that maintains the runtime information of how often the corresponding module has been executed (see attribute count) and when its last execution has finished (see attribute time).

These two attributes are initialized with zero and the attribute uid with an identifier based on the associated megamodelModule. Moreover, an ExecutionContext maintains a model counter (similar to a program counter) in terms of the current relationship that points to the currently executed operation or transition of its associated megamodel module or to null if the module is not running. This relationship of an ExecutionContext is used when executing the associated megamodel module (see Appendix B.5). Finally, the SP identifies the single megamodel encapsulated in each megamodelModule.

The remaining SPs are executed for each MegamodelModule/Megamodel identified by the previous SP. They match either all InitialOperations, FinalOperations, DecisionOperations, or OperationBehaviors of the Megamodel.[2] The OperationBehaviors cover the ModelOperations and MegamodelCalls (*i. e.*, the basic and the complex model operations). For each matched operation, that is, in the end for all operations of the Megamodel, an action calls the SD shown in Figure 105, which initializes the runtime information for an operation. Hence the SD shown in Figure 104 iterates over all MegamodelModules of the architecture and for each of them, it initializes the runtime information and iterates over all operations of the encapsulated Megamodel to initialize the runtime information for each operation.

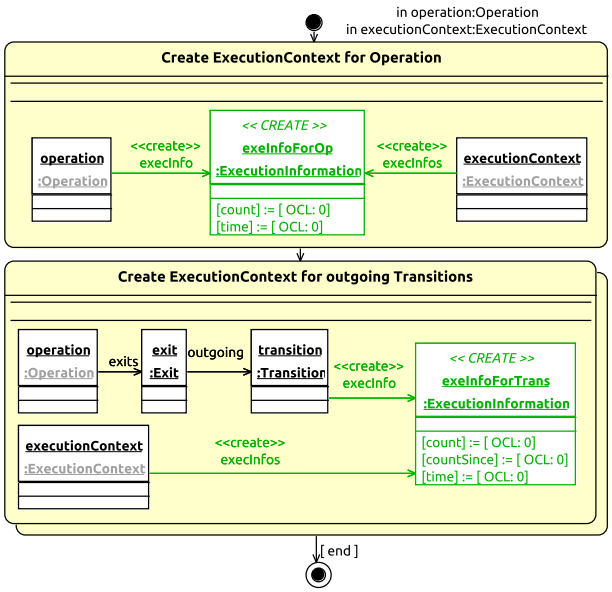


Figure 105: Creating and Initialization the Runtime Information of an Operation.

The SD shown in Figure 105 that is called for each operation initializes the runtime information for an operation as well as for each outgoing transition of the operation. The first SP creates for the given operation and executionContext an ExecutionInformation element that captures how often the operation has been executed (see attribute count) and when the last execution has finished (see attribute time). Both attributes are initialized with zero. The second SP creates for each outgoing Transition of the given operation an ExecutionInformation element that captures how often the transition has been executed (see attribute count), how often the transition has not been executed consecutively but any other outgoing transition of the operation (see attribute countSince), and when the last execution of the transition has finished (see attribute time). These three attributes are initialized with zero.

2 The EUREMA metamodel supports navigating from a Megamodel to operations based on these types but not to all operations contained in the Megamodel regardless of the operation type (*cf.* Appendix A.1). Consequently, for each of these four operation types, an SP is required to navigate to all operations of the Megamodel.

B.3.3 *Initializing the Triggers of Megamodel Modules*

As discussed in Section 6.4.2, megamodel modules at the lowest layer of the adaptation engine that directly sense and effect the adaptable software are asynchronously triggered based on events emitted by the software, based on time, or based on a combination of events and time. To support the asynchronous triggering based on events, an event queue is required for each of these megamodel modules. Such queues are represented in the EUREMA model and they are all created by one run of the SD shown in Figure 106.

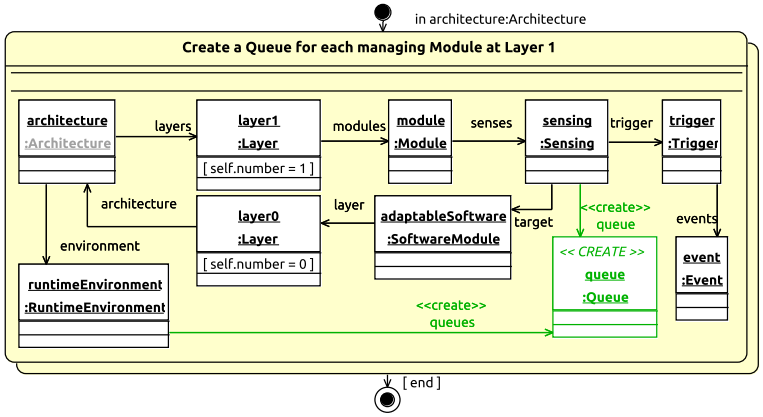


Figure 106: Initializing the Queues for Megamodel Modules at Layer 1.

This SD creates for each module located at the layer 1 (*i. e.*, the lowest layer of the adaptation engine) that senses the adaptable software at layer 0 (*i. e.*, the lowest layer of the overall self-adaptive software) a Queue element associated to the corresponding sensing relationship and to the runtimeEnvironment managing all queues if the trigger of the module specifies at least one event. Hence, the trigger is event-based, which requires a queue to capture the runtime events emitted by the adaptable software. These runtime events are then consumed from the queue to trigger the execution of the corresponding megamodel module, which we discuss in the following section.

B.4 TRIGGERING OF MEGAMODEL MODULES

The execution of a megamodel module starts when the module is triggered. The execution semantics of triggers has been discussed in Section 6.4.2 where we distinguished two general cases. The first case considers megamodel modules that are located at the lowest layer of the adaptation engine (*i. e.*, at layer 1) while the second case considers megamodel modules at higher layers of the engine. These two cases differ in their triggering strategies.

B.4.1 *Triggering Megamodel Modules at Layer 1*

We start with the first case in which a megamodel module at layer 1 is asynchronously triggered by events emitted from the adaptable software, by time, or by a combination of events and time (*cf.* Section 6.4.2). The SD defining the triggering of such a module is shown in Figure 107. If there is more than one of such a module, these modules are triggered and executed concurrently (*cf.* Section 6.3). Therefore, an individual SD as shown in Figure 107 is executed for each of these modules and all of them are executed concurrently.

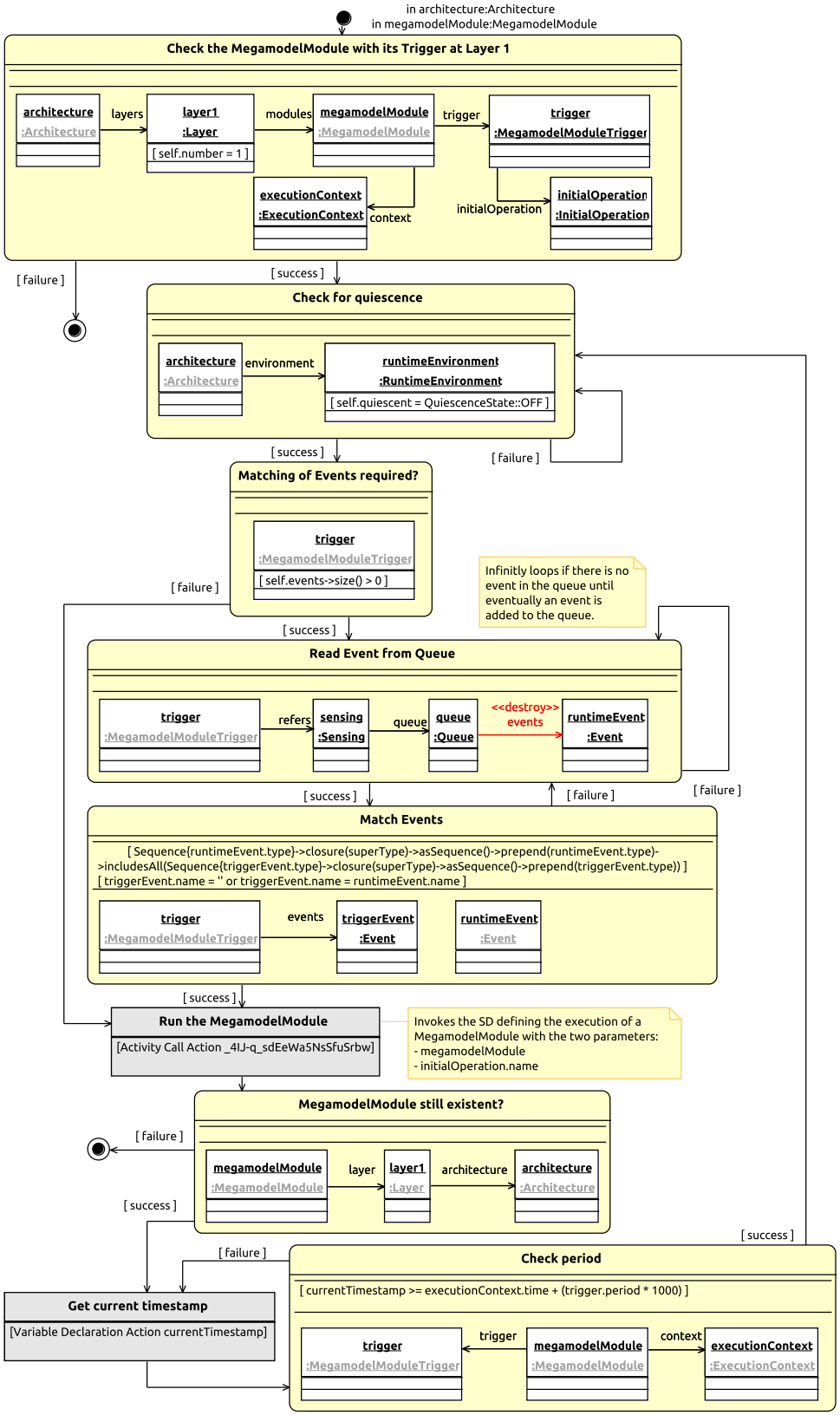


Figure 107: Triggering a Megamodel Module at Layer 1.

The SD has two parameters, the architecture of the overall self-adaptive software and the megamodelModule to be triggered. The first SP checks whether the megamodelModule is

located at layer 1 of the architecture and whether it has a trigger. The SP further obtains the executionContext of the megamodelModule and the initialOperation specified in the trigger. If this check fails, the SD terminates. Otherwise, it starts triggering the megamodelModule.

Therefore, the second SP checks whether the attribute quiescent of the RuntimeEnvironment is set to OFF, which means that the adaptation engine is not in a quiescent state and should not achieve such a state. Otherwise, any triggering of megamodel modules at layer 1 is blocked to achieve or keep quiescence (*cf.* Section 6.6.2). Consequently, the SD would not trigger the module but loop here until quiescence is deactivated (*i. e.*, the attribute quiescent of the RuntimeEnvironment is set to OFF).

If quiescence is deactivated, the third SP checks whether the trigger of the given module specifies any event, that is, whether the trigger is event-based. If it is not, the module can be directly executed (see the action named Run the MegamodelModule that calls the SD specifying the execution of a megamodel module, which will be discussed in Section B.5). Otherwise, a runtime event is required to trigger the module. Therefore, the fourth SP named Read Event from Queue consumes the first runtimeEvent from the FIFO queue. The events relationship from a Queue to the Events is ordered such that always the first event is consumed. If there is no event in the queue, the SP loops infinitely until the adaptable software eventually adds a runtime event to the queue.[3] Having consumed the first runtimeEvent from the queue, the next SP compares in pairs the events specified in the trigger and the runtimeEvent. It checks whether at least one triggerEvent matches the runtimeEvent event based on the types and names of both events. Particularly, the type of the runtimeEvent must be the same or a subtype of the type of the triggerEvent (see first OCL constraint in the SP) as well as the runtimeEvent and triggerEvent must have both the same name if the triggerEvent has a name (see second OCL constraint in the SP). The matching of trigger and runtime events is discussed in more detail in Section 6.4.2. If the events do not match, the next runtime event is consumed from the queue (see SP named Read Event from Queue). Otherwise, the megamodelModule is executed by the Run the MegamodelModule action that invokes the SD specifying the execution of a megamodel module (see Section B.5).

Having executed the megamodel module either with or without an event, the subsequent SP checks whether the executed module still exists in the layered architecture. A megamodel module can be destroyed and thus removed from the architecture if the execution has terminated with a destructive final operation. In this case, the module will not be triggered any more such that the SD terminates. Otherwise, if the module still exists, it can be triggered again. Therefore, the following action obtains the currentTimestamp (in milliseconds). Using this timestamp, the subsequent SP checks with an OCL constraint whether the period (in seconds) specified in the trigger has elapsed after the module has finished its execution at executionContext.time (in milliseconds). The value of executionContext.time is set by the SD that specifies the execution of a module and that has been invoked previously by the action Run the MegamodelModule. If the constraint is false, the control flow goes back to obtain the currentTimestamp and to check again the period. These two steps are repeated until the constraint evaluates to true, that is, until the period has elapsed. If the trigger

3 The behavior of the adaptable software is not in the scope of EUREMA and therefore not specified as part of the EUREMA model or the execution semantics of the EUREMA language. However, to execute the specification of the semantics, an SD such as the one shown in Figure 100a on Page 330 can be used to add events to a queue.

does not specify any period (*i. e.*, the period is zero), the constraints always evaluates to true since the currentTimestamp is always greater than the executionContext.time.[4]

When the period after the execution of the given megamodel module has eventually elapsed, the triggering of the module starts again with the SP named Check for quiescence.

B.4.2 *Triggering Higher-Layer Megamodel Modules*

The second case concerns the triggering of megamodel modules at higher layers of the adaptation engine by intercepting the execution of lower-layer megamodel modules. The interception can happen before or after executing an operation or when executing a transition (*i. e.*, a control flow link) between two operations. This results in three options to trigger a higher-layer megamodel module as defined by the SDs in Figures 108, 109, and 110.

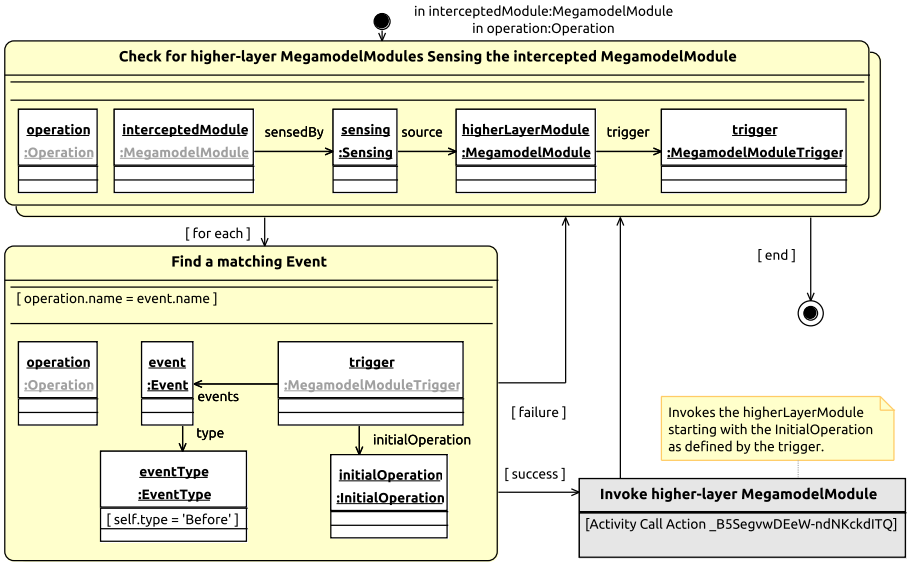


Figure 108: Triggering a Higher-Layer Megamodel Module Before Executing an Operation.

As shown by the first SP of the SD in Figure 108, the execution of a megamodel module, the interceptedModule, has been intercepted *before* having executed the given operation. This pattern searches for all higherLayerModules with a trigger that sense the intercepted module.[5] Here, the explicit sensing relationships come into play, which have been established by an initialization step (see previous discussion in Section B.3). For each of these higher-layer modules, the second SP checks whether the trigger of the module activates, that is, whether at least one of the trigger events and the interception point match. Therefore, the trigger event must be of type Before and the name of the trigger event must be the same as the name of the given operation (see OCL constraints in the eventType element and SP node). If this is the case, the SD defining the execution of a megamodel module (see Section B.5) is invoked by the call action to synchronously execute the higherLayerModule.

4 The fact that a trigger has to specify at least the period or one or more events is assured by the well-formedness of the EUREMA language discussed in Section A.1. Therefore, the execution semantics does not check the validity of a trigger. An invalid trigger does not define a period and does not define any event.

5 The well-formedness of the EUREMA language assures that a megamodel module sensing another module is located at the adjacently higher layer than the sensed module (see Section A.1). Therefore, the structuring of the individual megamodel modules into layers is not checked in the SDs for the execution semantics.

The triggering of a higher-layer module by intercepting a lower-layer module *after* an operation of this lower-layer module has been executed is specified by the SD in Figure 109. This SD is similar to the SD in Figure 108 except that the trigger event must be of type After.

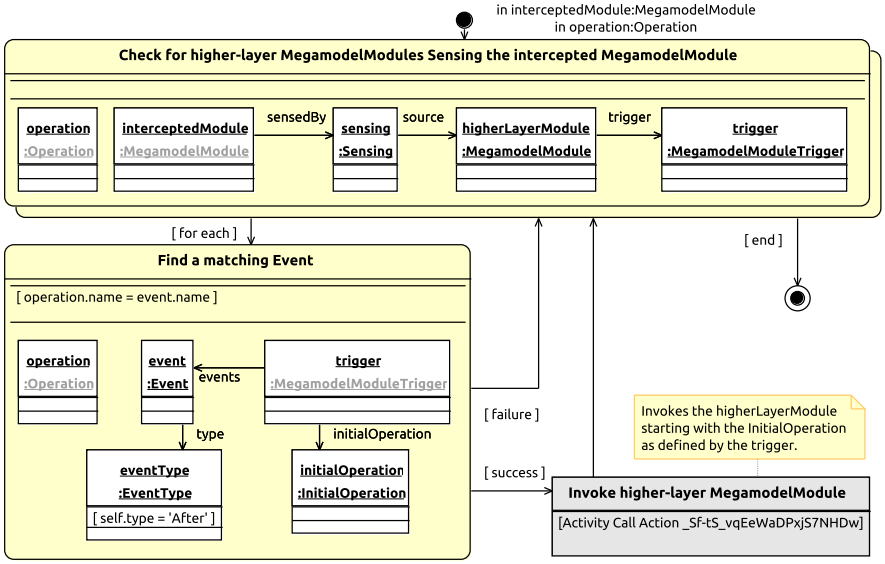


Figure 109: Triggering a Higher-Layer Megamodel Module After Executing an Operation.

Finally, triggering a higher-layer module by intercepting a lower-layer module when executing a transition of this lower-layer module is specified by the SD in Figure 110.

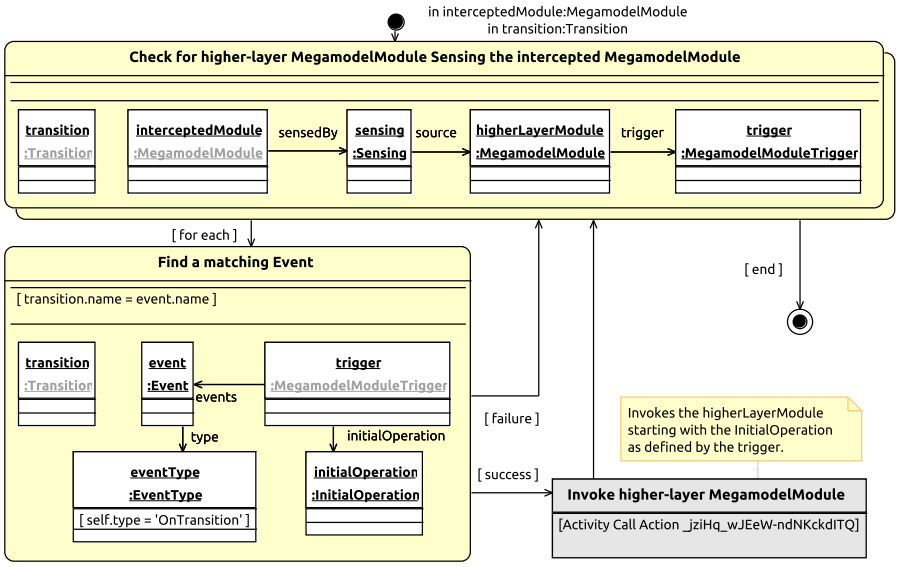


Figure 110: Triggering a Higher-Layer Megamodel Module When Executing a Transition.

In contrast to the SDs in Figures 108 and 109 where an operation is given, here a transition is given whose execution is currently intercepted. The trigger is activated if the trigger event is of type OnTransition and if the name of the trigger event is the same as the name of the intercepted transition (see OCL constraints in the eventType element and SP node). If the trigger is activated, then the call action invokes the SD defining the execution of a megamodel module (see Section B.5) to synchronously execute the higherLayerModule.

These three SDs shown in Figures 108, 109, and 110 are invoked and executed while executing a (lower-layer) megamodel module, which will be discussed in the following section. Particularly, they are invoked when the execution reaches the corresponding interception point, that is, before or after executing an operation or when executing a transition.

B.5 EXECUTION OF A MEGAMODEL MODULE

Having triggered a megamodel module, the module is executed, particularly, the encapsulated Megamodel instance (*i.e.*, the FLD instance) is executed as discussed in Section 6.4.3. The corresponding execution is specified by the SD shown in Figure 111.

The first SP checks whether the given megamodelModule is currently executed, that is, whether its executionContext is among the activeContexts of the runtimeEnvironment. If this is the case, a code snippet is invoked to raise an exception since concurrent executions of the same module are not supported (see the call action named Raise concurrent execution exception). In general, megamodel modules are not re-entrant. Otherwise, the unique execution context of the module is obtained and added to the activeContexts of the runtimeEnvironment by the second SP. The active contexts are those contexts whose modules are currently executed. Hence, the execution keeps track of the currently running modules.

Then the actual execution of the module starts by executing its initial operation (the SD shown in Figure 112 on Page 343 is invoked), followed by alternately executing a transition (the SD shown in Figure 114 on Page 344 is invoked) and an operation (the SD shown in Figure 115 on Page 346 is invoked) until a final operation is reached. If a final operation is reached, the execution of the module has terminated. Therefore, the current relationship (*i.e.*, the *model counter* pointing to the currently executed operation or transition) is removed such that it points to null. Afterwards, the runtime information of the executed module which is maintained by the module's execution context is updated: the count is incremented by one and time is set to the current timestamp. Moreover, the execution context of the executed module is removed from the active contexts of the runtime environment since the module is not running any more. Finally, if the execution of the module has terminated with a destructive final operation (see OCL constraint in the last SP in Figure 111 on the next page), the module and its execution context are destroyed and thus removed from the architecture. Consequently, this module will not be triggered and executed again. Otherwise, the module and its execution context remain in the architecture. In both cases, the SD returns the name of the executed final operation as a return status.

In the following, we discuss the internals of executing a megamodel module, that is, the execution of initial operations, transitions, and the other kinds of operations (*i.e.*, model operations, megamodel calls, decision operations, and final operations).

B.5.1 *Execution of an Initial Operation*

The execution of an initial operation is defined by the SD in Figure 112. This SD is invoked once by the action Execute InitialOperation of the SD executing the megamodel module (see Figure 111 on the next page). As shown in Figure 112, given the executing megamodelModule and the name of the initial operation (initOpName), the first SP checks whether the current relationship (*i.e.*, the model counter) of the module's execution context does not point to any Executable that is either an operation or a transition. This is expressed with a Negative Application Condition (NAC) defining a pattern that must not exist in the model. In this case, there must not be an Executable to which the current relationship points. However, if

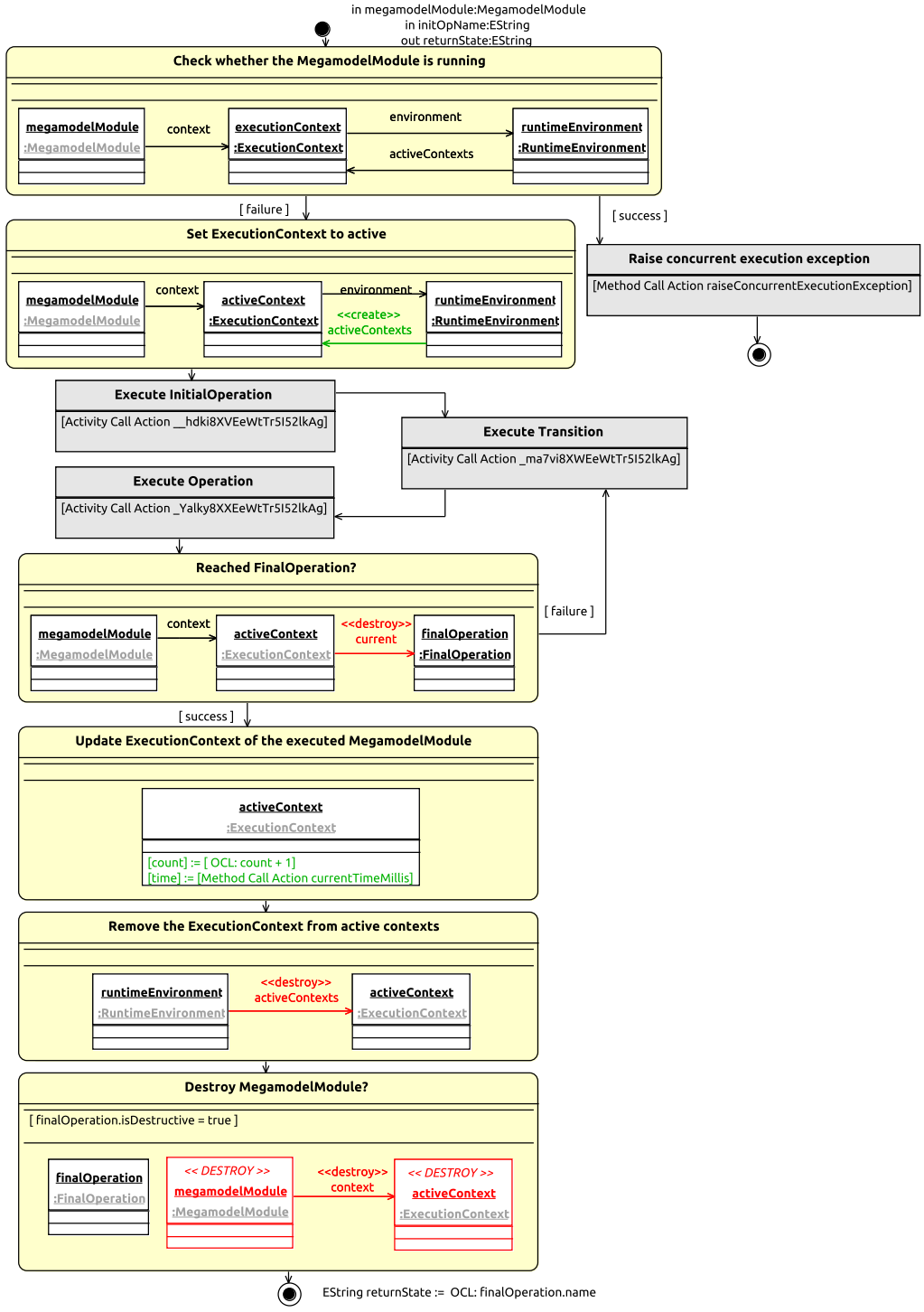


Figure 111: Executing a Megamodel Module.

the NAC exists, a call action invokes a method to raise an exception since the module is already running by currently executing the specific executable. This double-checks that the given megamodelModule is not already running.

Otherwise, the initialOperation to be executed is obtained by the given name (initOpName) and the current relationship now points to this operation (see SP named Find InitialOperation with the given name). The subsequent action calls the SD shown in Figure 108 on Page 339 to check for any triggering of higher-layer megamodel modules *before* executing the initial

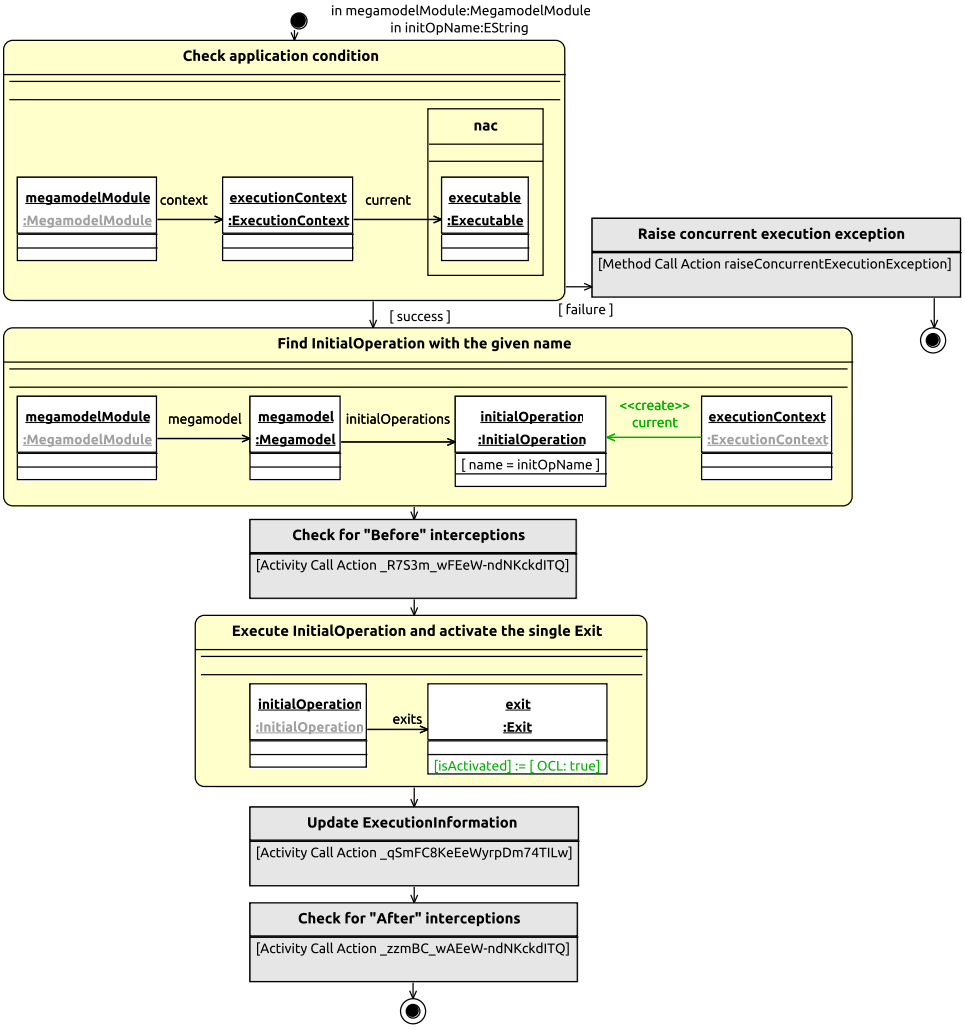


Figure 112: Executing an Initial Operation.

operation. The actual execution of the initial operation is trivial, since it only identifies the unique exit of the operation and sets its boolean flag isActivated to true. This flag is used to execute the subsequent transition, which will be discussed in the following subsection.

Having executed the initial operation, the runtime information of the operation is updated by invoking the SD depicted in Figure 113 on the next page. For the given operation, this SD obtains the unique executionInformation and increments the count by one as well as sets the time to the current timestamp. Finally, an action calls the activity shown in Figure 109 on Page 340 to check for any synchronous triggering of higher-layer megamodel modules *after* executing the initial operation. This action finishes the execution of the initial operation such that the SD depicted in Figure 112 terminates.

B.5.2 *Execution of a Transition*

After the execution of an operation, a transition is going to be executed (*cf.* SD shown in Figure 111 on the previous page). The SD in Figure 114 specifies the execution of a transition.

Particularly, given the currently executed megamodelModule, the first SP obtains the unique executionContext of the module to identify the operation that has just been executed

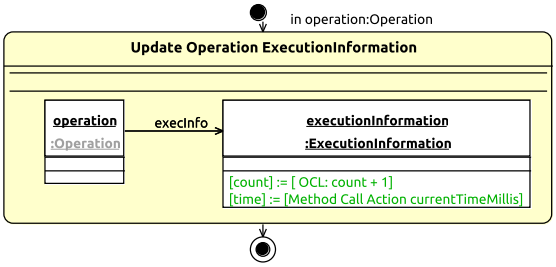


Figure 113: Updating the Runtime Information of an Operation.

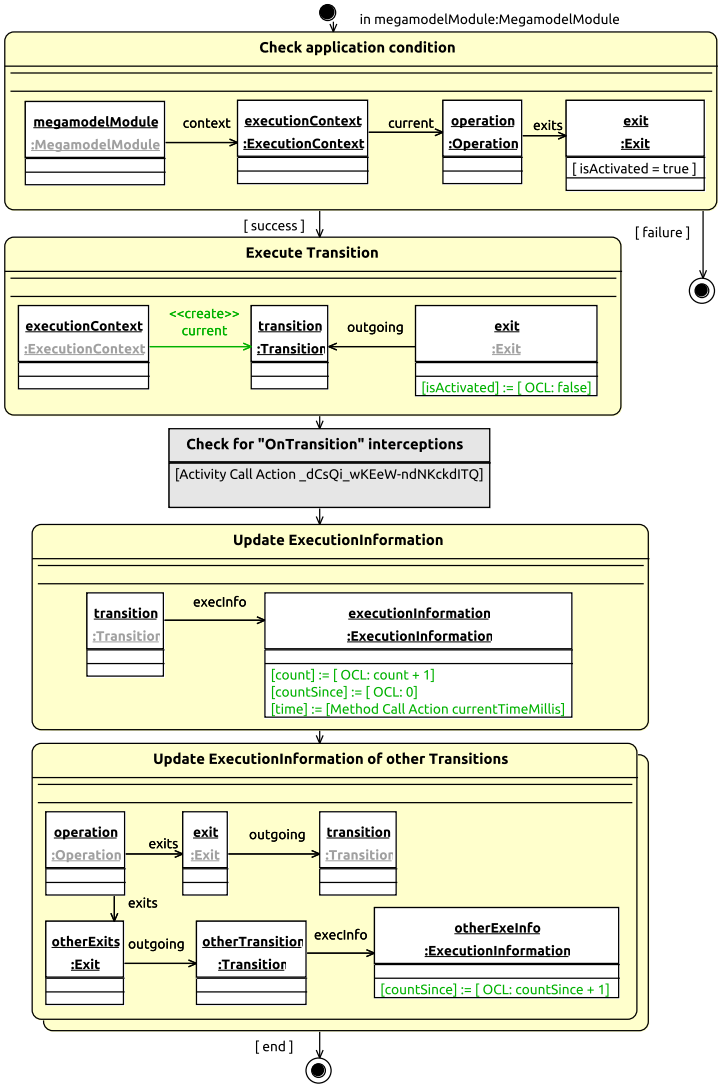


Figure 114: Executing a Transition.

as well as the activated exit of this operation. The execution of an operation results in activating exactly one exit of the operation as defined by the SDs in Figures 112 on the previous page and 115 on Page 346. The second SP obtains the unique outgoing transition of the activated exit, sets the current relationship to this transition, and deactivates again the exit by setting the isActivated attribute to false. Since a transition has no associated behavior, this also executes the transition. Then, an action calls the activity shown in Figure 110 on Page 340 to check for any synchronous triggering of higher-layer megamodel modules when execut-

ing the transition. Afterwards, the runtime information (see executionInformation element) of the executed transition is updated, that is, count is incremented by one, countSince is reset to zero, and time is set to the current timestamp. Finally, the runtime information of all the other outgoing transitions of the same operation that has been executed just before is updated, that is, countSince is incremented by one for each of these otherTransitions.

B.5.3 *Execution of a Model, Megamodel, Decision and Final Operation*

As shown in the SD in Figure 111 on Page 342, after executing a transition, the subsequent operation is executed. Since the initial operation has already been executed (see previous subsection), we consider in the following the execution of model operations, megamodel calls (*i. e.*, complex model operations), decision operations, and final operations. The execution of these kinds of operations is defined by the SD shown in Figure 115.

Given the currently executed megamodelModule (see input parameter of the SD), the first SP obtains the module's unique executionContext to identify the transition that has just been executed. The current relationship of the executionContext points to this transition. The target of the transition is an entry of the operation to be executed now. The second SP obtains this operation and sets the current relationship of the module's executionContext to this operation. *Before* actually executing the operation, the subsequent action calls the SD shown in Figure 108 on Page 339 to check for any synchronous triggering of higher-layer modules.

Afterwards, various SPs are used to identify the specific type of the currently executed operation. At first, the SP named Execute ModelOperation checks whether the operation is of type ModelOperation. If so, the next SP identifies the operationImpl:SofwareModule to which the operation is bound. This software module implements the operation and it is synchronously invoked by the subsequent Run implementation action. Particularly, the action retrieves the concrete implementation class stored in operationImpl.implementation, which is then invoked with the runtime models, that are the input of the operation, as a parameter. This invocation eventually returns the runtime models specified as output of the operation and a returnState. The following SP named Activate Exit uses this returnState to identify the exit of the operation with the same name. This exit is then activated by setting its attribute isActivated to true. The activated exit determines the transition to be executed next.

If the operation is not a ModelOperation, then the SP named Execute MegamodelCall checks whether the operation is of type MegamodelCall (*i. e.*, a complex model operation invoking a megamodel module). If so, the next SP identifies the calledModule:MegamodelModule to which the operation is bound. This megamodel module is then synchronously invoked by the subsequent action named Invoke MegamodelModule to start execution with the initial operation having the same name as the taken entry of the operation. The invocation returns the name of the final operation, in which the execution of the invoked module (*i. e.*, the calledModule) terminates, as the returnState. Similar to the previous case, the returnState determines the exit of the operation to be activated and thus the transition to be executed next (see SP named Activate Exit). If the operation is a ModelOperation or MegamodelCall that is not bound to a SoftwareModule respectively MegamodelModule, the target of the invocation is not defined such that an unbound operation exception is raised (see corresponding action after identifying the software or megamodel module).

If the operation is not a MegamodelCall, then the SP named Execute DecisionOperation checks whether the operation is of type DecisionOperation. If so, the subsequent SP evaluates all conditions expressed with the language and evaluated by the parser discussed in Section A.3. Particularly, the SP obtains *all* (note the cascaded SP node) exits and thus all

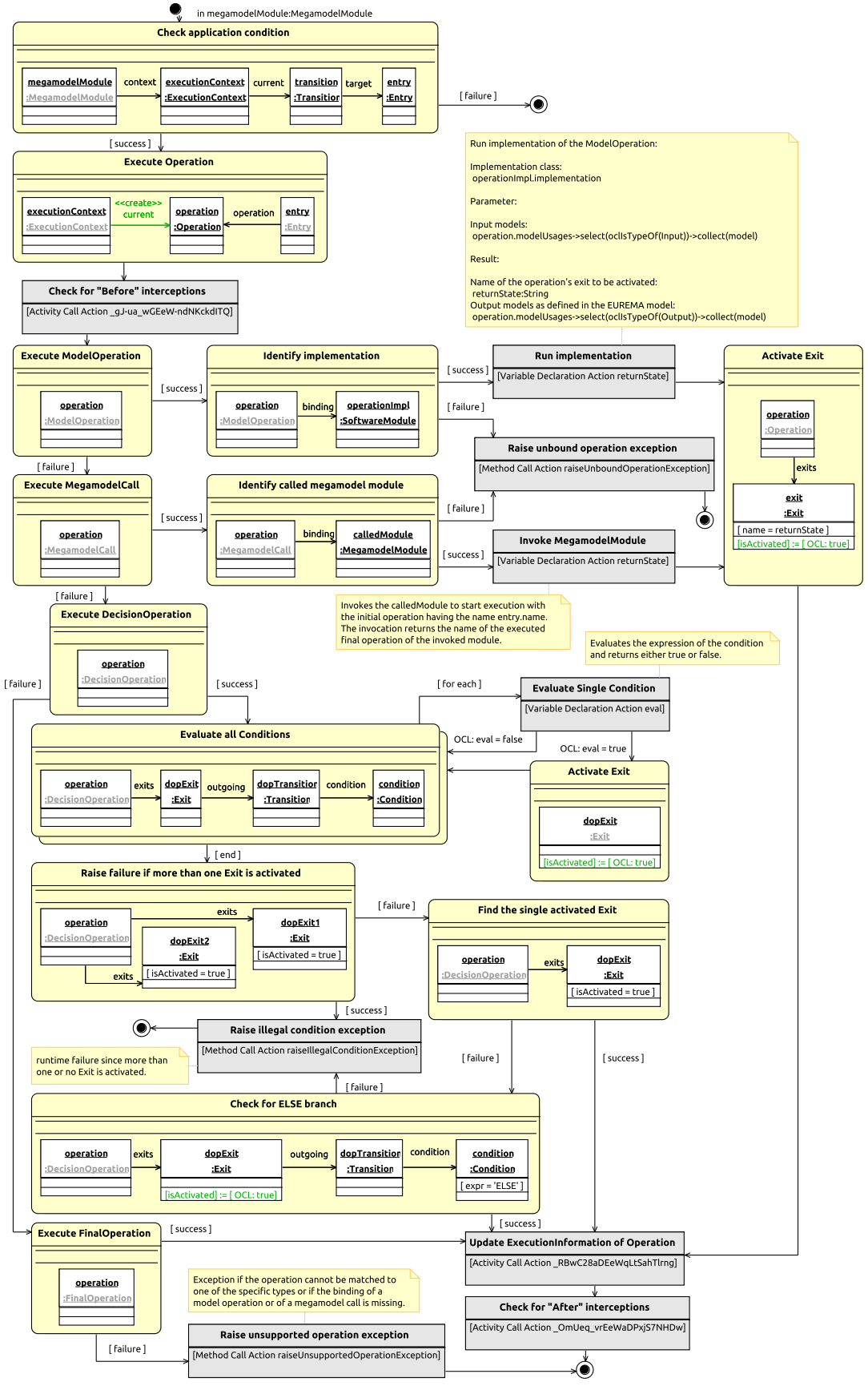


Figure 115: Executing an Operation.

outgoing transitions of the decision operation (see dopExit and dopTransition objects). Each of these transitions is labeled with a condition. All of these conditions are individually evaluated by the action Evaluate Single Condition, which results either to true or false.

If the condition of a transition is evaluated to false, the next condition is obtained and evaluated. If it is evaluated to true, the corresponding exit (dopExit) of the decision operation being the source of the transition is activated (see SP named Activate Exit that sets the attribute isActivated to true). Then, the next condition is obtained and evaluated.

Having evaluated the conditions of all outgoing transitions of the decision operation, a check is made whether more than one exit of the operation is activated (see subsequent SP searching for two activated exits). This leads to an action raising an illegal condition exception since we require that all conditions of one decision operation are disjoint. If two activated exits cannot be found, the single activated exit needs to be identified. If this single activated exit can be found, the decision operation has been successfully executed since we uniquely identified an activated exit and thus a transition to be executed next. Otherwise, no exit has been activated such that the SP named Check for ELSE branch searches for a default transition, that is, a transition whose condition is an ELSE expression (see OCL constraint in the condition object). If there is such a default transition, the corresponding exit is activated (*i.e.*, the attribute isActivated of the dopExit is set to true). Otherwise, an illegal condition exception is raised since all transitions have been evaluated to false and no default transition exists.

If the operation is not a DecisionOperation, then the SP named Execute FinalOperation checks whether the operation is of type FinalOperation. This operation type is the very last possible type. FinalOperations have no associated behavior such that nothing has to be done.

However, if the operation is even not a FinalOperation, an unsupported operation exception is raised. In this case, the type of the operation is not known and thus not supported.

When the operation has been executed—regardless whether it is a ModelOperation, MegamodelCall, DecisionOperation, or FinalOperation—its runtime information is updated. This is done by the action Update ExecutionInformation of Operation that calls the SD shown in Figure 113 on Page 344, which we discussed previously. Finally and *after* the operation has been executed, an action calls the SD shown in Figure 109 on Page 340 to check for any synchronous triggering of higher-layer megamodel modules.

If the operation that has just been executed is a ModelOperation, MegamodelCall, or DecisionOperation, an exit has been activated. This activated exit determines the transition to be executed next. We discussed the execution of a transition in the previous subsection. If the operation is a FinalOperation, an activated exit is not required since a FinalOperation denotes the termination of the execution of the megamodel module.

B.6 QUIESCENCE OF ALL MEGAMODEL MODULES

Finally, we discuss the execution semantics to achieve quiescence of the whole adaptation engine, that is, of all megamodel modules of the layered architecture. This is required for safe adaptation of the LD model and thus of the layered architecture (see Section 6.6.2). In contrast, safe adaptation of individual feedback loops in terms of FLD models does not require quiescence of the whole engine but only of the individual FLD model, which is achieved by intercepting the FLD model under adaptation (see Section 6.6.1). This interception mechanism has been discussed when executing a megamodel module (see Section B.5), during which triggers of higher-layer megamodel modules are executed (see Section B.4).

In general, there is a life cycle of quiescence: First, quiescence is activated, which means that quiescence of all megamodel modules should be achieved. In the second phase, quiescence is achieved by allowing already running megamodel modules to terminate and by blocking the initiations of new executions of megamodel modules at the lowest layer of the adaptation engine. Third, quiescence eventually has been achieved such that safe changes of the adaptation engine are possible. Afterwards, quiescence is deactivated, which releases the blocking of newly initiated executions of the megamodel modules. Hence, the regular execution of the changed adaptation engine resumes.

To activate quiescence of all megamodel modules, the attribute quiescent of the RuntimeEnvironment must be set from OFF to BLOCKING. This is done be the SD shown in Figure 116. As a consequence, the triggering of new executions of megamodel modules at the lowest layer of the adaptation engine is blocked (*cf.* the corresponding SD in Figure 107 on Page 337 that does not trigger any megamodel module at the lowest layer of the adaptation engine when the attribute quiescent is set to BLOCKING or ON).

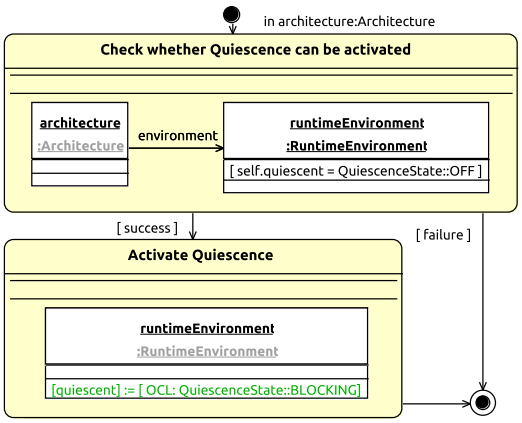


Figure 116: Activating Quiescence of all Megamodel Modules.

However, currently running megamodel modules may continue their execution until they terminate. When all megamodel modules have terminated, a quiescent state has been reached. The RuntimeEnvironment keeps track of the running modules by maintaining a list of all active ExecutionContexts. Each megamodel module has its individual execution context that is active when the module is currently executed, otherwise it is inactive (*cf.* SD in Figure 111 on Page 342 that adds the corresponding execution context to the active contexts just before starting the execution of the megamodel module and that removes this context from the active contexts after the execution of the module). Hence, when all execution contexts are inactive, all megamodel modules and thus the whole adaptation engine are quiescent. This check is specified by the SD shown in Figure 117.

This SD is invoked after quiescence has been activated by the SD in Figure 116 and executed concurrently to the other SDs to continuously perform the check. Given the layered architecture, the first SP obtains the RuntimeEnvironment and performs the check using a NAC stating that there does not exist any activeContext in the activeContexts relationship of the RuntimeEnvironment. In other words, the activeContexts relationship does not point to any execution context. If this is the case, quiescence has been reached and identified such that the attribute quiescent of the RuntimeEnvironment is set from BLOCKING to ON by the second SP. Otherwise, the first SP loops until quiescence has been reached.

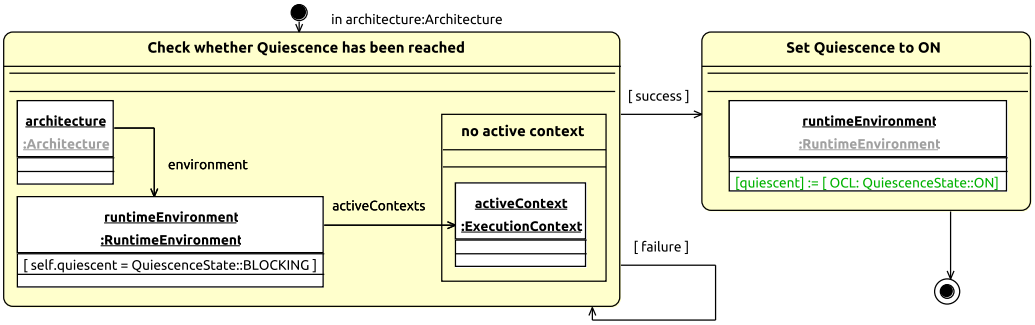


Figure 117: Identifying Quiescence of all Megamodel Modules.

Being in the quiescent state, the LD model and thus the adaptation engine can be safely adapted. Such an adaptation must not modify any element in the EUREMA model that is concerned with the execution as well as the attributes and relationships of these elements. Specifically, these elements are the RuntimeEnvironment, ExecutionContext, ExecutionInformation, and the Queue. The adaptation of the LD model itself is not specified with EUREMA and therefore not covered by the execution semantics discussed here.

After an adaptation of the LD model, the initialization discussed in Appendix B.3 might have to be updated. For instance, if the adaptation adds a megamodel module to the LD model and thus to the adaptation engine, this module requires an ExecutionContext element and each operation and transition of the megamodel encapsulated in the module requires an ExecutionInformation element to enable the execution of the module.

Finally, to resume execution after the adaptation of the LD model, quiescence must be deactivated, which is specified by the SD shown in Figure 118. This SD is invoked after the adaptation of the LD model. It deactivates quiescence by turning the attribute quiescent of the runtimeEnvironment from ON to OFF.

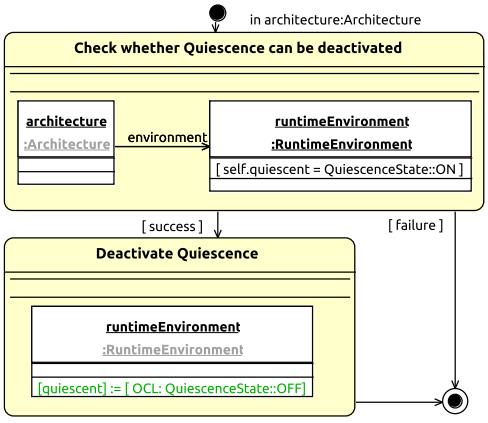


Figure 118: Deactivating Quiescence of all Megamodel Modules.

Having deactivated quiescence, the attribute quiescent of the RuntimeEnvironment is set to OFF such that the triggering of new executions of megamodel modules at the lowest layer of the adaptation engine is not blocked any more (see SD in Figure 107 on Page 337). Consequently, the adapted adaptation engine resumes execution following the semantics discussed in Appendices B.4 and B.5.

C

SUPPLEMENTARY EVALUATION RESULTS

In this appendix, we show the evaluation results of the repeated runs of the experiments that we did not present in Section 9.3.

C.1 SELF-REPAIRING MRUBIS

Consistent results of the remaining four runs of the experiment discussed in Section 9.3.2:

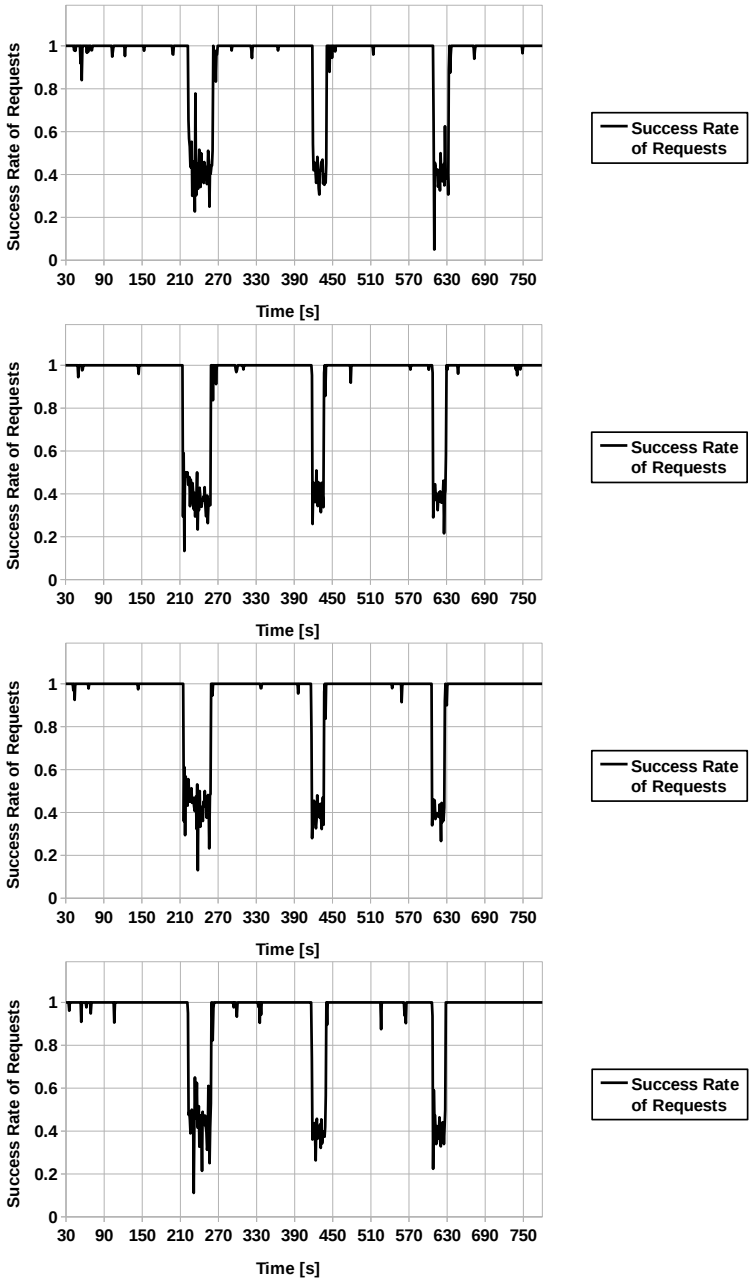


Figure 119: Results of Four Experiment Runs for Self-Repairing mRUBiS.

C.2 SELF-OPTIMIZING MRUBIS

Results of the remaining four runs of the experiment discussed in Section 9.3.3:

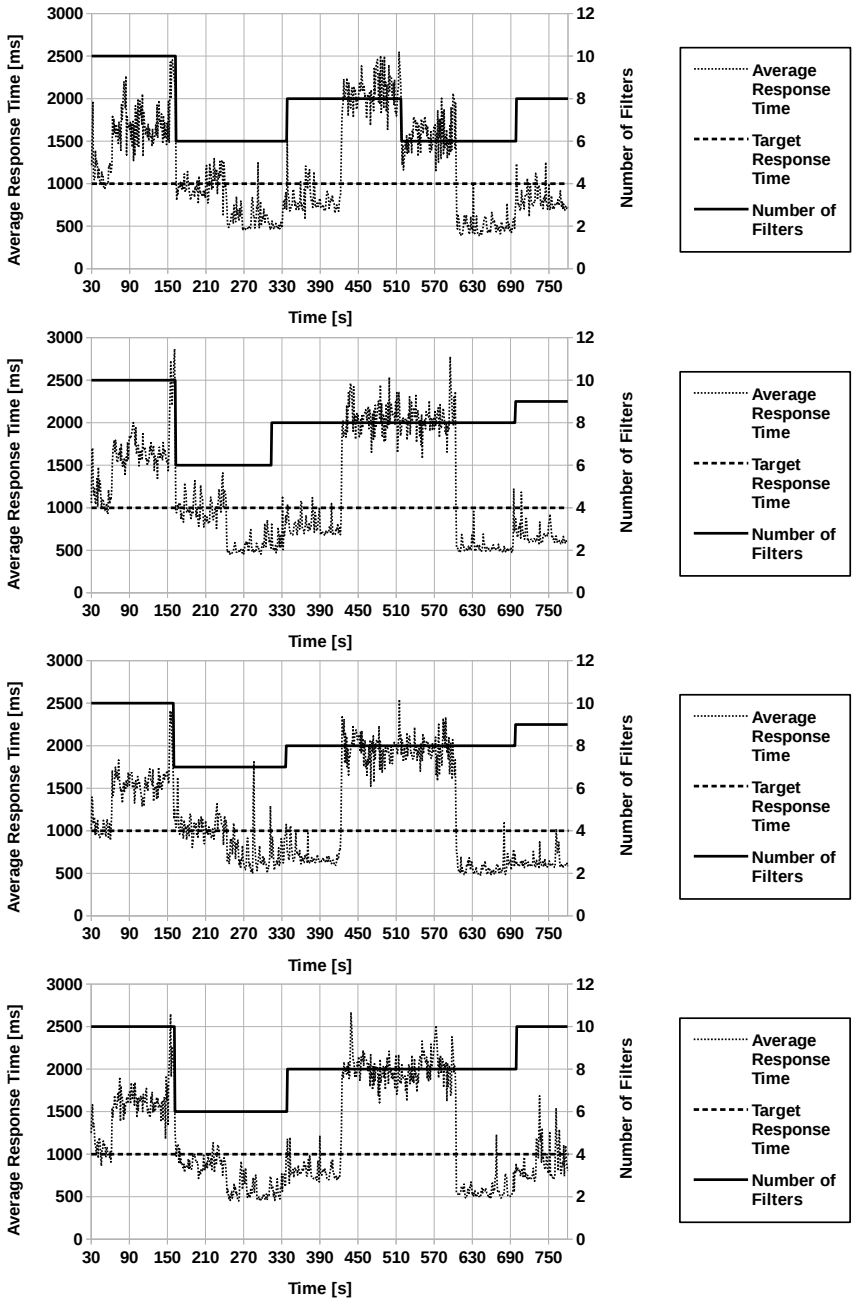


Figure 120: Results of Four Experiment Runs for Self-Optimizing mRUBiS.

The first run of the experiment shown in Figure 120 is completely consistent to the run discussed in Section 9.3.3. Both of these runs are consistent with the last three runs in Figure 120 except of one situation. We briefly discuss this situation in the following by contrasting the first with the last three runs in Figure 120.

In each run of the experiment, the self-optimization feedback loop is executed at around seconds 150, 330, 510, and 690. In contrast to the first run in Figure 120, each of the last three runs shows that the third execution of the feedback loop at around second 510 does

not perform any self-adaptation (*i. e.*, the number of filters is unchanged) despite that the average response time is around 2000 ms and thus above the upper threshold of 1200 ms.

The reason for this is that the average response time shown in Figure 120 are measured by the clients and computed for time frames of 1 s. In contrast, the feedback loop measures the response time on the server side and computes the average response time over the time interval since its last execution, in this case, the interval is from around seconds 330 to 510. In the first half of this interval, the average response time is between 600 and 1000 ms while in the second half, after we add load to the filters at second 420, it is around 2000 ms. Consequently, the feedback loop obtains an average response time for this interval of 1199/1094/1192 ms respectively for the last three runs of the experiment. These values are below the upper threshold of 1200 ms to trigger a self-adaptation for removing filters.

For the feedback loop, the peak of the response time caused by the injection of load at second 420 was not high enough to trigger a self-adaptation and it dissolved as we removed again load at second 600. Thus, the behavior of the feedback loop was not sensitive enough to identify this peak. However, if this peak remains in mRUBiS, the feedback loop will identify and handle it.

This situation shows that it is important to determine appropriate periods for executing a feedback loop, intervals for monitoring the response time, and thresholds to trigger a self-adaptation. For instance, this situation might not occur if either the period is larger (in this case, the feedback loop is executed later such that it observes for a longer time higher response times), the monitoring interval is shorter (in this case, the feedback loop only takes more recent measurements into account, which are higher response times), or the thresholds are more fuzzy (in this case, the feedback loop performs a self-adaptation if the average response time is very close to the upper threshold as in this case).

However, we did not improve the self-optimization by tuning the parameters of the feedback loop such as the period, monitoring interval, or thresholds since the focus of the experiment is on demonstrating the general ability of an EUREMA-based feedback loops to perform self-adaptation. This ability is nevertheless demonstrated by the experiment. Finally, the need of tuning feedback loop parameters is not specific to EUREMA and applies to all approaches for architectural self-optimization.

C.3 COORDINATING THE SELF-REPAIR AND SELF-OPTIMIZATION OF MRUBIS

In this section, we show the evaluation results for coordinating the self-repair and self-optimization feedback loops as discussed in Section 9.3.4.

C.3.1 *Sequencing Complete Feedback Loops*

The remaining evaluation results of the experiment that investigates the coordination of sequencing the feedback loops are shown in Figure 121. These results are generally consistent with the discussion in Section 9.3.4. However, we observe that the self-optimization feedback loop does not always identify a peak of the average response time caused by the injected load while the peak dissolves afterwards when load is removed from mRUBiS. The peak is not strong enough such that the feedback loop would trigger an adaptation. Such situations occur during the last three runs of the experiment in Figure 121 for the execution of the feedback loops at second 540. Particularly, the average response time remains at the same level before and after second 540 when the feedback loops are executed. A detailed discussion of such situations is given in Section C.2.

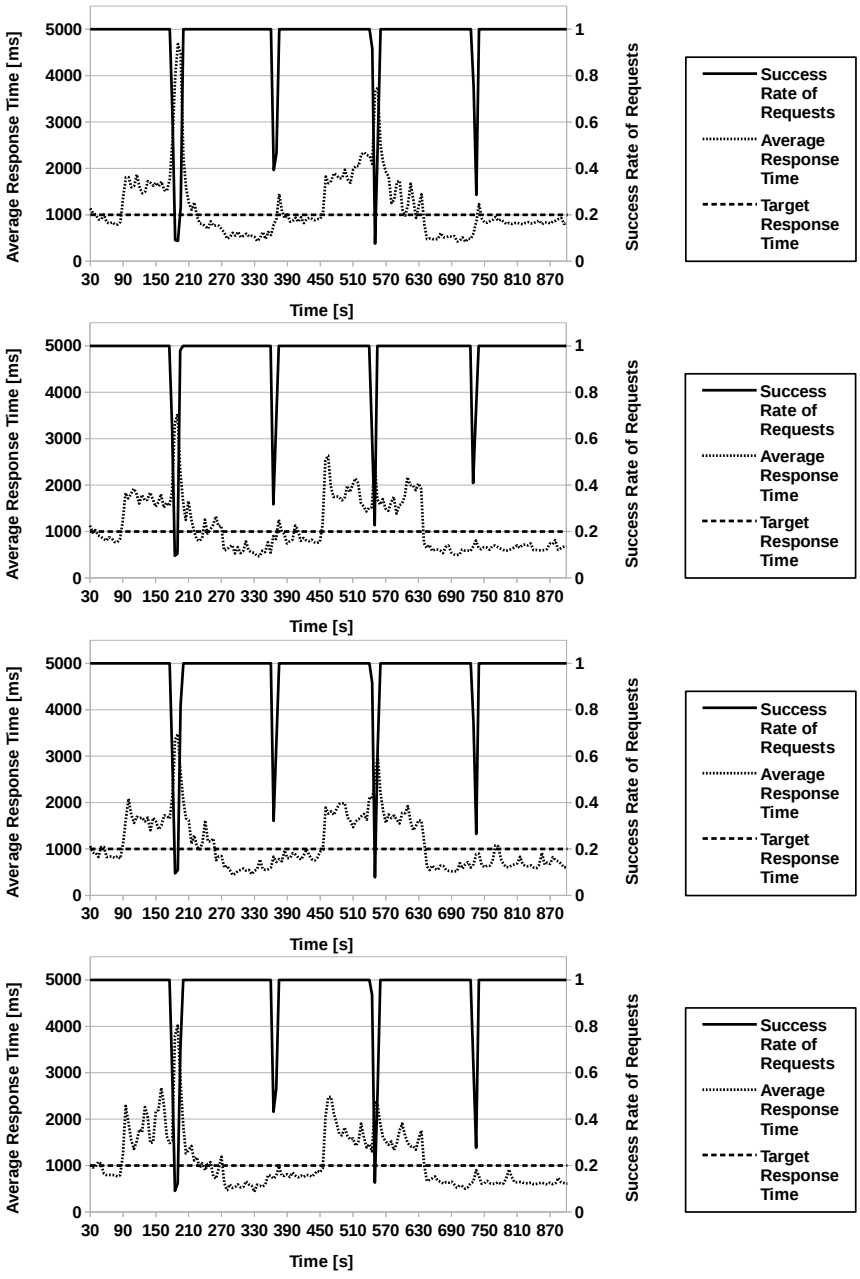


Figure 121: Results of Four Experiment Runs for Sequencing Feedback Loops.

C.3.2 *Sequencing Adaptation Activities of Feedback Loops*

The remaining evaluation results of the experiment that investigates the coordination of sequencing the adaptation activities of feedback loops are shown in Figure 122. These results are consistent to the results of the corresponding experiment discussed in Section 9.3.4.

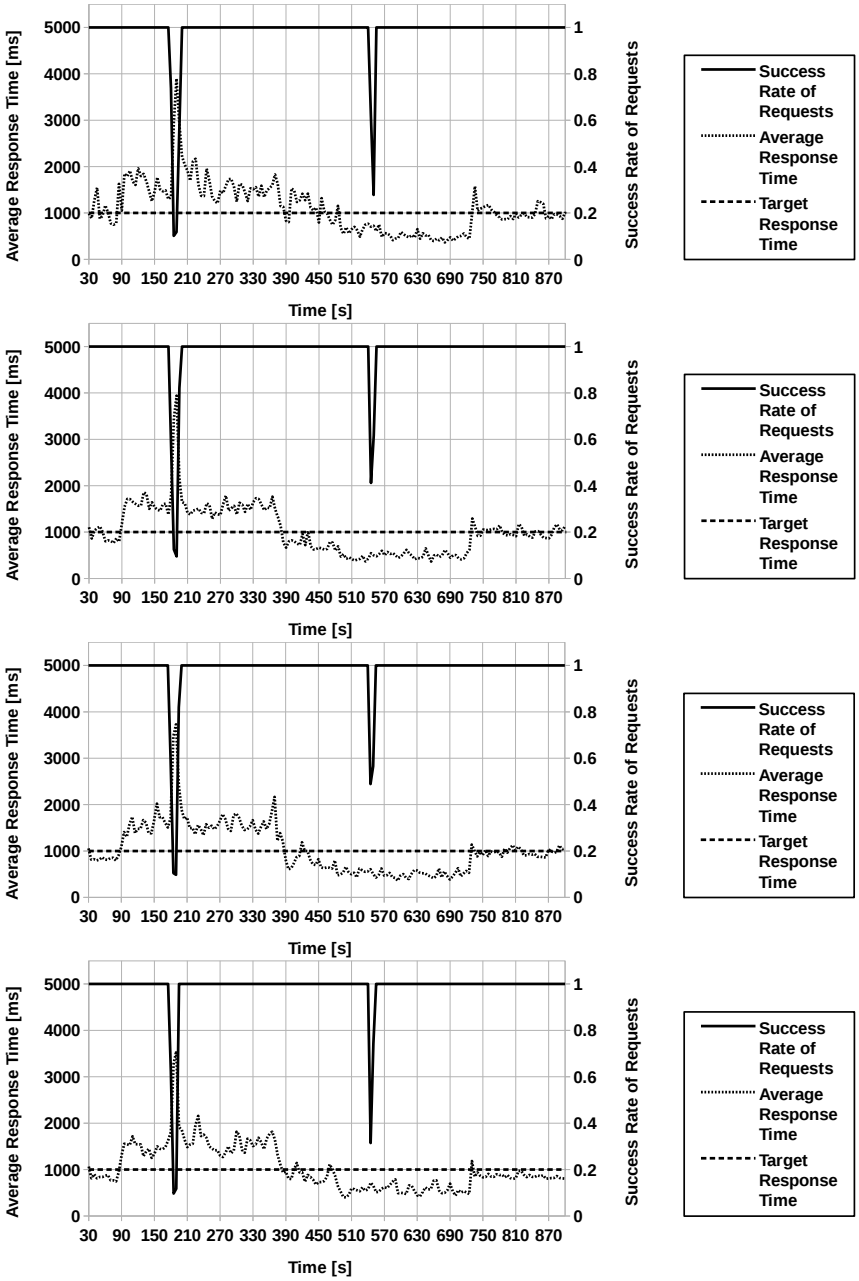


Figure 122: Results of Four Experiment Runs for Sequencing Activities.

C.4 THREE-LAYER ARCHITECTURE FOR SELF-REPAIRING MRUBIS

The remaining evaluation results of the experiment that investigates the three-layer architecture for self-repairing mRUBiS are shown in Figure 123. These results are consistent to the results of the corresponding experiment discussed in Section 9.3.5.

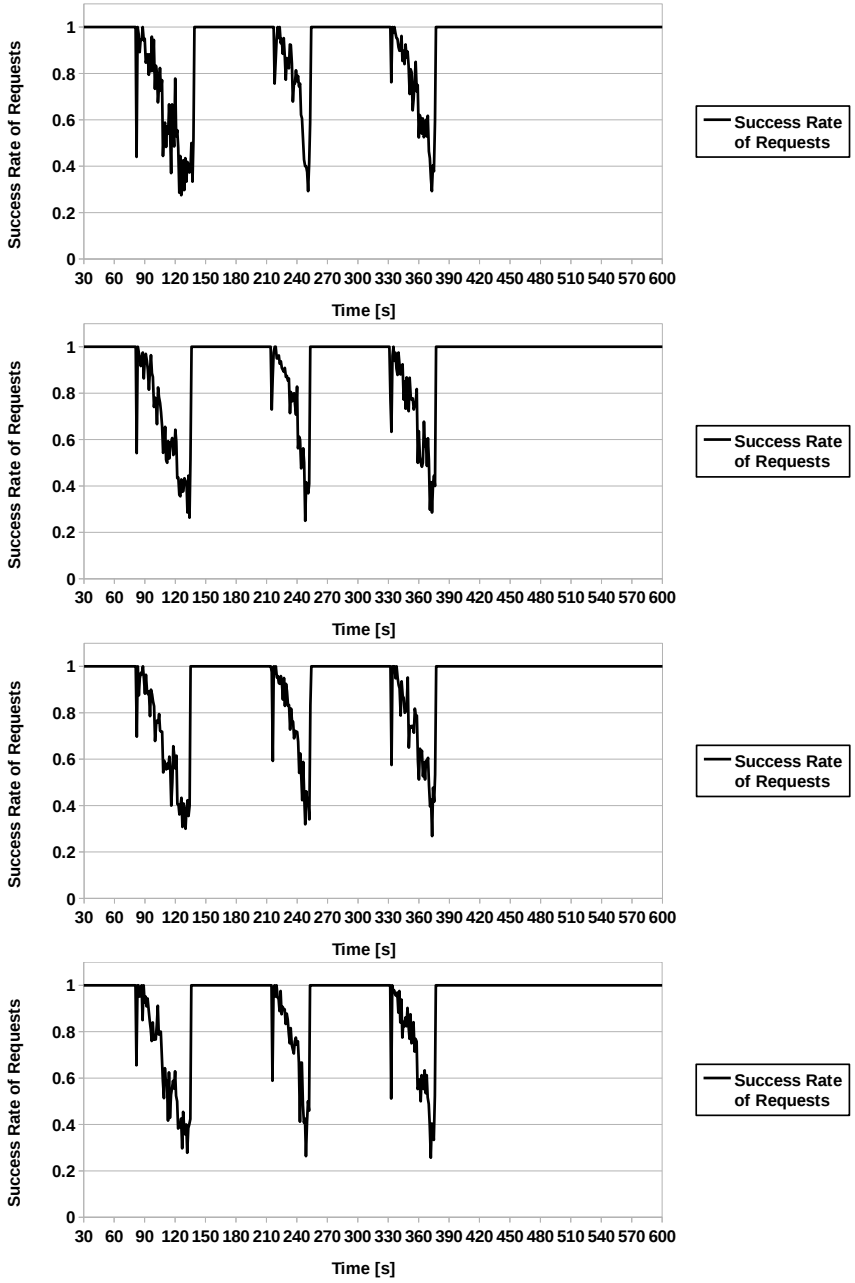


Figure 123: Results of Four Experiment Runs for the Three-Layer Architecture.

D

OVERVIEW OF PUBLICATIONS

This section provides an overview of all publications that have been created in the context of EUREMA and that served as a basis for this thesis.

Publications on EUREMA. EUREMA was motivated by the research on using multiple and abstract runtime models of the adaptable software for self-adaptation [30]. We detailed the use of such runtime models for monitoring [28, 29] and adaptation [22]. Targeting the definition of runtime models as well as the causal connection between these models and the adaptable software, this work serves as a foundation for EUREMA. Therefore, it is realized within the implementation framework (*cf.* Chapter 8) that supports connecting EUREMA-based feedback loops to the adaptable software. This implementation framework further incorporates former work that addresses the instrumentation and code-based adaptation of EJB applications [3, 20]. Finally, this work on using multiple runtime models motivated EUREMA since it requires means to capture at design and run time the runtime models and their interplay to systematically develop and execute feedback loops.

For this purpose, the idea of using *megamodels* at runtime was proposed [31, 32]. Such runtime megamodels became the underlying principles of the EUREMA language. An initial version of EUREMA was presented in [21] and extended and refined in [24, 25] almost to the version as presented in this thesis.

Complementary Publications. In the context of EUREMA, further research has been conducted on runtime models addressing systems-of-systems [13], service management [9], adaptation models in general [23, 27] and utility-driven adaptation in particular [6, 7], the unification of self-adaptation and evolution [26], and the testing feedback loops [14]. Finally and after the submission of this thesis, mRUBiS [18] has been published as an exemplar for developing, evaluating, and comparing self-adaptation solutions to the research community [19].

Additional Publications. Finally, I co-authored publications that address self-adaptive software in general [15, 16, 33] and in particular software engineering processes for developing, operating, and maintaining such software [1], the (de)composition of assurances for such software [17], mechanisms for leveraging runtime models in such software [2] while one mechanism is graph transformations [8], and, the use of control theory for self-adaptation [4, 5]. Finally, I recently co-authored publications on self-aware computing systems [10–12], which are systems that are self-reflective, self-predictive, and self-adaptive.