Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software: Executable Runtime Megamodels

Thomas Vogel, Holger Giese

Technische Berichte Nr. 66

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 66

Thomas Vogel | Holger Giese

Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software

Executable Runtime Megamodels

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6382/
URN urn:nbn:de:kobv:517-opus-63825
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-63825

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-227-8

Abstract

The development of self-adaptive software requires the engineering of an adaptation engine that controls and adapts the underlying adaptable software by means of feedback loops. The adaptation engine often describes the adaptation by using runtime models representing relevant aspects of the adaptable software and particular activities such as analysis and planning that operate on these runtime models. To systematically address the interplay between runtime models and adaptation activities in adaptation engines, *runtime megamodels* have been proposed for self-adaptive software. A runtime megamodel is a specific runtime model whose elements are runtime models and adaptation activities. Thus, a megamodel captures the interplay between multiple models and between models and activities as well as the activation of the activities.

In this article, we go one step further and present a modeling language for *ExecUtable RuntimE MegAmodels* (EUREMA) that considerably eases the development of adaptation engines by following a model-driven engineering approach. We provide a domain-specific modeling language and a runtime interpreter for adaptation engines, in particular for feedback loops. Megamodels are kept explicit and alive at runtime and by interpreting them, they are directly executed to run feedback loops. Additionally, they can be dynamically adjusted to adapt feedback loops. Thus, EUREMA supports development by making feedback loops, their runtime models, and adaptation activities explicit at a higher level of abstraction. Moreover, it enables complex solutions where multiple feedback loops interact or even operate on top of each other. Finally, it leverages the co-existence of self-adaptation and off-line adaptation for evolution.

Keywords Model-Driven Engineering, Modeling Languages, Modeling, Models at Runtime, Megamodels, Model Execution, Self-Adaptive Software, Adaptation Engines, Feedback Loops

Contents

List of Figures

Chapter 1

Introduction

Self-adaptation capabilities are required for many modern software systems that are self-aware, context-aware, mission-critical, or ultra-large-scale in order to dynamically adapt their configuration in response to changes in the system itself, the environment, or the requirements [Cheng et al. 2009; de Lemos et al. 2013].

The development of self-adaptive software following the *external approach* [Salehie and Tahvildari 2009] separates the 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 in response to changing requirements and observed changes in the adaptable software and its operational environment.

This separation eases the development because it decouples the adaptable software from the adaptation engine, and both are integrated by well-defined sensor and effector interfaces. However, the feedback loop then becomes a crucial element of the overall software architecture, which has to be understood and explicitly designed for engineering self-adaptive software [Shaw 1995; Müller et al. 2008; Brun et al. 2009].

Additionally, even *multiple* feedback loops might have to be considered [Kephart and Chess 2003; Brazier et al. 2009; Weyns et al. 2012]. On the one hand, the adaptation engine may not necessarily employ only a single feedback loop but rather multiple of them in parallel to handle different concerns such as self-repair or self-optimization, to distinguish between localized and global adaptation, or to decentralize control in general. On the other hand, there are also cases where the feedback loops have to operate on top of each other as, for example, needed for the different layers of the reference architecture for self-managed systems proposed by Kramer and Magee [2007].

Furthermore, separating the adaptation engine from the adaptable software makes the knowledge about the adaptable software, which is used by the adaptation engine, another crucial element in developing self-adaptive software. This knowledge concerns representations of the running adaptable software, their synchronization with the running adaptable software through sensors and effectors, and the way adaptation is analyzed and planned. Regarding the analysis and planning of adaptation, this includes the strategic knowledge determining how to identify and handle adaptation needs like performance problems.

While traditionally architecture description languages are employed for such representations in self-adaptive software at runtime [Oreizy et al. 1998; Garlan et al. 2004; Georgas et al. 2009], *runtime models* [France and Rumpe 2007; Blair et al. 2009] that follow *model-driven engineering* (MDE) principles and that leverage the benefits of MDE for runtime abstractions of the adaptable software are emerging today [Morin et al. 2009a; Vogel and Giese 2010; Song et al. 2011]. It is further likely that the adaptation engine does not only employ a single runtime model but rather multiple and specialized models at the same time to handle different concerns such as failures or performance. Likewise, Blair et al. [2009, p.25] have observed "that in practice, it is likely that multiple [runtime] models will coexist and that different styles of models may be required to capture different system concerns." This makes it necessary to simultaneously consider multiple runtime models and the interplay between them when engineering and executing adaptation engines [Vogel et al. 2011].

Finally, it cannot be expected that the self-adaptive software automates and takes over all the adaptation activities that are usually performed off-line in the context of maintenance and evolution. Thus, besides realizing an adaptation engine for the self-adaptation, a solution for the co-existence of such an engine with off-line adaptation, and thus, with typical maintenance and evolution is required [Gacek et al. 2008; Andersson et al. 2013].

All these aspects constitute core requirements for self-adaptive software, which have to be considered when engineering adaptation engines.

1.1 State-of-the-Art in Engineering Adaptation Engines

In the following, we review state-of-the-art approaches and show that they do not simultaneously support all the core requirements for engineering adaptation engines as just outlined, namely making feedback loops explicit in the design of self-adaptive software, and supporting multiple feedback loops, layered architectures for feedback loops, and the co-existence of self-adaptation and off-line adaptation, while exploiting runtime models and MDE.

There exists a large body of work on feedback loops to control systems. In particular, *autonomic computing* has achieved results by applying concepts of control theory to runtime parameter adaptation of software systems [Kokar et al. 1999; Hellerstein et al. 2004]. However, self-adaptation oftentimes considers dynamic software architectures in addition to parameters [McKinley et al. 2004], which prevents a direct application of control theory concepts and requires new means for engineering adaptation engines.

A popular way to engineer self-adaptive software are framework-based approaches that use some form of models (cf. [Salehie and Tahvildari 2009]). For example, frameworks employ models to specify self-adaptive software including the adaptation as mappings of assertions to adaptation actions [Schmidt et al. 2008] or as transitions between configurations of the adaptable software [Bencomo and Blair 2009]. These models are used for generating partial code for adaptation engines to simplify their development. The structure of the resulting engines supporting single feedback loops are static and pre-defined by these frameworks. Moreover, the created models do not make the feedback loop explicit and they are not kept alive at runtime, for example, to execute and dynamically adjust the adaptation engine.

In contrast, frameworks like *Rainbow* [Garlan et al. 2004], *MADAM* [Floch et al. 2006], *MUSIC* [Rouvoy et al. 2009], *DiVA* [Morin et al. 2008, 2009a,b], or *GRAF* [Amoui et al. 2012] maintain runtime models that specify the adaptation and capture the knowledge used by the feedback loops. These models can be modified at runtime by engineers, especially to replace adaptation strategies to adjust the adaptation logic. However, support for dynamically adjusting a feedback loop is limited since these frameworks support only single feedback loops, whose structuring of adaptation activities cannot be adjusted at runtime in contrast to specific models, like adaptation strategies, consumed by the activities. Additionally, the runtime models do not explicitly specify complete feedback loops because each of these frameworks prescribe a single feedback loop and just offers customization points, like to inject adaptation strategies. This is motivated by their focus to reduce development efforts for adaptation engines at the expense of limited flexibility. Thus, when developing a specific self-adaptive software, these frameworks do not support feedback loops that are entirely and individually designed by engineers for the specific case.

All the approaches discussed so far support adaptation engines with single feedback loops and do not address multiple, interacting feedback loops. Kephart et al. [2007] consider interactions between two feedback loops that manage competing concerns (energy consumption vs. performance). For this specific case, they propose a coordination solution based on utility functions that has been "established through trial and error" [Kephart et al. 2007, p.24]. In contrast, a generic synchronization protocol for multiple feedback loops is presented by de Oliveira et al. [2012], which supports mutual exclusive access to knowledge and the asynchronous triggering among feedback loops. However, the protocol is restricted since a feedback loop may only trigger another loop from the execute activity but not from the monitor, analyze, or plan activities. Thus, directly coordinating, e.g., the analysis activities of multiple feedback loops is not supported. In [Gueye et al. 2012], the coordination between multiple feedback loops is realized by a distinct controller that decides which feedback loop may exclusively perform an adaptation based on the states of the other loops. This decision is exactly specified by automata models describing the states of the feedback loops and by a coordination policy, which are used for generating the controller. However, these models do not specify the feedback loops and their coordination at the architectural level of self-adaptive software. Other approaches addressing multiple feedback loops are implementation frameworks that aim at reducing development efforts without prescribing a specific solution for the interaction or coordination. Vromant et al. [2011] provide reusable components that support the distributed communication among multiple feedback loops or adaptation activities. Cheng et al. [2004] provide an abstraction layer between the adaptable software and multiple feedback loops. Through this layer, all feedback loops have consistent access and knowledge about the adaptable software. For a case study, they apply a specific coordination solution for two feedback loops ensuring that only one loop is active at any time.

All of the approaches discussed so far, either addressing single or multiple feedback loops, either provide specific and pre-defined solutions or generic implementation support, which results in adaptation engines whose structure cannot be dynamically adapted at runtime.

Dynamically adapting feedback loops is addressed by approaches adopting layered architectures, where a higher-layer feedback loop adjusts the feedback loop at the layer below. In our previous work on *Mechatronic UML* [Burmester et al. 2004, 2008] for the model-driven development of self-optimizing mechatronic systems, we extended UML to specify and generate a hierarchical self-adaptation scheme that addresses control, hard real-time reconfiguration, and soft real-time planning by distinct feedback loops at different layers [Hestermeyer et al. 2004]. However, the adaptation is defined before deployment and cannot be evolved through off-line adaptation at runtime, among others, since the models are not kept alive at runtime. Sykes et al. [2008] and Heaven et al. [2009] propose a three-layer architecture that distinguishes between component-based control, architectural (re)configuration, and high-level task (re)planning. Plans generated by the highest layer are executed by the middle layer that generates new configurations for the lowest layer. However, the current solution focuses on synthesizing initial plans before the system is started, but it does not support task replanning at runtime. Thus, the highest-layer feedback loop adapting the middle layer has not been realized. In contrast, *PLASMA* [Tajalli et al. 2010] supports replanning and adapting the middle layer in a similar, layered architecture. However, the extent of this adaptation is not clear since the architecture of the middle layer is pre-defined by engineers. Moreover, the focus of *PLASMA* is to provide a framework that automates the generation and enactment of plans while the employed feedback loops, their adaptation activities, and knowledge are not explicitly modeled for all layers. Finally, the number of layers (three) and the number of feedback loops for each layer (one) seem to be immutable in these approaches [Sykes et al. 2008; Heaven et al. 2009; Tajalli et al. 2010]. Thus, multiple feedback loops for a layer, or (dynamically) changing the number of layers and feedback loops are not supported.

Changing the number of layers and feedback loops can be seen as an extensive adaptation evolving the self-adaptive software. Besides adapting itself, self-adaptive software has to be open for off-line adaptation by means of maintenance and evolution [Andersson et al. 2013]. Gacek et al. [2008] discuss the idea of having two intertwined feedback loops for self-adaptation and off-line adaptation, but they do not present a solution realizing this idea. As discussed above, frameworks utilizing runtime models often support to change those models, e.g., to add or remove adaptation strategies at runtime. In this context, though focusing on self-adaptation, Morin et al. [2009a] claim to support evolution as changes performed manually on runtime models, which is not substantiated to an integrated co-existence that properly executes those changes to the running system. In this context, they propose an initial step in [Morin et al. 2009c]: an engineer changes models in the development environment, while the same kind of models are also used by the adaptation engine at runtime. Assuming there is no self-adaptation in progress, these changes are executed to the running system only if the system fulfills constraints defined by the engineer. However, changes that affect the structure of the adaptation engine's feedback loop, the number of feedback loops, or the number of layers are not considered.

All approaches discussed so far do not make the feedback loops, their adaptation activities and (runtime) models, and their interactions or coordination explicit in the architectural design of self-adaptive software. A few approaches exist that address the explicit modeling of feedback loops when designing self-adaptive software. In [Hebig et al. 2010], we proposed a UML profile to make feedback loops and the interplay of multiple feedback loops explicit in architectural design and analysis using UML models. A feedback loop is modeled at the abstraction level of controllers, thus abstracting from individual adaptation activities and runtime models. In contrast, Weyns et al. [2012] present a formal reference model for self-adaptive systems that supports the description of feedback loops including adaptation activities and runtime models used by the activities. The goal of the reference model is to support the systematic engineering of self-adaptive systems by providing a means to formally describe and evaluate design alternatives early in the development process. The models created by both approaches [Hebig et al. 2010; Weyns et al. 2012] are used for the architectural design when developing self-adaptive software, but they are not used at runtime for operating or adapting the software.

Summing up, state-of-the-art approaches for engineering self-adaptive software aim at reducing development efforts by generating adaptation engines or providing reusable frameworks for the engines. The resulting adaptation engines often consist of single feedback loops, whose structure is rather static and pre-defined by the frameworks. This limits their adaptation either during development, dynamically in layered architectures at runtime, or through evolution (off-line adaptation). In general, there exist only preliminary work on layered architectures for self-adaptive software and on the co-existence of self-adaptation and off-line adaptation. Moreover, approaches providing runtime support for self-adaptive software do not address the explicit modeling of the feedback loops or their adaptation activities and knowledge. Thus, even when multiple runtime models are used for the knowledge within a feedback loop, these models and their interplay are not explicitly addressed during design and operation of the self-adaptive software. State-of-the-art frameworks do not consider runtime models, like runtime megamodels, that describe whole feedback loops and leverage the execution and adaptation of feedback loops. In contrast, approaches tackling the explicit modeling of feedback loops are focused on the design of self-adaptive software and they do not provide any runtime support that leverage those models at runtime for executing or adjusting the software. Thus, state-of-the-art approaches do not simultaneously support all the core requirements for engineering adaptation engines as previously outlined.

1.2 Approach: EUREMA – ExecUtable RuntimE MegAmodels

For self-adaptive software that is driven by runtime models, we proposed in [Vogel et al. 2011] the generic idea to utilize *runtime megamodels* that have runtime models as their elements and that describe the processing of these models by adaptation activities as model operations.[1] In this article, we go one step further and present a complete MDE approach called *ExecUtable RuntimE MegAmodels* (EUREMA) that enables the specification and execution of complex adaptation engines for self-adaptive software directly supporting feedback loops and runtime models. The EUREMA language considerably eases the development of adaptation engines by supporting a domain-specific modeling solution and the EUREMA runtime interpreter supports the execution of the adaptation engines, in particular the feedback loops. Moreover, EUREMA explicitly maintains the different runtime models, the interplay between these models, and the model operations performing adaptation activities and working on these models. Thus, the maintenance of runtime models and model operations continues beyond the initial development-time of the software.

The proposed EUREMA modeling language is specific for the development of adaptation engines for self-adaptive software and is based on general modeling concepts for structural and behavioral diagrams. Therefore, EUREMA supports two types of diagrams to specify and describe adaptation engines: A behavioral *feedback loop diagram* (FLD) is used to model a complete feedback loop or individual adaptation activities and runtime models of a feedback loop. An FLD is considered as a *megamodel module* that encapsulates the details of a feedback loop or adaptation activities. A structural *layer diagram* (LD) describes how the different megamodel modules and the adaptable software are related to each other in a concrete situation of the self-adaptive software. Thus, an LD provides an abstract and complete view of an instance of the self-adaptive software by reflecting its architecture. This architectural view considers feedback loops as black boxes encapsulated in megamodel modules, while white-box views on megamodel modules are provided by FLDs.

Thus, EUREMA models specify feedback loops and their structuring in adaptation engines. Thereby, the models make the feedback loops explicit in the architectural design of the self-adaptive software and they are kept alive at runtime and executed by an interpreter, which supports the design and the execution of adaptation engines.

1.3 Contribution

This article discusses the EUREMA approach with the following major contributions: (a) we thoroughly discuss requirements for adaptation engines and feedback loops driven by runtime models, which has influenced the design of the EUREMA language, (b) using the EUREMA language, feedback loops are explicitly specified at a higher level of abstraction by capturing the interplay of adaptation activities and runtime models as well as the interactions between multiple feedback loops, (c) the knowledge used within a feedback loop is refined to multiple runtime models that are captured by EUREMA models, (d) EUREMA models specifying feedback loops are kept alive at runtime, which leverages layered architectures for dynamically adjusting the feedback loops, (e) the co-existence of self-adaptation with off-line adaptation is introduced to support evolution of the self-adaptive software by dynamic layers, and (f) we evaluated EUREMA by discussing how EUREMA addresses the requirements, by modeling three approaches to self-adaptive software from the literature, and by quantifying the runtime performance of the interpreter for EUREMA models.

This article is a revised and extended version of [Vogel and Giese 2012a] that introduced the basic concepts of the EUREMA language and therefore initially addressed the contributions (b), (c), and (d). In contrast, this article refines these contributions by extending the language with triggers for feedback loops and with layer diagrams (LDs) providing architectural views of the self-adaptive software. Finally,

[1] In the research field of MDE and in particular of model management for model-driven software development, a *megamodel* refers to a model that has models as its elements and that captures the relationships between the contained models by means of model operations, like model transformations (cf. [Barbero et al. 2007; Bézivin et al. 2003, 2004; Favre 2005]).

this article presents the novel contributions (a), (e), and (f) that have not been discussed in the initial paper.

In contrast to the state-of-the-art in engineering self-adaptive software, we propose a seamless model-driven engineering approach. The modeling language supports the design as well as the execution of feedback loops in adaptation engines. Thereby, EUREMA improves 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 or layers in layered architectures. In contrast to existing modeling languages for self-adaptive software, EUREMA provides improvements by keeping the models alive at runtime for executing and adjusting feedback loops either dynamically or by off-line adaptation.

1.4 Outline

The rest of the article is structured as follows. The next section discusses in detail the terminology, the basic concepts, and the core requirements for self-adaptive software. Then, we introduce the basic EUREMA concepts for modeling single and multiple feedback loops in Sections 3 and 4, respectively. These concepts are refined for layered architectures of adaptation engines in Section 5 and for the co-existence of self-adaptation and off-line adaptation in Section 6. We discuss the execution of EUREMA models in Section 7 and evaluate EUREMA in Section 8. Finally, the article concludes and outlines future work.

Chapter 2

Terminology, Concepts, and Requirements

In this section, we clarify terminology, introduce relevant core concepts of self-adaptive software, and refine the core requirements for engineering self-adaptive software as outlined in the introduction (Section 1).

The *external (architectural) approach* is typically adopted in self-adaptive software [Salehie and Tahvildari 2009]. Thus, this article considers this approach as depicted in Figure 2.1. It 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 adaptation engine implements the *adaptation logic* as a feedback loop, which constitutes self-adaptation.

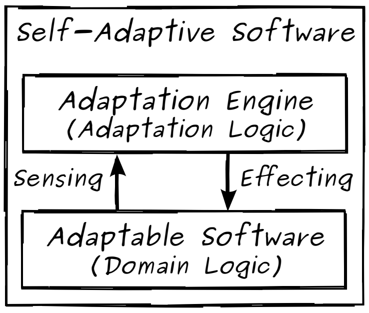


Fig. 2.1: External approach

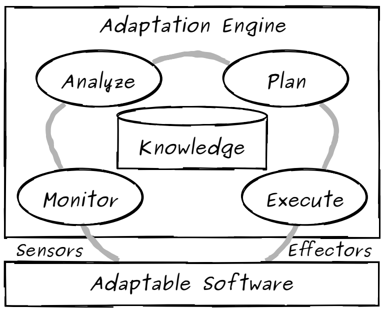


Fig. 2.2: MAPE-K

Thus, the engineering of adaptation engines and feedback loops is essential for the external approach to self-adaptive software. This requires a modeling language to design and specify adaptation engines and related techniques to support the implementation and execution. In the following, we refine the general requirements for self-adaptive software as discussed in the introduction (Section 1) for the particular case of a modeling language for specifying, implementing, and executing adaptation engines. Therefore, we will discuss requirements with respect to feedback loops and their explicit specification, runtime models used as the knowledge within feedback loops, sensors and effectors, layered architectures for adaptation engines, and the co-existence of off-line adaptation and self-adaptation.

2.1 Feedback Loops

The separation between the adaptation engine and the adaptable software makes the *feedback loop* between the adaptable software and the adaptation engine a crucial element of the overall architecture, which has to be made explicit in the design and analysis of self-adaptive software (cf. [Shaw 1995; Kokar et al. 1999; Hellerstein et al. 2004; Müller et al. 2008; Brun et al. 2009]). Thus, feedback loops have to be explicitly modeled.

A more detailed view of the external approach and the feedback loop between the adaptable software and the adaptation engine is provided by the *MAPE-K* cycle (**M**onitor/**A**nalyze/**P**lan/**E**xecute-**K**nowledge) as proposed by Kephart and Chess [2003] and depicted in Figure 2.2. The adaptation engine (called *Autonomic Manager* in [Kephart and Chess 2003]) is refined to four adaptation activities sharing a common knowledge base. The adaptable software (called *Managed System* in [Kephart and Chess 2003]) is *monitored* and *analyzed*, and if changes are required, adaptation is *planned* and *executed* to this software.

As sketched in Figure 2.2, the modeling language should support the specification of adaptation activities that form a feedback loop. This includes the coordination of activities within one feedback loop, which is called *intra-loop coordination* by Vromant et al. [2011], by means of the control flow for these activities. By explicitly specifying adaptation activities, the control flow makes the ordering and dependencies between individual activities explicit and it enables the well-defined coordination and execution of a whole feedback loop.

Moreover, it has to be specified *when* a feedback loop should be executed, i.e., the language should capture *triggering conditions* for initiating the execution of feedback loops. Examples are time or event-based triggers as well as combinations of them.

Additionally, even multiple feedback loops might have to be considered [Kephart and Chess 2003; Brazier et al. 2009; Weyns et al. 2012]. On the one hand, the adaptation engine may not necessarily employ only a single feedback loop but rather multiple of them in parallel to handle different concerns such as self-repair or self-optimization [Kephart et al. 2007; Vogel et al. 2009, 2010; Vogel and Giese 2010; Frey et al. 2012], to distinguish between localized and global adaptation [Cheng et al. 2004; de Oliveira et al. 2012; Gueye et al. 2012], or to decentralize control in general [Vromant et al. 2011; Weyns et al. 2013]. The language should therefore support the modeling of multiple, interacting, and potentially distributed feedback loops and their coordination. Thus, as sketched in Figure 2.3, besides modeling multiple feedback loops also means for *inter-loop coordination* (cf. [Vromant et al. 2011]) and *distribution* are required.

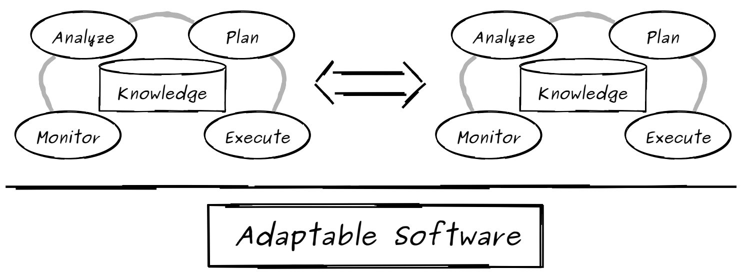


Fig. 2.3: Multiple feedback loops and inter-loop coordination

Furthermore, besides specifying feedback loops, the language should support the *execution* of feedback loops based on the specifications. This includes *concurrency* to simultaneously execute multiple feedback loops or the *incremental* execution of adaptation activities within a feedback loop. Moreover, the language support for execution also has to cover the other cases, like reflection, that are discussed later.

2.2 Knowledge & Runtime Models

In the MAPE-K cycle, the adaptation activities are the computations performed for self-adaptation while the knowledge base provides the information required for these computations. It has been observed by Weyns et al. [2012, p.8:56] 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 the explicit treatment of the knowledge part that is substantiated in our case to a set of specific kinds of runtime models to leverage benefits of MDE for runtime adaptation (cf. [France and Rumpe 2007; Blair et al. 2009]). Blair et al. [2009, 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." Since this definition is focused on runtime models that reflect the adaptable software, we take a broader perspective on runtime models by considering all models that are used on-line by any adaptation activity of a feedback loop. Thus, analysis rules and adaptation strategies are examples for further runtime models. Based on a comprehensive literature review, we proposed a categorization of runtime models in [Vogel et al. 2011], which is depicted in an extended version in Figure 2.4.

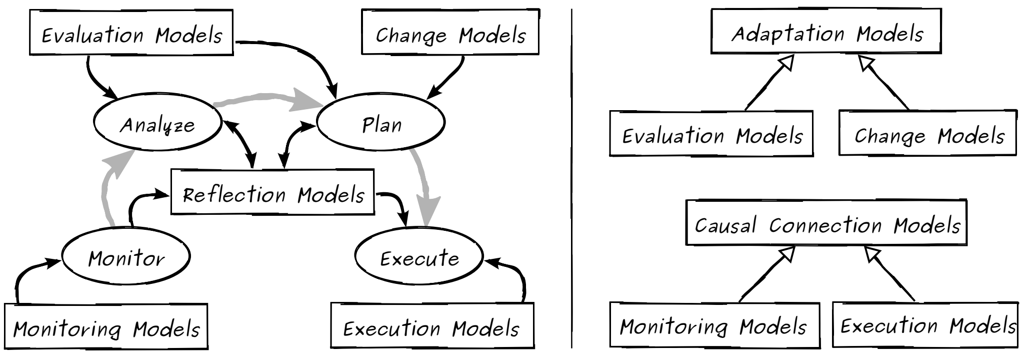


Fig. 2.4: Runtime models for feedback loops

Reflection Models reflect the adaptable software and its environment. The monitor observes the adaptable software and its environment, and updates the reflection models. Thereby, *Monitoring Models* are used that specify the mapping of system-level observations to the abstraction level of reflection models. The reflection models are analyzed to identify adaptation needs. *Evaluation Models* specify the reasoning, e.g., by defining constraints that are checked on the reflection models. If adaptation needs have been identified, 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, like utility preferences, guide the exploration of this space to find an appropriate adaptation, e.g., 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.

Evaluation and change models do not have to be strictly separate models, which is exemplified by event-condition-action rules that address the analyze (evaluating the condition) and the plan (applying the actions) activities in one step. Thus, we might combine evaluation and change models to *Adaptation Models* (see the right hand side of Figure 2.4) specifying the analysis and planning activities that are concerned with decision-making for adaptation [Vogel and Giese 2012b]. Monitoring and execution models are concerned with the synchronization of the adaptable software and the reflection models. This is known as the causal connection [Maes 1987] playing a major role in self-adaptive software [Andersson et al. 2009], such that we consider them as *Causal Connection Models*.

This categorization, which refines the abstract notion of the knowledge base used in MAPE-K and extends our initial proposal [Vogel et al. 2011], shows that multiple and different kinds of runtime models are used simultaneously in a feedback loop. Consequently, we require a language that is able to capture different kinds of runtime models, how individual adaptation activities use these models, and how this usage effects the execution of activities.

2.3 Sensors and Effectors & Monitor and Execute

The adaptation engine and the adaptable software are connected by sensors and effectors (cf. Figure 2.2). Thus, the modeling language has to cover when the monitor and execute activities, that use these sensors and effectors, respectively, are activated to maintain the causal connection between the reflection models and the adaptable software at runtime (cf. Section 2.2). This includes updating the reflection models according to observed sensor data, and changing the adaptable software according to the planned adaptation in the reflection models. Since sensors and effectors usually depend on the specific adaptable software, we 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. However, for triggering conditions of feedback loops, the language should support referencing sensor data, e.g., to characterize sensor events that should initiate the execution of a feedback loop.

Programming language and middleware platforms have recognized the need for runtime management [CAC 2002; Issarny et al. 2007] by supporting the development of sensors and effectors or already providing them through application programming interfaces (APIs). Examples of such platforms are *Java Management Extensions (JMX)*[1] for *Java*, the *GlassFish*[2] application server, or the *OSGi*[3] platform. Additionally, work as been done to enrich such standard management APIs, e.g., for *GlassFish* [Bruhn and Wirtz 2007; Bruhn et al. 2008]. Thus, we can assume that software realized for such platforms provide sensors and effectors, which make the software observable and adaptable. Besides our work [Vogel et al. 2010; Vogel and Giese 2010], other approaches to adaptive software [Cheng 2008; Morin et al. 2009b; Song et al. 2011] also rely on this assumption.

Depending on the available sensors and effectors, *parameter adaptation*, *structural adaptation*, or even a combination of both [McKinley et al. 2004] can be realized. An adaptation engine 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. In particular, when focusing on adaptation at the level of software architectures, we require sensors and effectors supporting structural adaptation.

In addition, but only relevant for advanced cases in which the monitor and execute activities of feedback loops are adapted, the sensors and effectors might have to be adapted as well. This has been investigated especially for adaptive monitoring [Ramirez et al. 2010; Villegas and Müller 2010; Ehlers and Hasselbring 2011; Villegas et al. 2013]. This requires that the modeling language should also work for cases where the adaptable software is dynamically instrumented by adding, removing, or changing sensors and effectors.

Finally, for layered architectures as discussed in the following, adaptation engines are themselves subject to adaptation. In this case, the engines have to provide sensors and effectors to obtain reflective views of them as a basis for adaptation [Andersson et al. 2009]. Therefore, the modeling language for specifying feedback loops should support creating and maintaining reflective views of feedback loops.

[1] JMX Instrumentation and Agent Specification, v1.4, `http://www.jcp.org/en/jsr/detail?id=3` (05.03.2013)
[2] Server for the Java Enterprise Edition (Java EE), `http://glassfish.java.net/` (05.03.2013)
[3] OSGi Core Release 5 Specification, OSGi Enterprise Release 5 Specification, `http://www.osgi.org` (05.03.2013)

2.4 Layered Architecture

In addition to intra-loop and inter-loop coordination as already discussed, there are also cases where the feedback loops have to operate on top of each other (cf. Figure 2.5).

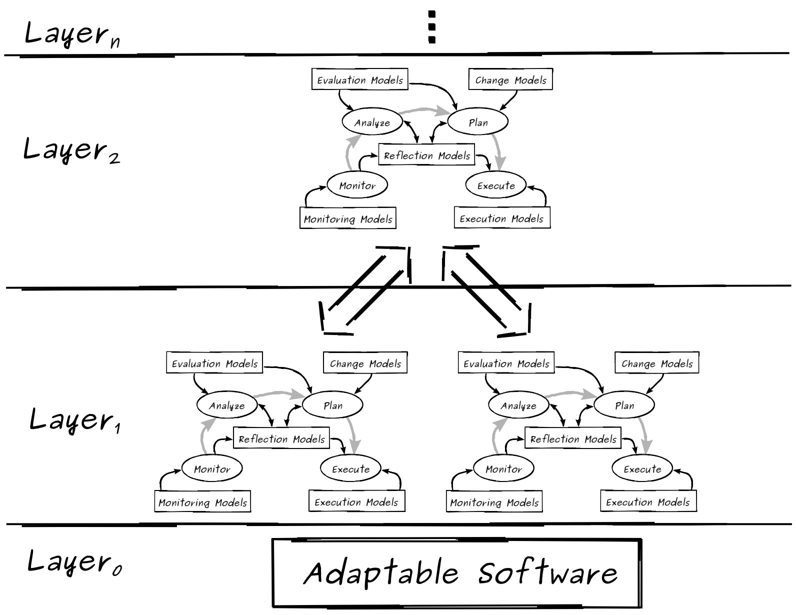


Fig. 2.5: Layered architecture for an adaptation engine

Such a layered architecture is needed for realizing adaptive control schemes [Isermann et al. 1992; Astrom and Wittenmark 1994; Kokar et al. 1999], hierarchical control architectures [Findeisen et al. 1980; Kephart and Chess 2003], layered architectures as proposed for robot software [Gat 1997], the reference architecture for self-managed software systems [Kramer and Magee 2007], or hierarchical structures with internal layers employed for the software of self-optimizing mechatronic systems [Hestermeyer et al. 2004]. Though the architecture proposed by Gat [1997], Kramer and Magee [2007], or adaptive control schemes [Isermann et al. 1992; Astrom and Wittenmark 1994; Kokar et al. 1999] usually consist of three or fewer layers, there are also approaches having potentially a higher number of layers [Findeisen et al. 1980; Hestermeyer et al. 2004].

In a layered architecture for adaptation engines, besides coordinating feedback loops at different layers, a feedback loop at a higher layer can adapt the feedback loop at the layer below. Therefore, some form of reflection of the lower-layer feedback loop has to be provided at runtime, which enables the adaptation of this feedback loop [Andersson et al. 2009]. If parameter adaptation is required, it would be sufficient that the higher-layer loop can change some variables of the lower-layer loop, like threshold values that effect the analysis and planning activities. However, for structural adaptation, the higher-layer feedback loop has to operate on a reflection model structurally representing the feedback loop at the layer below and not only the feedback loop's parameters.

Thus, the modeling language should leverage layered architectures by supporting adaptable feedback loops at $Layers_{1..n}$ and the provision of reflection models representing these feedback loops (in contrast to reflection models representing the adaptable software discussed in Section 2.2) to enable their structural adaptation by higher-layer feedback loops. In this context, *declarative* and *procedural reflection* as proposed by Maes [1987] for programming languages should be supported. In declarative reflection, a separate representation of the program is maintained and used for meta-computations, like adaptation. In contrast, procedural reflection maintains no separate representation and the program is directly used by meta-computations. Additionally, the language should support specifying feedback loops at $Layers_{2..n}$ that perform parameter and structural adaptation of the feedback loops at the layers below by operating on the corresponding reflection models.

2.5 Off-line Adaptation

Finally, 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 usually performed off-line in the context of maintenance and evolution. However, it cannot be expected that self-adaptive software is able to cope with all needs for evolution itself and thus, to fully automate and take over *all* required off-line adaptation activities.

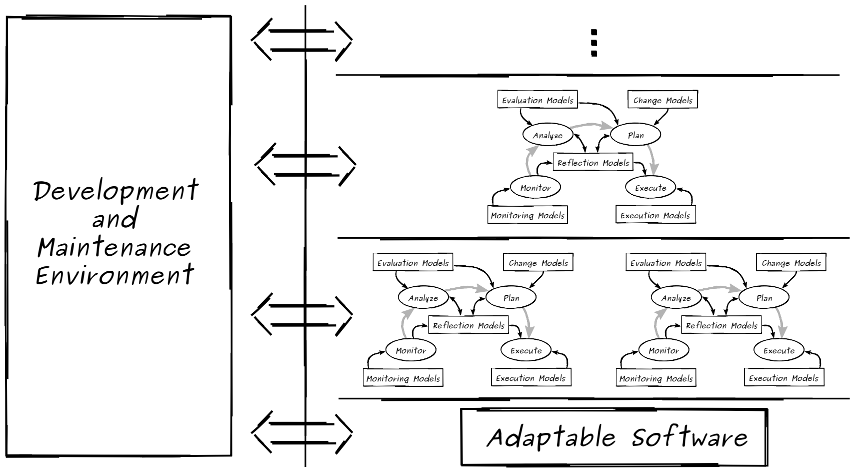


Fig. 2.6: Off-line adaptation of self-adaptive software

Consequently, besides realizing a software with an adaptation engine for the self-adaptation, a solution for the co-existence of such an engine with off-line adaptation, and thus with typical maintenance and evolution is required [Gacek et al. 2008; Morin et al. 2009c; Andersson et al. 2013]. Similar to [Andersson et al. 2013], 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 or maintenance environments. In contrast, if an adaptation activity is performed internally to the self-adaptive software, we refer to *on-line* adaptation. Of particular interest is the enactment 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 language and its runtime environment should support the co-existence of self-adaptation and maintenance/evolution, for example, by providing an interface to monitor the adaptation engine and to integrate adaptations that have been analyzed and planned off-line to the running self-adaptive software for on-line execution.

As depicted in Figure 2.6, such a co-existence has to support that the adaptable software can be adjusted at runtime even though multiple feedback loops operate on top of it, and that the feedback loops at different layers can be adjusted at runtime. These adjustments are adaptations analyzed and planned off-line by engineers in a *Development and Maintenance Environment* and executed on-line to the running self-adaptive software.

Chapter 3

Modeling a Feedback Loop

This section introduces the EUREMA modeling language for engineering adaptation engines in self-adaptive software. The language is based on the concept of megamodels that originates from the research field of model management in model-driven software development. A *megamodel* refers to a model that contains other models and relationships between these contained models while the relationships constitute operations working on these models, like a model transformation [Barbero et al. 2007; Bézivin et al. 2003, 2004; Favre 2005]. EUREMA adopts this generic concept for specifying and executing feedback loops in self-adaptive software by considering the feedback loop's knowledge as runtime models and the individual adaptation activities as model operations working on these runtime models.

Thus, EUREMA (mega)models explicitly specify feedback loops by capturing the runtime models, the interplay between these models, and the flow of adaptation activities operating on these runtime models. Moreover, EUREMA models are kept alive at runtime for executing feedback loops. Thereby, megamodel concepts are leveraged at runtime such that a feedback loop's runtime models and adaptation activities are explicitly maintained at runtime beyond the initial development-time of the self-adaptive software.

The proposed EUREMA modeling language is specific for the engineering of adaptation engines for self-adaptive software and based on general modeling concepts for behavioral and structural diagrams. Therefore, EUREMA provides two types of diagrams: (1) A behavioral *feedback loop diagram* (FLD) is used to model a complete feedback loop or parts of a loop by means of individual adaptation activities and runtime models. An FLD constitutes a *megamodel module* that encapsulates the details of a feedback loop or adaptation activities. (2) A structural *layer diagram* (LD) describes how the different megamodel modules and the adaptable software are related to each other in a concrete situation of the self-adaptive software. Thus, an LD provides an abstract and complete view of an instance of the self-adaptive software restricted to its architecture. This view considers feedback loops as black boxes encapsulated in megamodel modules, their relationships to each other and to the adaptable software, while white-box views of megamodel modules are provided by FLDs.

In the following, we discuss the EUREMA language and its diagram types, and we show by illustrative examples how a feedback loop can be specified by EUREMA models.

3.1 Overview of the EUREMA Language

The EUREMA language we are proposing specifies a feedback loop in an FLD by means of *models*, *model operations*, and the *control flow* between operations. The language shares characteristics with flowcharts and data flow diagrams: models are the data that represent, e.g., the adaptable software, and model operations organized in a control flow are computations that use and work on models. As an example, an operation can be an engine that checks constraints defined in a model on an architectural model of the adaptable software.

We modeled the MAPE-K feedback loop with EUREMA to give an overview of the language. Figure 3.1 shows the FLD that is framed and labeled with its name *MAPE-K*. This feedback loop has an initial and a final state (*Start* and *Executed*, respectively) as entry and exit points for its execution. The 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, like *monitored* of the *Monitor* operation. The *control flow* between operations is explicitly specified by solid arrows. Operations work on runtime *models* represented by rectangles and the usage of models as *inputs* or *outputs* is depicted by dotted arrows. Thus, this megamodel specifies the sequential execution of the four adaptation activities *Monitor*, *Analyze*, *Plan*, and *Execute* that operate all on the *Knowledge* that is captured in a runtime model.

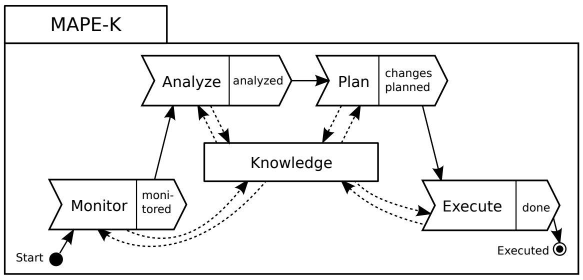


Fig. 3.1: Feedback Loop Diagram (FLD) for *MAPE-K*

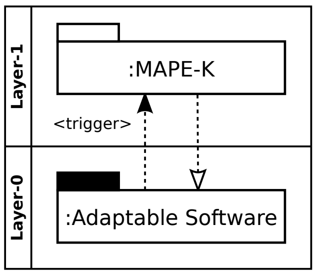


Fig. 3.2: Layer Diagram (LD) for *MAPE-K*

Since we consider adaptation engines with potentially multiple feedback loops and a layered architecture (cf. Section 2), feedback loops are located at a certain layer and they have relationships or dependencies to other feedback loops or to the adaptable software. To properly design, analyze, and understand adaptation engines with multiple feedback loops in layered architectures, an abstract and complete view of the self-adaptive software is required, which makes all feedback loops and their relationships to other feedback loops or to the adaptable software visible. Therefore, EUREMA provides the layer diagram (LD) as it shown for MAPE-K in Figure 3.2. This diagram reflects an instance view of the self-adaptive software and it specifies that an instance of the *MAPE-K* feedback loop is located at *Layer-1* and that it directly senses and effects the *Adaptable Software* instance at *Layer-0*. Sensing relationships are reflected by dotted arrows with filled arrowheads, effecting relationships by dotted arrows with hollow arrowheads. A feedback loop that is specified by an FLD in EUREMA is depicted as a package with a white tab. In contrast, a package with a black tab represents (software) modules, like the adaptable software, that are not specified by EUREMA. Finally, partitions in LDs represent the layers of the self-adaptive software.

In general, a feedback loop modeled with EUREMA requires the specification of a trigger that determines *when* an instance of the feedback loop should be executed. In EUREMA, triggers for a feedback loop refer to occurrences of events emitted from the modules (adaptable software or feedback loops) that are observed by the feedback loop. Thus, a trigger is defined in the LD by annotating the corresponding sensing relationship through which events are emitted and observed. Considering the LD for MAPE-K (Figure 3.2), the trigger is annotated to the arrow representing the sensing relationship between the *:MAPE-K* feedback loop and the *:Adaptable Software*. Since the MAPE-K feedback loop is a blueprint, we omit details of triggers here, but we discuss them for the following examples.

Summing up the overview of the EUREMA language, white-box views and therefore detailed behavioral specifications of feedback loops are addressed by FLDs (e.g., Figure 3.1), while an LD provides a complete structural view of an instance of the self-adaptive software with black-box views of feedback loops (e.g., Figure 3.2). In the following, we discuss EUREMA in detail for advanced cases of adaptation engines and feedback loops, which also enhances and refines the modeling language for both types of diagrams.

3.2 Modeling a Single Feedback Loop

To discuss EUREMA in detail, we consider a component-based application to be the adaptable software. We assume that the application is adaptable at the architectural level by changing parameter values of components, by adding/removing components from the architecture, and by changing the composition of components. Tackling adaptation at the architectural level is a popular approach [Oreizy et al. 1998, 2008; Garlan et al. 2004; Morin et al. 2009a] as it provides promising abstractions for parameter and structural adaptation [McKinley et al. 2004]. We keep the adaptable software generic and abstract in this article because we do not want to discuss a specific adaptation engine for a specific concern and adaptable software. In contrast, we want to cover different variants and requirements for adaptation engines, which allows us to discuss the whole spectrum of modeling concepts in EUREMA.

At first, self-repair capabilities should be added to the adaptable software by a feedback loop as shown in the LD in Figure 3.4. The FLD depicted in Figure 3.3 specifies the behavior of the *Self-repair* feedback loop that aims for automatically recovering the adaptable software from failures. Since this FLD specifies all adaptation activities for self-repair, i.e., the monitor, analyze, plan and execute activities (MAPE), the *:Self-repair* megamodel module in the LD is labeled with *MAPE*. Inspired by [Weyns et al. 2013], we use such labels to indicate which adaptation activities are realized by a megamodel module as defined in the FLD.

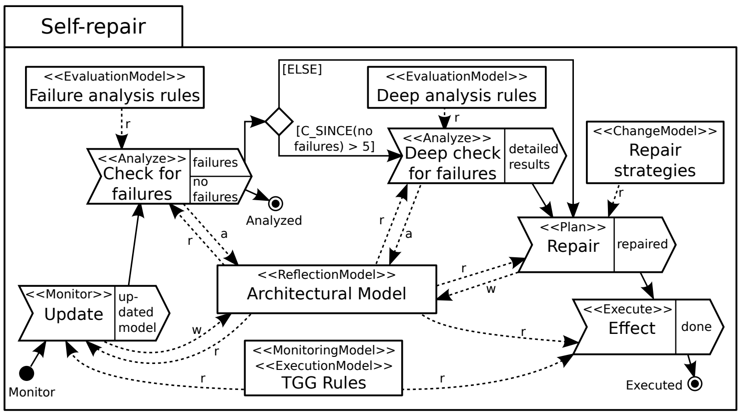


Fig. 3.3: FLD for *Self-repair*

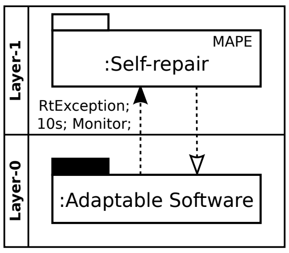
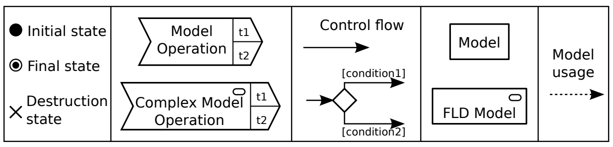
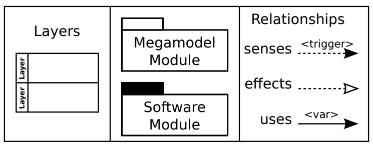


Fig. 3.4: LD for *Self-repair*



(a) FLDs



(b) LDs

Fig. 3.5: Concrete syntax of the EUREMA language for (a) FLDs and (b) LDs

In contrast to the FLD for *MAPE-K* (Figure 3.1), this FLD employs refined concepts of the language, whose concrete syntax is depicted in Figure 3.5. The refined concepts 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 refers to counter and timing information about the execution of the feedback loop. Thus, conditions are generic and use execution information related to EUREMA concepts to branch the control flow, while the different exit compartments of model operations may depend on domain-specific information only known internally to user-defined runtime models and model operation implementations. The other concepts of the concrete syntax for FLDs and LDs have already been outlined in Section 3.1 or they will be discussed in the rest of the article. Additionally, to support the modeler's perception of FLDs, elements in FLDs can be substantiated by labels or stereotypes.

Model operations are assigned to the typical adaptation activities of a feedback loop: ≪*Monitor*≫, ≪*Analyze*≫, ≪*Plan*≫, and ≪*Execute*≫ (cf. Section 2.1). Models are stereotyped based on the purpose they serve in self-adaptive software, which resulted from a categorization of runtime models discussed in Section 2.2: ≪*MonitoringModel*≫, ≪*ExecutionModel*≫, ≪*CausalConnectionModel*≫, ≪*ReflectionModel*≫, ≪*EvaluationModel*≫, ≪*ChangeModel*≫, and ≪*AdaptationModel*≫. Finally, the use of models by model operations is substantiated to **c**reating, **d**estroying, **w**riting, **r**eading, and **a**nnotating models. While reading a model does not have any side effects, writes modify the model in a way that potentially affects the adaptable software, and annotations to a model enrich a model without affecting the adaptable software.

With the FLD depicted in Figure 3.3, we modeled an extended version of the self-repair setting used in [Vogel and Giese 2010]. 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 adaptable software is synchronized with the running software. Thus, based on observations of the running software, the *Update* operation keeps the *Architectural Model* up-to-date. The following analysis is conducted by the *Check for failures* operation that employs *Failure analysis rules* on the *Architectural Model*. These rules define checks and constraints to identify critical failures. If no critical failures are identified, the feedback loop terminates. Otherwise, adaptation is required to repair these failures. At first, a decision is made whether further analysis is needed. This is the case when the condition holds, which checks whether the last execution of the *Check for failures* operation that has identified *no failures* happened more than five consecutive executions in the past. Thus, the past five runs of the feedback loop were not able to repair the failures and a more thorough analysis may provide useful information for the planning activity. This planning activity uses the analysis results annotated by the previous operations to the *Architectural Model* to select suitable *Repair strategies*. The selected strategies change the *Architectural Model* to prescribe a reconfiguration of the running adaptable software. This reconfiguration is executed to the adaptable software by the *Effect* operation that synchronizes the changes on the *Architectural Model* to the adaptable software and that finally terminates one run of the feedback loop.

This example illustrates how adaptation activities can be considered as abstract model operations that work on runtime models. Besides the control flow between multiple operations, the interplay between operations and runtime models has to be made explicit because the models are the basis for coordinating different adaptation activities within a feedback loop. Thus, this interplay is similar to dependencies between adaptation activities, which are relevant for properly specifying and executing feedback loops in EUREMA.

3.3 Trigger for Feedback Loops

In general, an EUREMA specification of a feedback loop includes a trigger that determines *when* an instance of the feedback loop should be executed. We consider occurrences of events as triggers for feedback loop instances. Events are emitted from modules that are either the adaptable software or feedback loops. We assume that the events are modeled in a type hierarchy that is referenced by trigger specifications. Thereby, a trigger specification for a feedback loop may only refer to events emitted from those modules that are sensed by the feedback loop. Thus, triggers are specified in LDs by annotating the corresponding sensing relationships that reveal the flow of events from one module to another module.

This is exemplified in the LD in Figure 3.4 that defines a trigger for the *:Self-repair* feedback loop sensing the *:Adaptable Software*. The trigger specification *RtException; 10s; Monitor;* defines that the *:Self-repair* feedback loop starts execution if the *:Adaptable Software* emits an event of type *RtException* notifying about a runtime exception in the adaptable software, and when ten seconds since the last execution of the *:Self-repair* feedback loop have expired. Finally, *Monitor* points to the initial state of the self-repair feedback loop (cf. FLD in Figure 3.3), in which the execution should start.

In general, EUREMA supports a simple language for specifying triggers. A trigger for a feedback loop instance consists of three parts: *<events>; <period>; <initialState>;*. The first part, *<events>*, refers to a list of event types. If an event of one of these types occurs, the trigger is activated. The second part, *<period>*, defines the minimal time period between two consecutive runs of the 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, even if the required event occurs before the specified time period has elapsed, the next run of the feedback loop instance will be delayed until the time period eventually has elapsed. Delaying the execution of the feedback loop instance avoids thrashing effects due to the proliferation of events and it allows the adaptation being executed by the previous run of the feedback loop instance to take effect in the self-adaptive software. Likewise, selecting specific event types in the *<events>* part of a trigger also serves as a filter that avoids the execution of a feedback loop instance for every occurring event. Finally, the third and thus the last part of a trigger, *<initialState>* , refers to the initial state as defined in the FLD, in which the feedback loop instance should start its execution.

The first two parts of a trigger specification are optional, but one of them must be present. 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 executes the feedback loop instance when the corresponding events have occurred and the current run of the instance has terminated.

In EUREMA, an instance of a feedback loop is not reentrant, and thus, any events that occur while the feedback loop instance is running are queued. Thus, there are no concurrent executions of the same feedback loop instance.

3.4 Modularizing Feedback Loop Diagrams

Besides modeling a complete feedback loop in a single FLD, EUREMA supports the modular specification of feedback loops. Thus, individual adaptation activities of a feedback loop are specified in distinct FLDs that can be composed to form a complete feedback loop. The motivation for a modular specification is twofold. First, it provides further abstractions for the engineer. Specifying a more complex feedback loop also makes the related FLD complex and hard 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, this supports options for reuse and variability modeling. Parts of a feedback loop that are specified in dedicated FLDs can be reused in other feedback loops or they can be replaced by alternative parts specified in other FLDs. The latter leverages the modeling of variability for feedback loops in an adaptation engine.

For instance, the analysis activity of the *Self-repair* feedback loop depicted in Figure 3.3 can be abstracted and specified in its own FLD called *Self-repair-A* as shown in Figure 3.6. This analysis activity has one initial state (*Start*) and two final states reflecting whether critical failures have been identified (*Failures*) or not (*OK*). This FLD can be (re)used and invoked as a megamodel module by other FLDs. Therefore, we introduce the concept of a *complex model operation*, which is shown for the *Self-repair-A* FLD in Figure 3.7.

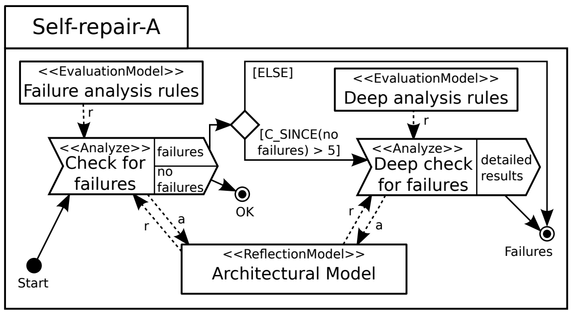


Fig. 3.6: FLD for the analyze activity of self-repair

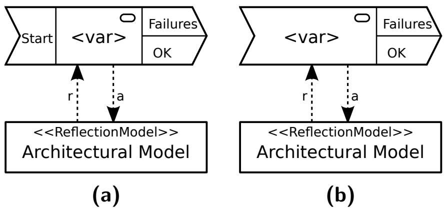


Fig. 3.7: Complex model operation

A complex model operation defines a signature to invoke a megamodel module defined by an FLD. In general, a signature refers to the initial and final states of the corresponding FLD and to runtime models that have to be provided as parameters of the invocations. In the example, based on the initial and final states of the *Self-repair-A* FLD (Figure 3.6), the complex model operation shown in Figure 3.7a 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 the feedback loop using a complex model operation can properly invoke a module by defining the initial state for starting the execution of the module, and that it can properly resume execution after the invoked module has terminated by referring to the final states. If an FLD 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 depicted in Figure 3.7b that shows the 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. Moreover, the signature of an FLD defines which runtime models have to be provided as parameters when invoking a module. In the example, it is the *Architectural Model* as shown in Figure 3.7, while the other runtime models used in the *Self-repair-A* megamodel module, namely *Failure analysis rules* and *Deep analysis rules*, are provided by the module itself. Since we consider self-adaptive software that evolves throughout its lifetime, EUREMA adopts a dynamic typing approach in contrast to a static type system for megamodel modules and complex model operations.

Finally, a complex model operation is labeled with an icon, a small rounded rectangle, to distinguish it from the other type of model operations in FLDs and to reveal that it uses and invokes another megamodel module defined by an FLD. Moreover, when using a complex model operation in an FLD, it must be given a name for the variable <*var*> (cf. Figure 3.7), which is bound to a concrete megamodel module that actually should be invoked.

For example, to (re)use the analysis activity by means of the FLD shown in Figure 3.6, one of the complex model operations depicted in Figure 3.7 can be used like a normal model operation in any other

FLD specifying a feedback loop. Figure 3.8 depicts the FLD for the *Self-repair* feedback loop that uses the complex model operation, named *Analyze*, to invoke the analysis activity. Thus, a complex model operation used in an FLD abstracts from another FLD and it synchronously invokes the adaptation activities specified in the abstracted FLD when being executed. Thus, an FLD used by another FLD does not need a trigger specification as its execution is triggered by an explicit call when the corresponding complex model operation is executed.

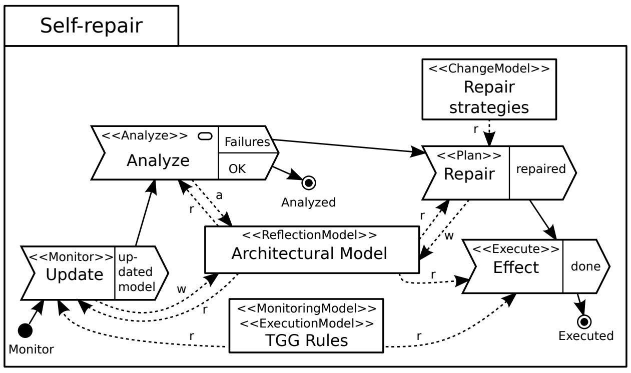


Fig. 3.8: FLD for the *Self-repair* feedback loop using a complex model operation depicted in Figure 3.7b to invoke the analysis activity defined in the FLD shown in Figure 3.6.

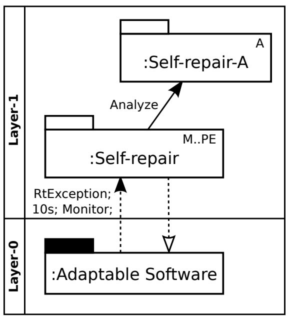


Fig. 3.9: LD for *Self-repair*

Usage relationships between megamodel modules defined by FLDs are dependencies that have to be made explicit, which is addressed in EUREMA by structural LDs. The LD for the self-repair example shown in Figure 3.9 explicitly models that the *:Self-repair* feedback loop uses the *:Self-repair-A* module. A usage relationship is reflected by a solid arrow with a filled arrowhead and it is labeled by the variable name of the corresponding complex model operation (*Analyze* in this example) to bind the operation to the concrete megamodel module (*:Self-repair-A* in this example) to be invoked. Moreover, the LD shows by the labels *M..PE* and *A* attached to megamodel modules that the *:Self-repair* module directly realizes the monitor, plan, and execute activities, while the analyze activity has been extracted and realized by the *:Self-repair-A* module.

Altogether, the specification of the self-repair feedback loop as shown in Figures 3.6, 3.8, and 3.9 is equivalent to the specification shown in Figures 3.3 and 3.4 considering functionality. The FLD shown in Figure 3.8 also has the same trigger specification as the FLD shown in Figure 3.3. The only difference is the number of FLDs used for the specification, which is motivated by design decisions concerning appropriate abstractions and modularity. For example, besides the analysis activity, all adaptation activities of a feedback loop can be specified in distinct FLDs, and a high-level FLD comprising four complex model operations for each activity (monitor, analyze, plan, and execute) integrates all the corresponding FLDs. In general, the depth of the abstraction and the related invocation relationships are not restricted. This leverages different abstraction levels for modeling feedback loops and it assists software engineers in modeling and understanding feedback loops.

Moreover, it supports reuse of feedback loop fragments and it reveals variation points in feedback loops. Assuming we have modeled an additional analysis activity called *Self-repair-A2* in a distinct FLD, which employs a different analysis technique than the activity *Self-repair-A*, both activities are alternative analyses to be used by the *Self-repair* feedback loop. This constitutes a variation point reflected in the LD in Figure 3.10. If the FLDs of both alternatives have the same signature, both of them can be used by the same complex model operation. Then, to switch between these alternatives, it is sufficient to change the binding between the complex model operation and the megamodel module, e.g., by re-routing the usage relationship *Analyze* to point to *:Self-repair-A2* instead of *:Self-repair-A* in the LD (cf. Figure 3.10).

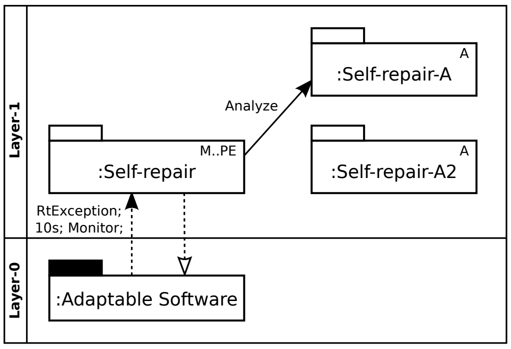


Fig. 3.10: Variability for a complex model operation

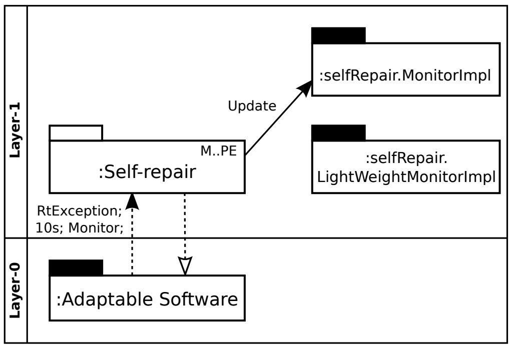


Fig. 3.11: Variability for a model operation

The same idea is applied in EUREMA to bind an (atomic) model operation to the software module actually implementing the behavior of this operation. EUREMA models and in particular FLDs specify when a model operation should be executed, which runtime models are used as input and output, and the return states of the operation. Thus, model operations in FLDs have to be bound to software modules providing implementations for these operations. The LD in Figure 3.11 depicts an example showing that the *Update* operation of the *:Self-repair* module is bound to an implementation, the software module *:selfRepair.MonitorImpl*. Thus, when the *Update* operation is executed, this software module is invoked with the runtime models specified as input of the operation in the FLD (Figure 3.8), namely the *Architectural Model* and the *TGG Rules*. Software modules are typically code-based or reused MDE tools, like a model synchronization engine in this particular case. Thus, software modules are not modeled by EUREMA and therefore, they are represented in LDs by packages with black tabs.

Likewise to alternative megamodel modules for complex model operations (cf. Figure 3.10), alternative software modules for atomic model operations can be modeled in LDs, like shown in Figure 3.11. The software module *:selfRepair.LightWeightMonitorImpl* is an alternative for the *:selfRepair.MonitorImpl* module, and the *Update* operation can be bound to one of them and switch between them by re-directing the usage relationship *Update* from one software module to the other one. In the examples so far and in the following ones, we omit the modeling of software modules in LDs since they just define the binding of model operations to their implementations, which is relevant for the execution, but they are considered as black boxes by EUREMA and provide no further information relevant for the design of feedback loops.

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 execution of feedback loops and adaptation engines. Moreover, they can be leveraged at runtime to adjust feedback loops by switching between these variants, like between different analysis techniques or between different implementations of model operations. Summing up, specifying adaptation engines in multiple FLDs supports modular design and different abstraction levels for feedback loops, reuse of feedback loops fragments, and identifying and exploiting variability of feedback loops during design and execution of self-adaptive software.

Chapter 4

Modeling Multiple Feedback Loops

Having presented the modeling of a single feedback loop, this section discusses how EUREMA addresses the modeling of multiple feedback loops for an adaptation engine. Multiple feedback loops are likely to be employed in a self-adaptive software if multiple concerns, like self-repair or self-optimization, have to be handled, if locality is relevant, like localized and global adaptation, or if control should be decentralized. In the following, we consider the case of multiple concerns. Each concern is handled by an individual feedback loop, because each concern requires its specific runtime models and model operations.

Based on our example discussed so far, the self-adaptive software employs a self-repair feedback loop to handle failures at runtime (cf. Section 3) and it should be additionally equipped with a self-optimization feedback loop to manage performance. This additional loop is specified in the FLD shown in Figure 4.1. From the modeling perspective, the *Self-optimization* feedback loop is quite similar to the self-repair loop. The *Update* and *Effect* operations synchronize the *Architectural Model* with the adaptable software for monitoring and for executing adaptation. A difference to the self-repair feedback loop is that the self-optimization feedback loop has two initial states either initiating the loop with the monitor or the analyze activity. Moreover, in contrast to the self-repair feedback loop, the analysis and planning activities of the self-optimization feedback loop do not only use the *Architectural Model* to exchange information but additionally a *Queueing Model*. It is used to identify bottlenecks in the adaptable software or reasonable values for parameters given by the *Parameter variability* model to adjust the configuration of the adaptable software, which aims for resolving the identified bottlenecks. The self-optimization feedback loop should be triggered in its initial state *Monitor* when the load on the adaptable software's platform increases, which causes a *LoadIncrease* event, and when 60s after the last execution of this feedback loop have expired. This is specified by the trigger in the LD shown in Figure 4.2.

In general, EUREMA supports the specification of multiple feedback loops by distinct FLDs. However, employing multiple feedback loops in a self-adaptive software raises questions of possible interferences or interactions between these feedback loops, which might require coordination of these feedback loops. These questions are discussed in the following.

4.1 Independent Feedback Loops

Assuming there are no interferences between multiple feedback loops used in an adaptation engine, e.g., because the different concerns are not competing or conflicting, the corresponding feedback loops

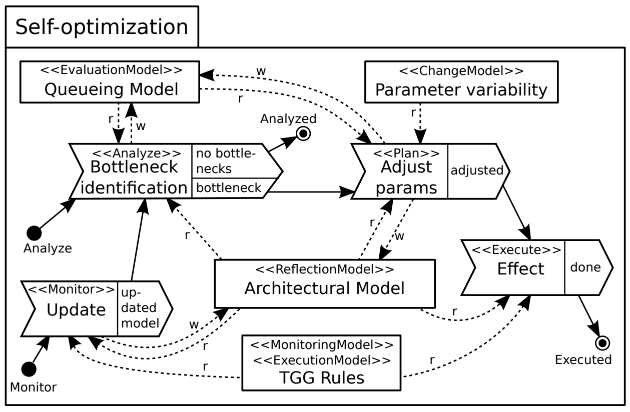


Fig. 4.1: FLD for *Self-optimization*

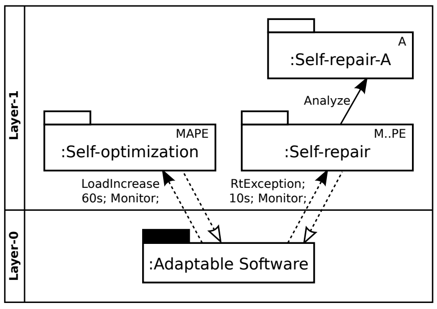


Fig. 4.2: LD for two feedback loops

can be specified and executed completely independent from each other. Both feedback loops have individual triggers that might be activated concurrently. Consequently, the feedback loops might run concurrently and there is no direct interaction or coordination between them.

This is specified in the LD depicted in Figure 4.2 that shows two feedback loops sensing and effecting the adaptable software. In contrast to the LD depicted in Figure 3.9 for the case of a single feedback loop, this LD just adds the *:Self-optimization* feedback loop in parallel to the *:Self-repair* feedback loop within the same layer. Each of the two feedback loops maintains an *Architectural Model* reflecting the adaptable software by the monitoring activity (cf. FLDs in Figures 3.8 and 4.1). By monitoring the adaptable software and updating the architectural model, a feedback loop can observe the adaptation executed to the adaptable software by the other feedback loop. Thus, based on the monitoring, both feedback loops indirectly interact, but there are no direct interactions between the two feedback loops in place.

4.2 Coordination of Multiple Feedback Loops

Though the concurrent and independent execution of multiple feedback loops is conceivable, there are usually interferences or interactions between feedback loops that have to handled by coordinating adaptations of the different loops. For example, there are typical concerns that are competing, like failures and performance, which might result in conflicting adaptations. The self-repair feedback loop might perform an adaptation to heal a failure in the adaptable software, which might degrade the software's performance controlled by the self-optimization feedback loop. The other way around, an adaptation to optimize the performance might cause failures in the adaptable software. Such potential conflicts require the coordination of the feedback loops.

In EUREMA, the coordination of multiple feedback loops is explicitly modeled with FLDs. Such mega-model modules coordinate other modules that specify individual feedback loops by synchronizing their execution. In the following, we discuss two basic design alternatives for coordinating two feedback loops, in particular for self-repair (Figure 3.8) and self-optimization (Figure 4.1). Both alternatives are modeled with FLDs and they ensure the coordinated execution of the feedback loops either by sequencing complete feedback loops or individual adaptation activities of the different feedback loops.

4.2.1 Sequencing Complete Feedback Loops

A simple way to coordinate two feedback loops is to execute them sequentially. This is specified in the FLD depicted in Figure 4.3, which uses complex model operations to synchronously invoke the individual feedback loops. Covering multiple capabilities, like self-repair and self-optimization, leads toward self-management, which motivated the name of the FLD. Explicitly coordinating multiple feedback loops facilitates the sharing of runtime models or even adaptation activities among the loops. As specified by the FLD depicted in Figure 4.3, both feedback loops use the same instances of the *Architectural Model* and *TGG Rules*. Moreover, they share the monitoring activity in certain situations where one feedback loop also performs the monitoring for the other loop as discussed in the following.

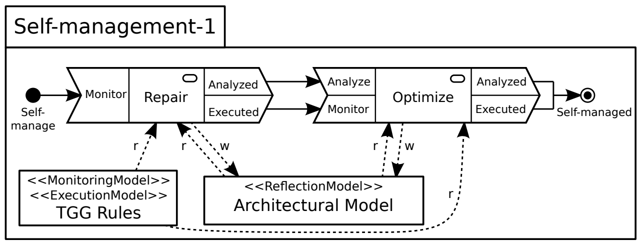


Fig. 4.3: FLD for *Self-management-1*: sequencing complete feedback loops by invoking the self-repair loop (Figure 3.8) followed by the self-optimization loop (Figure 4.1).

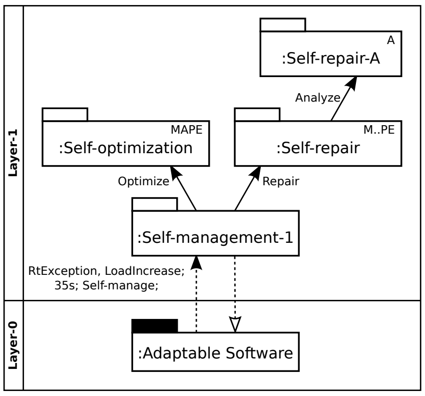


Fig. 4.4: LD for *Self-management-1*

In this self-management example, a higher priority is assigned to repairing failures than to optimizing the performance because failures are often more harmful than slow response times. Moreover, optimizing the performance of a failing system before the failures have been repaired is not reasonable.

Therefore, the self-repair feedback loop is executed before the self-optimization loop. In the FLD shown in Figure 4.3, *Repair* invokes the self-repair feedback loop as specified by the FLD in Figure 3.8 to start in its initial state *Monitor*. Thus, the monitoring and analysis activities are carried out, while the first one updates the *Architectural Model* to reflect the current state of the adaptable software, and the latter one analyzes this model for failures. Depending on whether critical failures have been found or not, the 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 have been identified, the self-repair feedback loop does not need to plan and execute any adaptation and it terminates in the state *Analyzed*. The subsequent self-optimization feedback loop may immediately start with the analysis activity because the monitoring activity of the previous self-repair loop already updated the shared *Architectural Model* and no adaptation has been performed by the self-repair loop on this model and the adaptable software. Thus, the complex model operation *Optimize* in Figure 4.3 invokes the *Self-optimization* loop that begins execution in the initial state *Analyze* (cf. the control flow that connects the exit compartment *Analyzed* of the *Repair* operation to the entry compartment *Analyze* of the *Optimize* operation). If no bottlenecks have been identified, the self-optimization feedback loop terminates in the state *Analyzed*. Otherwise, it carries out the plan and execute activities, and terminates in the state *Executed*.

On the other hand, if the self-repair feedback loop has identified failures, it plans and executes changes to the adaptable software and it terminates in the state *Executed* (cf. Figure 3.8). Thus, the adaptable software has been adapted, which requires that the subsequent self-optimization feedback loop performs the monitoring activity to observe the effects of this adaptation. Thus, the self-optimization loop is invoked by the complex model operation *Optimize* shown in Figure 4.3 to begin execution in the initial state *Monitor* (cf. the control flow that connects the exit compartment *Executed* of the *Repair* operation to the entry compartment *Monitor* of the *Optimize* operation). After carrying out the monitoring and analyze activities, the self-optimization feedback loop either terminates or, if required, performs the plan and execute activities similar to the previous case.

This coordination design synchronizes different feedback loops by sequentially executing them and by using the adaptable software as a synchronization point if one loop performs an adaptation. Thus, an adaptation performed by one feedback loop is executed to the adaptable software before another feedback loop starts its execution with the monitoring activity to observe the executed adaptation and its effects. If a feedback loop does not perform any adaptation, the subsequent loop may start right away with the analysis activity. Thus, there is no need for the subsequent feedback loop to perform the monitoring activity because the previous feedback loop already performed the monitoring and it has not adapted the reflection model (the *Architectural Model* in our example) and the adaptable software. Otherwise, monitoring would be required for observing the effects of the executed adaptation.

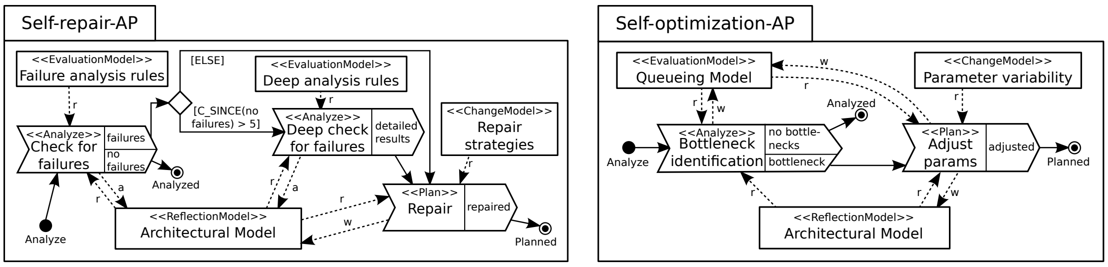
By modeling the coordination in an FLD (Figure 4.3), the coordination becomes explicit at the architectural level and visible in the LD as shown in Figure 4.4. The *:Self-management-1* module uses the *:Self-repair* and *:Self-optimization* modules through the complex model operations *Repair* and *Optimize* that are bound to the corresponding modules. The LD also highlights that the *:Self-management-1* module itself does not perform any adaptation activity because it has no label in contrast to the labels *MAPE* for *:Self-optimization* and *M..PE* for *:Self-repair*. Thus, the *:Self-management-1* just synchronizes the other modules in sensing and effecting the adaptable software.

Consequently, the execution of the *:Self-repair* and *:Self-optimization* feedback loops is not triggered individually and both feedback loops do not need any trigger specification. In contrast, a trigger is specified for the *:Self-management-1* module that activates the execution of these two feedback loops by synchronously invoking them with complex model operations. As specified in the LD in Figure 4.4, the trigger for the *:Self-management-1* module combines the individual triggers of the self-repair and self-optimization feedback loops discussed before. Thus, the *:Self-management-1* module is executed if an event of type *RtException* indicating potential failures or *LoadIncrease* indicating potential performance problems in the adaptable software occurs, and when at least 35s after the previous execution of this module have elapsed.[1] As discussed below, this trigger is also used for the other design alternative.

4.2.2 Sequencing Analysis and Planning of Feedback Loops

The other design alternative for coordinating multiple feedback loops synchronizes the feedback loops in shared monitor and execute activities, while the individual analyze and plan activities are executed sequentially. Therefore, in the example of coordinating the self-repair (Figure 3.8) and the self-optimization (Figure 4.1) feedback loops, the analyze and plan activities of each of these loops are specified in dedicated FLDs as shown in Figure 4.5. In simple terms, the analyze and plan activities have just been cut out from the above FLDs and no changes have been done to their specifications.

[1] To keep the example simple, we have chosen a period of 35s for the coordinated execution of both feedback loops as a compromise of the individual periods of 10s for the self-repair and 60s for the self-optimization loop. However, the example can be extended to support these individual periods: the period for the *:Self-management-1* module can be set to 10s and thus, the self-repair loop is executed in the same period by the operation *Repair* (cf. Figure 4.3). To execute the self-optimization loop with a period of 60s, decision nodes can be added between the operations *Repair* and *Optimize* in Figure 4.3 to check whether 60s since the last execution of the *Optimize* operation have elapsed. If this is the case, the *Optimize* operation will be executed, otherwise it will be skipped and potentially executed in the next run of *:Self-management-1*. Thus, EUREMA does not require that coordinated feedback loops must be executed with the same frequency.


Self-repair-AP
<<EvaluationModel>> Failure analysis rules
[ELSE]
<<EvaluationModel>> Deep analysis rules
[C_SINCE(no failures) > 5]
<<Analyze>> Check for failures
failures
no failures
Analyzed
<<Analyze>> Deep check for failures
detailed results
<<ChangeModel>> Repair strategies
<<Plan>> Repair
repaired
Planned
<<ReflectionModel>> Architectural Model
Analyze
r
a
w
Self-optimization-AP
<<EvaluationModel>> Queueing Model
<<ChangeModel>> Parameter variability
<<Analyze>> Bottleneck identification
no bottle-necks
bottleneck
Analyzed
<<Plan>> Adjust params
adjusted
Planned
Analyze
<<ReflectionModel>> Architectural Model

(a) Self-repair **(b)** Self-optimization

Fig. 4.5: Analyze and plan activities of (a) self-repair and (b) self-optimization.

The coordination of the self-repair and self-optimization feedback loops, specifically of their analyze and plan activities, is specified in the *Self-management-2* FLD depicted in Figure 4.6. Likewise to the previous examples, the *Update* and *Effect* operations synchronize the *Architectural Model* with the adaptable software respectively for monitoring and for executing adaptation. Since the self-repair and the self-optimization feedback loops work on the same instance of the *Architectural Model*, they share the monitor and execute activities. However, the analyze and plan activities are specific for the addressed concerns, namely failures and performance, and they decide if and how the system should be adapted. Thus, they must coordinate each other to tackle competing concerns and as a consequence potentially conflicting adaptations.

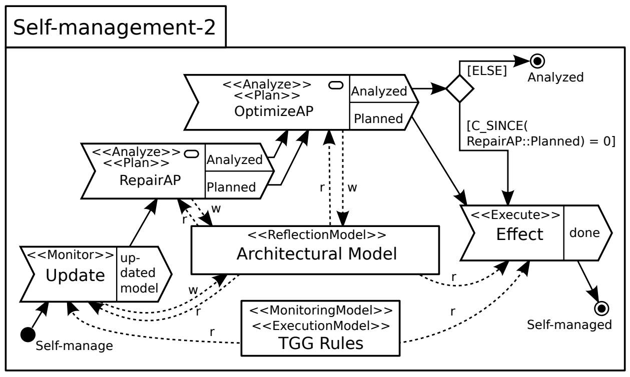

Self-management-2
<<Analyze>> <<Plan>> OptimizeAP
Analyzed
Planned
[ELSE]
Analyzed
[C_SINCE(RepairAP::Planned) = 0]
<<Analyze>> <<Plan>> RepairAP
Analyzed
Planned
<<ReflectionModel>> Architectural Model
<<Execute>> Effect
done
<<Monitor>> Update
up-dated model
<<MonitoringModel>> <<ExecutionModel>> TGG Rules
Self-manage
Self-managed
r
w

Fig. 4.6: FLD for *Self-management-2*: sequencing the analysis and planning activities of the self-repair (cf. Figure 4.5a) and self-optimization (cf. Figure 4.5b) feedback loops.

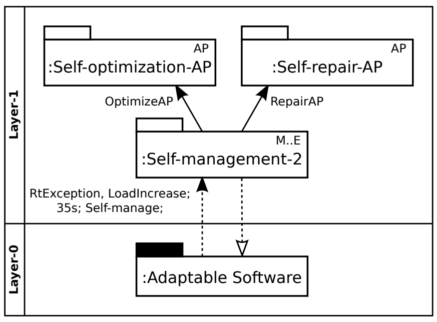


Fig. 4.7: LD for *Self-management-2*

Therefore, the analyze and plan activities for self-repairing failures are executed before the analyze and plan activities for self-optimizing performance. This is modeled in the FLD (Figure 4.6) by complex model operations sequentially and synchronously invoking the modules specifying the analyze and plan activities for both concerns as depicted in Figure 4.5. The *Architectural Model* is only modified by the self-repair's plan activity if the related analyze activity has identified critical failures in the adaptable software. These modifications are planned adaptations in order to repair the failures and they have been applied at the model level but they have not been effected to the adaptable software.

The subsequent analyze and plan activities of the self-optimization feedback loop use the *Architectural Model* that has been potentially modified by the self-repair's plan activity. If the model has not been modified, there are no conflicting adaptations. Otherwise, the adaptations proposed by the self-repair feedback loop must be handled by the self-optimization loop. This is determined by the implementation of the model operations of the feedback loops. Conceivable solutions are that the self-optimization feedback loop must adhere to the adaptation planned by the self-repair feedback loop, or the other way around, that it just has to consider the planned adaptation as a proposal and it may revoke the proposal in order to independently plan its own adaptation. Moreover, compromises between adaptations planned by both feedback loops are conceivable, e.g., based on utility functions.

Considering the FLD in Figure 4.6, 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*. Otherwise, the *Self-management-2* loop terminates in the state *Analyzed* because no critical failures and no bottlenecks have been identified, which does not require any adaptation to be planned and executed to the adaptable software.

The corresponding LD for this design alternative to coordinate the two feedback loops is depicted in Figure 4.7. The *:Self-management-2* module senses and effects the adaptable software and it uses the *:Self-repair-AP* and *:Self-optimization-AP* modules through the complex model operations *RepairAP* and *OptimizeAP*, respectively. Likewise to the other coordination design, the *:Self-management-2* module requires a trigger defining when it should be executed, while the *:Self-repair-AP* and *:Self-optimization-AP* modules are synchronously invoked by the *:Self-management-2* module using complex model operations. For the *:Self-management-2* module, the same trigger can be used as for the *:Self-management-1* module discussed above.

The LD of this coordination design highlights by the label *M..E* that the *:Self-management-2* module performs the monitor and execute activities, while the analyze and plan activities (*AP*) are performed by the *:Self-optimization-AP* and *:Self-repair-AP* modules.

This section and the previous one have discussed how adaptation engines are specified in EUREMA by means of behavioral FLDs defining feedback loops and LDs providing structural views of adaptation engines. Thereby, the coordination of multiple feedback loops can be specified with the same modeling concepts as used for specifying individual feedback loops. In particular, modeling the coordination with FLDs and LDs shows how the coordination can be explicitly specified at a high-level of abstraction. The coordination is achieved by executing interacting feedback loops or interacting adaptation activities of different loops in a controlled manner by sequential and synchronous invocations through complex model operations. Explicitly modeling the coordination enables the sharing of runtime models, like reflection models, or adaptation activities, like monitoring, among multiple feedback loops. This potentially reduces the monitoring costs at runtime. Moreover, if coordination among feedback loops is not required, EUREMA supports the specification and execution of individual feedback loops completely independent from each other.

Chapter 5

Modeling Layered Architectures

So far we have discussed the modeling of feedback loops for adaptation engines with a single layer. However, as argued in Section 2.4, there are particular cases, like adaptive control, where feedback loops have to operate on top of each other. This calls for a layered architecture of adaptation engines, which organizes feedback loops in layers. A feedback loop at a higher layer controls a feedback loop at the layer directly below, while the feedback loops at the lowest layer of an adaptation engine directly control the adaptable software.

Controlling a feedback loop requires adaptable feedback loops that can be adjusted by means of parameter or structural adaptation. 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 architectures for adaptation engines are modeled in EUREMA and how reflection models and adaptation of feedback loops are supported by EUREMA.

Feedback loops specified by EUREMA are adaptable by construction because EUREMA models are kept alive at runtime as feedback loop specifications and they are executed by an interpreter. The EUREMA interpreter is able to cope with dynamic changes of EUREMA models at runtime and even while executing these models. EUREMA supports adaptation of feedback loops that refer to concepts of the EUREMA modeling language.

As reflected in LDs, a feedback loop can be adapted by changing the binding between complex model operations and megamodel modules that are defined by FLDs, and between model operations and software modules implementing these operations. This has been discussed in Section 3.4 and exemplified by the LDs depicted in Figures 3.10 and 3.11. Additionally, LDs define the trigger for a feedback loop instance that can be dynamically adjusted by adapting its parameters <*events*>, <*period*>, and <*initialState*>. Thus, types of events whose occurrences trigger a feedback loop instance 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 adaptation modify the trigger specification or the composition among megamodel and software modules, but they do change internals of a module. While EUREMA completely abstracts from internals of software modules, internals of megamodel modules and therefore feedback loops as defined by FLDs can be dynamically modified.

Adapting internals of a feedback loop concerns EUREMA concepts used in FLDs, namely runtime models and model operations including their control flow and usage of runtime models. Thus, EUREMA supports to dynamically adjust runtime models used within a feedback loop, e.g., to replace the evaluation and change models that determine the analyze and plan activities. Moreover, model operations can be adjusted by adding, removing, or replacing them. This typically requires the adaptation of the control flow reflected by links between model operations, and the usage of runtime models reflected by links between model operations and runtime models. Besides this structural adaptation, the control flow can be adjusted by parameter adaptation when decision nodes are used. Decision nodes enable the conditional branching of the control flow that can be adapted by changing the conditions of individual branches.

In layered architectures of adaptation engines, such adaptations of feedback loops are conducted by other feedback loops operating at higher layers. In EUREMA, the modeling of higher-layer feedback loops that adapt lower-layer feedback loops is similar to modeling any feedback loop as discussed in Section 3. 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. This will be discussed by the following example.

We consider the self-adaptive software with self-repair capabilities as discussed in Section 3. Thus, as shown by the LD in Figure 3.9, the self-repair feedback loop directly senses and effects the adaptable software. This feedback loop aims for automatically healing failures in the adaptable software by applying pre-defined repair strategies (cf. FLD in Figure 3.8). However, these repair strategies need not to be able to handle all kinds of failures. This would require that all kinds of failures could have been anticipated when developing and deploying these strategies, which is usually not the case given the uncertainty concerning self-adaptive systems and their environments. Thus, the repair strategies defined in a runtime model have to be maintained and adapted at runtime. This task can be assigned to another feedback loop that synthesizes new repair strategies on demand and provides them to the self-repair feedback loop. Thus, the other feedback loop is located at a higher layer because it adapts the self-repair feedback loop and it performs time-consuming computations to synthesize new strategies. Besides sensing and effecting relationships between feedback loops at different layers, different time scales of feedback loops are a basic criteria for placing feedback loops in different layers (cf. [Kramer and Magee 2007]). A feedback loop realizing urgent or frequent adaptations must work at shorter time scales and thus, it is placed at a lower layer. In contrast, a feedback loop performing long-term or complex planning that is rather rarely required often work at longer time scales, which places the loop at a higher layer.

In the following, we discuss two options of the higher-layer feedback loop for our self-repair example, either applying *declarative reflection* (Section 5.1) or *procedural reflection* (cf. Section 5.2). In declarative reflection, user-defined reflection models can be applied, while in procedural reflection, the EUREMA models that specify feedback loops are directly used as reflection models.

5.1 Declarative Reflection – User-defined Reflection Models

At first, we discuss the option of declarative reflection between layered feedback loops. The higher-layer feedback loop employing a user-defined reflection model and controlling the self-repair feedback loop is specified by the FLD *Self-repair-strategies* shown in Figure 5.1. The monitor activity observes the self-repair feedback loop and maintains the *Self-repair Model* that reflects the self-repair loop. Using this reflection model, the analyze activity checks the success rates of the existing repair strategies and the plan activity synthesize new repair strategies. These new strategies replace the current strategies in the reflection model. This replacement is enacted to the self-repair feedback loop by the execute activity that removes the runtime model defining the current *Repair strategies* and adds the runtime model defining the new strategies to the corresponding instance of the FLD shown in Figure 3.8. This adaptation equips the self-repair feedback loop with a new runtime model that specifies the newly synthesized repair strategies to be used from now on.

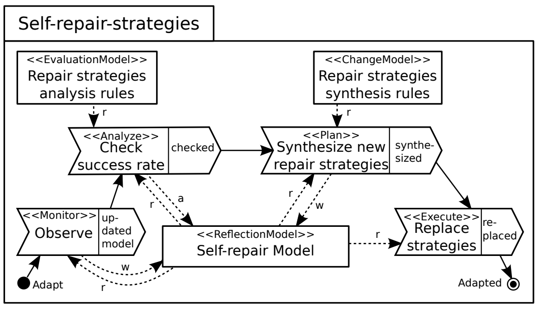


Fig. 5.1: FLD for *Self-repair-strategies*

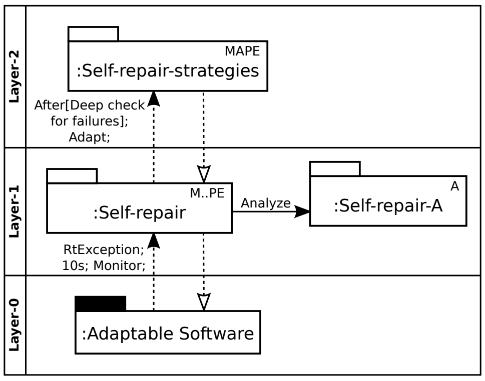


Fig. 5.2: Layered feedback loops for self-repair: the *:Self-repair-strategies* module at *Layer-2* senses and effects the *:Self-repair* module at *Layer-1*.

As defined by the LD depicted in Figure 5.2, an instance of the *Self-repair-strategies* feedback loop is located at *Layer-2* and it senses and effects the *:Self-repair* feedback loop at *Layer-1*. Sensing and effecting the *:Self-repair* module also includes the sensing and effecting of the *:Self-repair-A* module because the former uses the latter. Such a usage relationship is similar to an inclusion, i.e., the *:Self-repair* module includes the *:Self-repair-A* module.

The trigger for the *:Self-repair-strategies* module may refer to the sensed modules *:Self-repair* and *:Self-repair-A*. In this example, the trigger refers to the event *After[Deep check for failures]* (cf. Figure 5.2) that is emitted by the EUREMA interpreter. While executing a feedback loop defined by an FLD, the interpreter synchronously emits two types of events: *Before[opName]* and *After[opName]* events are emitted *before* and respectively *after* any model operation is executed, while *opName* refers to the name of the corresponding operation. In the example, the *:Self-repair-strategies* module is synchronously executed by starting in its initial state *Adapt* when the *Deep check for failures* model operation of the *:Self-repair-A* module has been executed. This model operation is only executed if more than five consecutive runs of the self-repair feedback loop at *Layer-1* were not able to repair the failures. This indicates that the current repair strategies are not able to heal the failure and new strategies are required.

A particular aspect of this layering of feedback loops is that the *Self-repair-strategies* feedback loop utilizes a user-defined reflection model, called *Self-repair Model* (cf. Figure 5.1). This 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. In the example of the *Self-repair-strategies* feedback loop, the *Observe* and *Replace strategies* operations are concerned with the synchronization of the *Self-repair Model* and the controlled self-repair feedback loop. Thus, the engineer may decide which information about the controlled feedback loop is covered by the reflection model as well as the abstraction level of the model. However, she must 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. This is similar to managing the causal connection between reflection models and the adaptable software as discussed for the examples in Sections 3 and 4.

In this case, the adaptable part of the self-adaptive software are feedback loops specified by EUREMA models that are executed by the EUREMA interpreter at runtime. Thus, sensors and effectors provided by EUREMA can be used to observe and adjust feedback loops by means of the EUREMA models that are kept alive at runtime. For sensing EUREMA models, they can be queried and events emitted by the EUREMA interpreter and the MDE infrastructure[1] notifying about the execution and changes of

[1] EUREMA and its models are based on the *Eclipse Modeling Framework* (EMF) that provides an event mechanism notifying about changes in EMF models.

these models can be used. For effecting EUREMA models, basic means to change models can be used, like changing attribute values of nodes, or adding and removing nodes and relationships.

Such basic sensors and effectors can be used by the engineer to implement model operations for the monitor and execute activities that maintain the causal connection between a feedback loop resp. EUREMA models and the user-defined reflection model.

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 model specifying and executing the lower-layer loop. Thus, two separate representations of the lower-layer loop are used: EUREMA models for specifying and executing the loop, and user-defined reflection models for adapting the loop.

However, the disadvantage is that both representation have to be synchronized to each other, which ensures the causal connection. However, this synchronization can be simplified since both representations are models conforming to MDE principles, i.e., they have potentially different metamodels but the same meta-metamodel. Thus, one-to-one copies of EUREMA models as reflection models can be directly provided by EUREMA, while MDE techniques can be employed, like model synchronization techniques [Giese and Wagner 2009], to keep both models with individual metamodels consistent to each other. The general applicability of model synchronization techniques for runtime reflection models has been shown in [Vogel et al. 2010; Vogel and Giese 2010].

5.2 Procedural Reflection – EUREMA-based Reflection Models

The other option for layered architectures applies procedural reflection between the feedback loops. Thus, the higher-layer feedback loop directly uses EUREMA models as reflection models of the controlled self-repair feedback loop. Thus, no separate representation of the self-repair feedback loop is maintained by the higher-layer feedback loop and the EUREMA models used for specifying and executing the self-repair feedback loop are also used for adapting it. The corresponding higher-layer feedback loop is defined by the FLD *Self-repair-strategies-2* shown in Figure 5.3. This FLD shows the reflection model *feedbackLoopModel*, which is labeled with an icon, a small rounded rectangle, to highlight that this model is directly the EUREMA model used for specifying and executing the controlled self-repair feedback loop. This reflection model is used by the *Self-repair-strategies-2* feedback loop to adapt the self-repair feedback loop. Since the *Self-repair-strategies-2* feedback loop does not maintain a separate representation of the self-repair feedback loop, the causal connection is ensured by construction. Thus, there is no need for explicit monitor and execute activities realizing the causal connection. In contrast, the FLD for the *Self-repair-strategies-2* just defines the analyze and plan activities, similar to the *Self-repair-strategies* feedback loop shown in Figure 5.1. However, in this case the analyze and plan activities are additionally stereotyped with ≪*Monitor*≫ and ≪*Execute*≫, respectively, since the monitor and execute activities are implicitly realized by them.

The LD shown in Figure 5.4 shows the *:Self-repair-strategies-2* module at *Layer-2* that senses and effects the *:Self-repair* module at *Layer-1*. The trigger for the *:Self-repair-strategies-2* module is the same as for the *:Self-repair-strategies* module discussed in Section 5.1, and even both LDs depicted in Figures 5.2 and 5.4 are quite similar except of one aspect. Since the *:Self-repair-strategies-2* module as specified by the FLD in Figure 5.3 uses an EUREMA model as the reflection model called *feedbackLoopModel*, this reflection model must be bound to a concrete EUREMA model at runtime. This binding is defined in the LD by the usage relationship named *feedbackLoopModel* that binds reflection model used in the *:Self-repair-strategies-2* module to the EUREMA model and in particular the FLD instance defining the *:Self-repair* module. Thus, for adapting the self-repair feedback loop, the *:Self-repair-strategies-2* module operates on the FLD instance of the *Self-repair* feedback loop, whose initial specification is shown in Figure 3.8. Thereby, it also uses the instance of the FLD *Self-repair-A*, whose initial

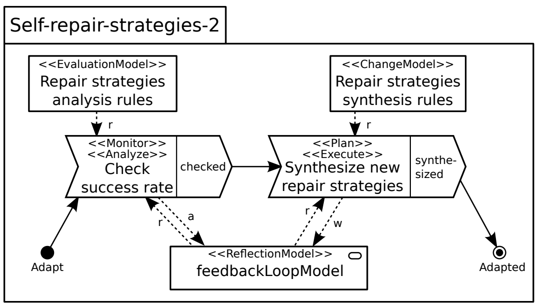


Fig. 5.3: FLD for *Self-repair-strategies-2*

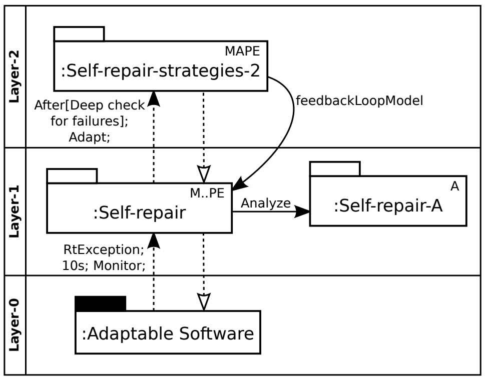


Fig. 5.4: Layered feedback loops for self-repair: the *:Self-repair-strategies-2* module at *Layer-2* senses and effects the *:Self-repair* module at *Layer-1*.

specification is shown in Figure 3.6, because this FLD is used and therefore included in the *Self-repair* FLD.

In general, the FLDs shown in the figures of this article are initial specifications of feedback loops since they are kept alive at runtime and they might be dynamically adapted, which changes the initial specification. In our example, the higher-layer feedback loop adapts the self-repair feedback loop by replacing runtime models that define repair strategies used by the self-repair feedback loop. The sequence diagram depicted in Figure 5.5 illustrates this behavior by describing the logical interactions between the layered feedback loops and their adaptation activities. The gray-shaded instances reflect modules, like the adaptable software or feedback loops, and the hollow instances individual model operations of feedback loops. To keep the sequence diagram simple, we omitted the execution of the initial and final states of the megamodel modules, which are technically specific kinds of model operations.

By *:RtException* events emitted by the *:Adaptable Software*, the trigger of the *:Self-repair* feedback loop is activated to start executing the loop (a). The *:Self-repair* loop executes the *:Update* operation, followed by the *:Analyze* operation (b). The latter one is a complex model operation that invokes the *:Self-repair-A* module (c), that sequentially executes the *:Check for failures* and *:Deep check for failures* operations (d). After the *:Deep check for failures* operations has been executed, the trigger of the *Layer-2* feedback loop is activated and the *:Self-repair-strategies-2* feedback loop is synchronously executed (e). This feedback loop instance sequentially executes the *:Check success rate* and *:Synthesize new repair strategies* operations (f). The latter operation provides new repair strategies to the *:Repair* operation of the *:Self-repair* feedback loop (g). In EUREMA, this provision of new repair strategies is actually done by adapting the FLD instance representing the *:Self-repair* module, in particular replacing the runtime model defining the current strategies with a new runtime model defining the new strategies. Then, the *:Self-repair-strategies-2* feedback loop, the *:Self-repair-A* module, and the *:Analyze* operation consecutively terminate (h). After the analyze activity, the *:Self-repair* feedback loop sequentially executes the plan and execute activities by means of the *:Repair* and *:Effect* operations (i). The *:Repair* operation applies the new repair strategies provided by the *:Self-repair-strategies-2* feedback loop and the *:Effect* operations enacts the adaptation on the *:Adaptable Software* (j).

The advantage of directly using EUREMA models as reflection models of feedback loops at runtime is that the causal connection is ensured by construction. This avoids the development of monitor and execute activities for higher-layer feedback loops and the need for specific sensors and effectors to create and maintain reflection models of feedback loops.

However, by using the same model of a feedback loop for specifying and executing as well as adapting the feedback loop, the adaptation cannot be decoupled from the specification and execution. Thus,

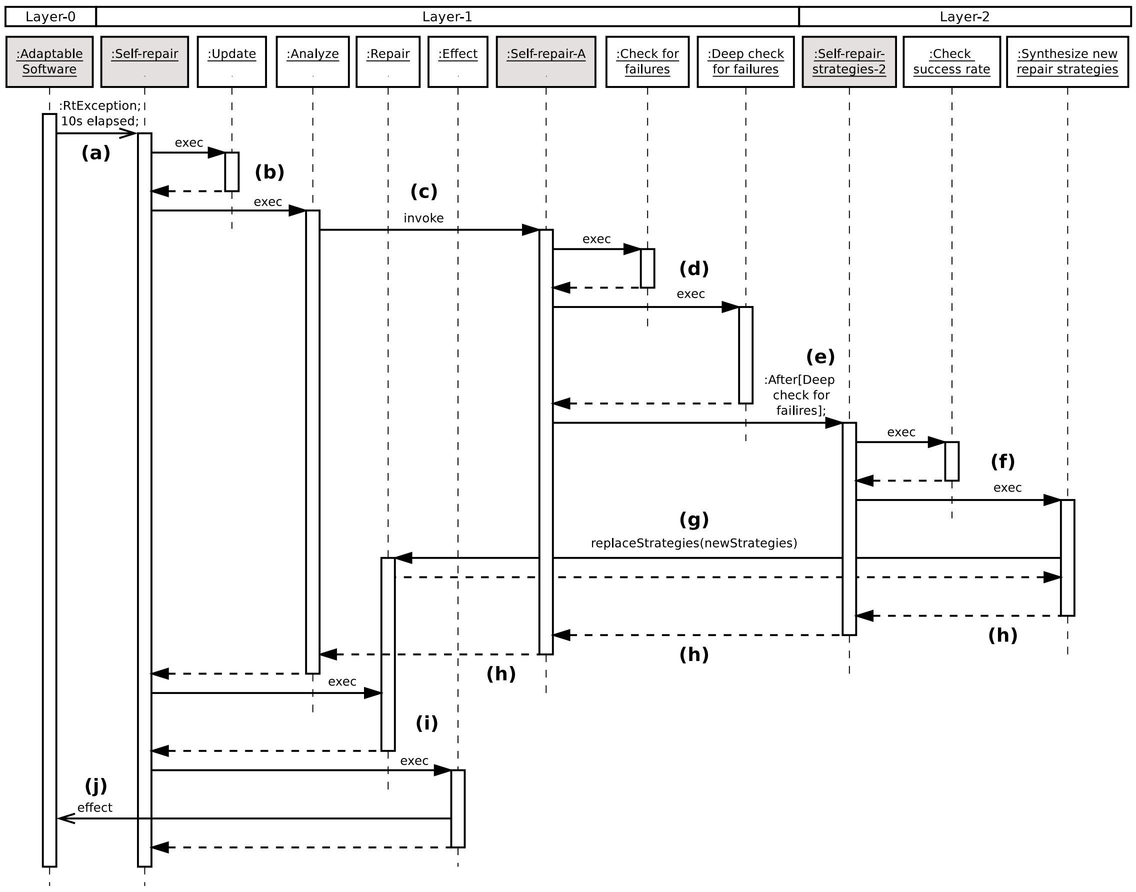


Fig. 5.5: Sequence diagram describing the logical behavior of layered feedback loops

any adaptation performed by the higher-layer feedback loop is immediately enacted to the lower-layer feedback loop.

Chapter 6

Modeling Off-line Adaptation

As we have outlined in Section 2.5, we cannot assume that the self-adaptation capabilities of the adaptation engine even with layered feedback loops are sufficient to automatically handle all adaptation needs over time. Therefore, we need a solution for the *co-existence* of self-adaptation and off-line adaptation.

In this context, a software engineer typically analyzes and plans an adaptation off-line in the development/maintenance environment followed by executing this adaptation on-line to the self-adaptive software. Such an adaptation can be a patch developed by an engineer to evolve the software in a way the capabilities of the adaptation engine is not able to accomplish. For instance, we discussed the need to evolve change models at runtime, like the repair strategies for our self-repair example in Section 5. Assuming there is no higher-layer feedback loop in place that manages the evolution of the repair strategies (cf. Figure 6.1), an engineer may develop new strategies and provide them as a patch to the self-adaptive software. In the following, we discuss the related support by EUREMA.

A software engineer may retrieve snapshots of EUREMA models, as visualized by LDs and FLDs, employed in the adaptation engine of the running self-adaptive software to monitor and analyze the adaptation process. In our example, based on the LD shown in Figure 6.1, which describes the current architecture of the self-adaptive software, and the related FLDs instances of the *:Self-repair* and *:Self-repair-A* modules reflecting the feedback loop, the engineer identifies the need for new repair strategies and develops them. EUREMA supports the subsequent execution of the adaptation, which is similar to a patch enacting the new repair strategies in the adaptation engine.

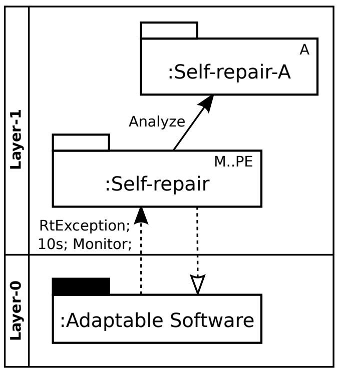


Fig. 6.1: Initial LD

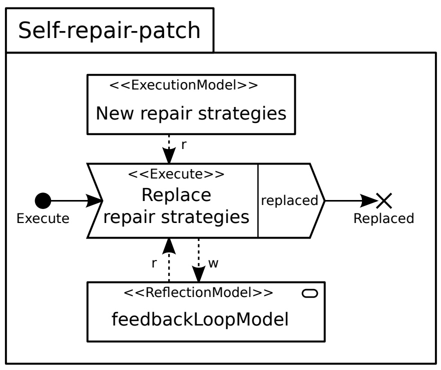


Fig. 6.2: FLD for *Self-repair-patch*

Therefore, an FLD specifying the execution of the adaptation or patch has to be modeled off-line and afterwards executed on-line in the adaptation engine. This is exemplified by the FLD shown in Figure 6.2 that specifies the *Self-repair-patch*. It defines a single model operation that replaces the repair strategies in the *feedbackLoopModel* reflecting and specifying the self-repair feedback loop. Thus, the runtime model *New repair strategies* is linked into the EUREMA model of the self-repair feedback loop visualized in Figure 3.8 and the current runtime model *Repair strategies* is removed. Then, the patch process terminates in a destruction state, denoted by a cross, which is a special final state of an FLD. It defines that an instance of the *Self-repair-patch* module is executed exactly once and when it reaches the final state, it is destroyed and erased.

Having defined the *Self-repair-patch* in the FLD, this module must be integrated into the adaptation engine of the running software for execution. Thus, it must be added to the LD (cf. Figure 6.1) visualizing the EUREMA runtime model representing the current self-adaptive software's architecture. Therefore, the *Self-repair-patch* module is uploaded to the adaptation engine through an interface provided by the EUREMA interpreter. Additionally, a rule as depicted in Figure 6.3 has to be specified and uploaded. This rule specifies how the *Self-repair-patch* module should be integrated into the layered architecture reflected by the LD in Figure 6.1. The elements of this rule that are annotated with $++$ are added to the EUREMA model as visualized by this LD if a match for the rule's elements having no annotation is found in the model.

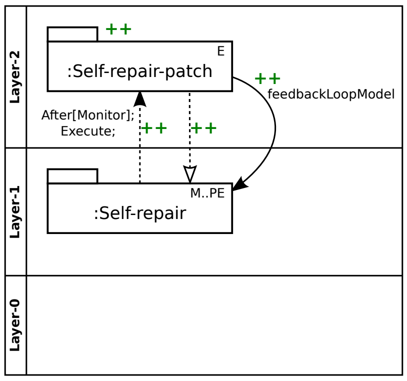


Fig. 6.3: The rule to change the initial LD shown in Figure 6.1

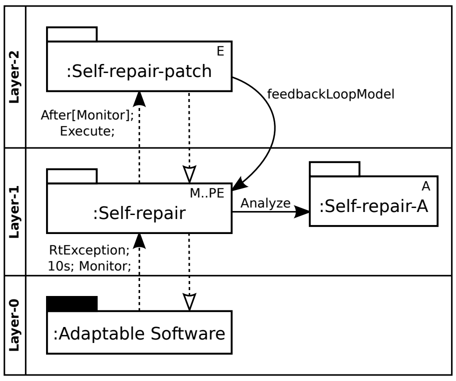


Fig. 6.4: The result of applying the rule depicted in Figure 6.3 on the LD depicted in Figure 6.1

By applying the rule, a match is found for the *:Self-repair* module at *Layer-1*. Thus, the *:Self-repair-patch* module is added to the model and connected by sensing and effecting relationships to the *:Self-repair* module. According to the idea of layered architectures (cf. Section 2.4), the *:Self-repair-patch* module is added to *Layer-2* because it adapts the self-repair feedback loop by replacing repair strategies. Moreover, the reflection model *feedbackLoopModel* used for adaptation by the *:Self-repair-patch* module (cf. FLD in Figure 6.2) is bound to the FLD instance used for executing the self-repair feedback loop (cf. FLD in Figure 3.8). This results into the layered architecture depicted in Figure 6.4.

While the *:Self-repair-patch* module is integrated into the layered architecture, the *:Self-repair* module may continuously run to perform self-adaptation. The *:Self-repair-patch* module will be executed after the integration has been done and its trigger is activated. The trigger *After[Monitor]; Execute;* intercepts the execution of the *:Self-repair* module after this module has executed its initial state in order to execute the *:Self-repair-patch* module for injecting the new repair strategies into the *:Self-repair* module. As discussed above for the FLD of the patch module, the *:Self-repair-patch* module will be executed exactly once and it will be destroyed and removed from the layered architecture afterwards. The resulting architecture is the same as the one before applying the patch (cf. Figure 6.1) except that new repair strategies that have been developed off-line are now used by the self-repair feedback loop.

This example shows how EUREMA supports the integration and execution of adaptation that have been planned and developed off-line by engineers. Thereby, the same modeling concepts of FLDs as for specifying feedback loops are used for specifying the execution of an off-line adaptation.

Likewise to the example of applying a patch to the adaptation engine, an engineer may specify a complete feedback loop off-line by FLDs, which are then uploaded and integrated to the adaptation engine. In contrast to the patch module that is added to the layered architecture, executed once, and then removed from the architecture, the feedback loop modules remain in the architecture until it is eventually removed by future (off-line) adaptations.

Assuming the engineer has developed mechanisms to automatically synthesize new repair strategies in our self-repair example, she may delegate the maintenance of the repair strategies to the adaptation engine. Thus, she specifies a corresponding feedback loop for *Layer-2* using FLDs and integrates this feedback loop module to the adaptation engine similar to the way it was done for the patch module. This results in a layered architecture with a hierarchy of feedback loops, which we discussed in Section 5. Thus, the architecture of the self-adaptive software evolves from a two-layer architecture (cf. Figure 6.1) to a three-layer architecture (cf. Figures 5.2 or 5.4). Thus, EUREMA addresses the need to evolve self-adaptive software by adding or removing feedback loops at different layers of the adaptation engine. This is realized by means of adaptation analyzed and planned off-line, but enacted on-line to the running software. This supports adaptation of self-adaptive software, which has not been anticipated when initially developing and deploying the software. In our example, the need to maintain repair strategies was not anticipated, but it can be covered by continuously patching the adaptation engine or by integrating a higher-layer feedback loop.

Furthermore, EUREMA provides basic support to integrate legacy software modules in adaptation engines. Such modules are treated as black boxes by EUREMA in contrast to megamodel modules that are specified by FLDs. Thus, EUREMA just addresses the activation of legacy modules by modeling sensing and effecting relationships in LDs. For example, Figure 6.5 defines the *native* triggering of the *:legacy.Self-optimization* module, which is controlled by glue code that hard-wires the legacy module to the adaptable software. Thus, the EUREMA interpreter cannot interfere or support off-line adaptation.

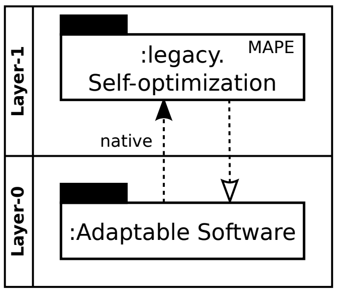


Fig. 6.5: Hard-wired legacy module

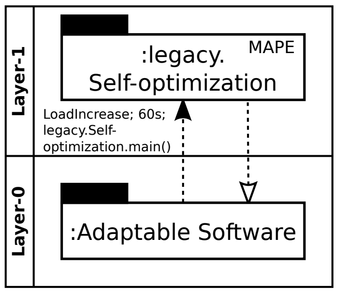

Layer-1
:legacy. Self-optimization
MAPE
LoadIncrease; 60s; legacy.Self-optimization.main()
Layer-0
:Adaptable Software

Fig. 6.6: EUREMA trigger for a legacy module

However, if it is possible to trigger the legacy module based on events similar to triggering feedback loops in EUREMA (cf. Section 3.3), a trigger for the legacy module can be specified and the EUREMA interpreter controls the triggering of the legacy module. This is exemplified in Figure 6.6 that defines the trigger *LoadIncrease; 60s; legacy.Self-optimization.main()* for the *:legacy.Self-optimization* module. Thus, the EUREMA interpreter triggers the legacy module with a period of *60s* when an event of type *LoadIncrease* occurs by invoking the method *legacy.Self-optimization.main()*. In this case, the interpreter controls the activation of the legacy module, which enables decommissioning the legacy module for migrating the adaptation engine to employ megamodel modules defined by the EUREMA language. This migration can be realized by the capabilities for off-line adaptation provided by EUREMA.

Chapter 7

Execution

In this section, we discuss the execution of feedback loops specified by EUREMA. Therefore, we present the metamodel because it precisely defines the EUREMA modeling language and it is the basis for operationalizing EUREMA models by defining the execution semantics. Finally, we provide basic information about the metamodel and interpreter implementations.

7.1 Metamodel and Execution Semantics

In general, a metamodel defines a modeling language by means of its abstract syntax while for visual modeling a concrete syntax based on the abstract syntax is employed. In this article so far, we used EUREMA's concrete syntax as shown in Figure 3.5 to visually model feedback loops and adaptation engines in FLDs and LDs. In the following, we discuss the underlying metamodel that defines the EUREMA language and that is the basis for executing corresponding models by the EUREMA interpreter.

Conceptually, the EUREMA language covers (1) the specification of adaptation activities, runtime models, and feedback loops (the engineer creates FLDs), (2) the specification of layered architectures of self-adaptive software (the engineer creates LDs), and (3) the execution of EUREMA models (the EUREMA interpreter executes these models). These three aspects can be identified in the metamodel of the EUREMA language depicted in Figure 7.1: the elements shaded light-gray are concerned with (1), the hollow elements with (2), while all of these elements and additionally the elements shaded dark-gray are relevant for (3).

7.1.1 (1) Adaptation Activities, Runtime Models, and Feedback Loops

For this aspect, the core concept is the megamodel as visualized by an FLD, which specifies a complete or partial feedback loop including its runtime models and adaptation activities/operations. Thus, a *Megamodel* contains *Model*s and different kinds of *Operation*s. Operations are linked with each other by *Transition*s defining the control flow between operations. Each Transition connects exactly one *source* to one *target* operation. Operations and transitions are the *Executable* elements of all *MegamodelElement*s.

The different operation types used in a megamodel are the following: at least one *InitialOperation* and one *FinalOperation* are required, which are the megamodel's initial and final states, respectively. A

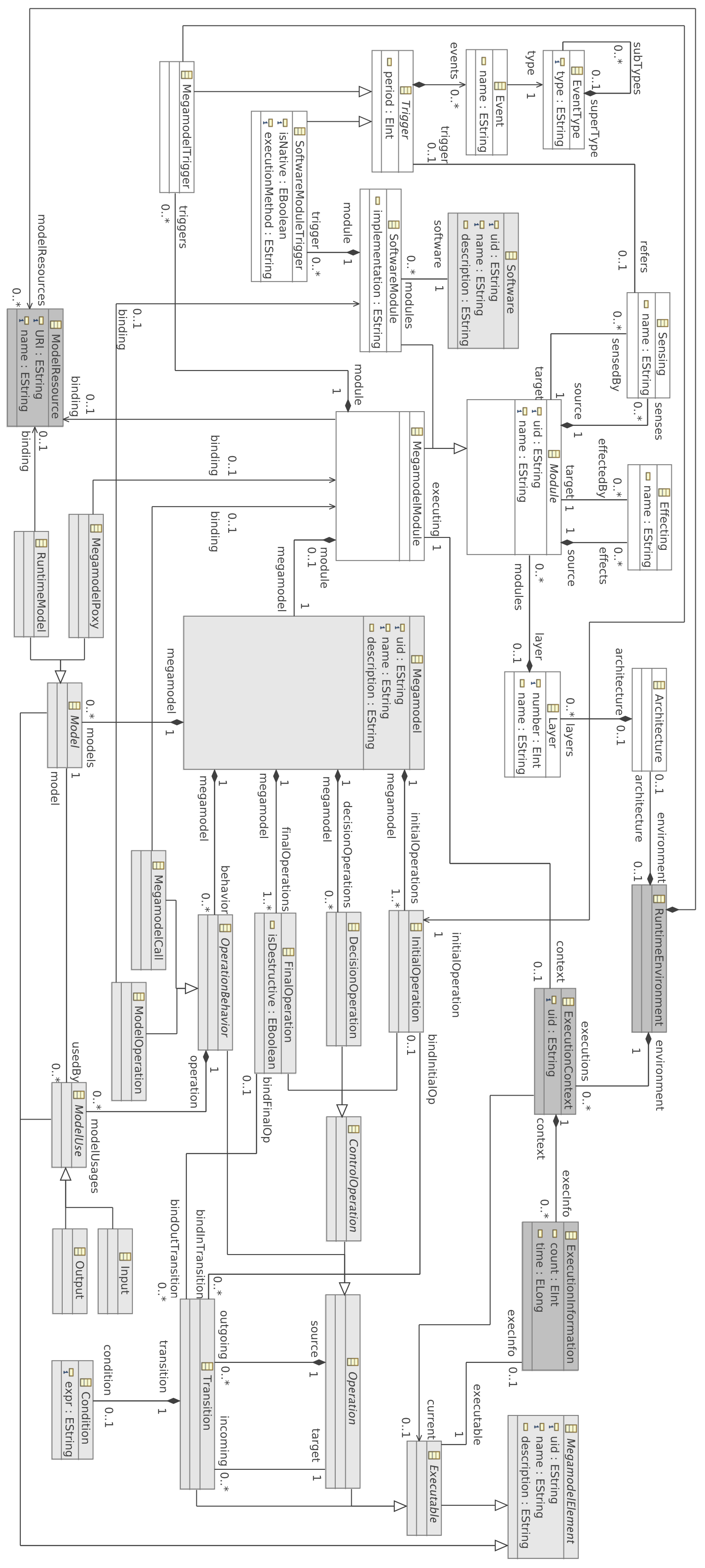


Fig. 7.1: Metamodel of the EUREMA language

FinalOperation can be *destructive* defining that the instance of the megamodel is executed exactly once and afterwards immediately destroyed. *DecisionOperation*s enable the branching of the control flow based on *Condition*s that are annotated to a *DecisionOperation*'s *outgoing OperationTransition*s (see the FLD in Figure 3.6 for an example). The megamodel's actual behavior is defined by *OperationBehavior*s, either *ModelOperation*s or *MegamodelCall*s, that use *Model*s as *Input* or *Output*. *ModelOperation*s are atomic computation units, for which implementations have to be provided (cf. *Software* element). *MegamodelCall*s are the *complex model operations* to invoke a *Megamodel* that specifies either parts of the same feedback loop or a complete and different feedback loop. This enables modular megamodels for describing a feedback loop (cf. Section 3.4) and the coordination of multiple loops (cf. Sections 4.2).

A model used by operations is either a *RuntimeModel* or a *MegamodelProxy*. A *RuntimeModel* can be an arbitrary model as an instance of an arbitrary metamodel, while a *MegamodelProxy* refers to a megamodel as an instance of the EUREMA metamodel. While all examples of megamodels shown as FLDs in this article employ runtime models, an EUREMA megamodel is used as a (reflection) model by another megamodel in layered architectures (cf. *feedbackLoopModel* in the FLD shown in Figure 5.3).

7.1.2 (2) Layered Architectures

The second aspect is concerned with the architecture of a specific instance of a self-adaptive software by means of the employed feedback loops, their relationships to each other and to the adaptable software. This is specified in LDs and addressed by the hollow elements of the EUREMA metamodel (cf. Figure 7.1).

A layered *Architecture* is modeled by multiple *Layer*s each of them containing *Module*s. Modules are concrete instances of feedback loops or software components that are employed within the self-adaptive software. Modules may monitor and adapt each other, which is represented by *Sensing* and *Effecting* relationships between modules.

*SoftwareModule*s instantiating *Software* represent the adaptable software, legacy adaptation components, or implementations of a megamodel's *ModelOperation*s, which are all considered as black boxes in EUREMA (see LDs in Figures 6.5, 6.6, and 3.11 for examples). The *implementation* attribute refers to information on how to initiate an execution of a legacy component or an operation implementation. A *MegamodelModule* is an instance of a complete or partial feedback loop that is specified by a *Megamodel* in an FLD. Thereby, the specification of a feedback loop by means of the FLD is directly used as the instance model of the feedback loop to be executed. In EUREMA, the specification of a feedback loop and the instance model of the feedback loop collapse because feedback loop instances can be individually adapted in layered architectures (cf. Section 5) or by off-line adaptation (cf. Section 6) by changing their individual specifications. Technically, if a feedback loop should be instantiated multiple times, copies of the original specification are created for each instance and wrapped in *MegamodelModule*s to provide an identifiable context for the instance.

When instantiating a feedback loop resp. its megamodel in a module, the dependencies to other modules have to be resolved. Thus, *ModelOperation*s must be bound to *SoftwareModule*s implementing the operations, *MegamodelCall*s, i.e., the complex model operations, must be bound to *MegamodelModule*s as targets of the invocations, and *MegamodelProxies* representing megamodels used within the instantiated feedback loop must be bound to *MegamodelModule*s as concrete megamodels. These bindings are defined in LDs by *use* relationships labeled with the name of the element to be bound (see LDs in Figures 3.10, 3.11, and 5.4 for examples).

Finally, triggers for activating modules are specified in LDs (cf. Section 3.3). A trigger may specify a *period* of activation and *Events* of user-defined *types*, whose occurrences cause the activation. Thereby, a trigger of a module refers to a *Sensing* relationship through which the events can be observed. Thus, the module, for which the trigger is specified, senses another module that emits the corresponding events. Triggers are either *SoftwareModuleTrigger*s for legacy adaptation components as discussed in Section 6, or *MegamodelModuleTrigger*s for feedback loop instances, which point to the initial operation for starting execution (cf. Section 3.3).

7.1.3 (3) Execution

For the last aspect, the execution of EUREMA models by the interpreter, additionally the metamodel elements shaded dark-gray are relevant (cf. Figure 7.1).

The *RuntimeEnvironment* for a layered *Architecture* manages the execution of *MegamodelModule*s by maintaining an *ExecutionContext* for each *MegamodelModule*. This context points to the *current*ly executed *Operation* or *Transition* and it maintains *ExecutionInformation*, especially *count* and *time*, for these *Executable*s. The *time* attribute represents the timestamp when the operation or transition has been executed the last time. For a transition, the *count* attribute reflects the number how often the source operation of the transition has been executed without taking the corresponding transition but another outgoing transition. If the corresponding transition is taken, *count* is reset to *0*. This information is maintained by the interpreter and used in expressing *Condition*s for exclusively branching the control flow, which are evaluated by the interpreter. For example, a counter can be compared with a constant to branch the control flow in a megamodel as it is shown in the FLD in Figure 3.3. For conditions, basic arithmetic and boolean operations on *time* and *count* are currently supported.

The language for expressing conditions is defined by a grammar and it is kept generic to clearly separate the abstraction levels between a megamodel and its contained models and operations. The interpreter works at the level of megamodels and it considers the individual models and operations as black boxes. Since conditions are evaluated by the interpreter, they must not utilize internal concepts of the models or operations. Otherwise, it would require that the interpreter must access such concepts, which would couple the interpreter implementation to the specific implementations of the models or operations. In the end, this would prevent reuse of the interpreter for different self-adaptive systems. However, if more advanced or application-specific conditions are needed, they can be modeled by appropriate *outgoing OperationTransition*s that can be seen as return states of operations and for which *time* and *count* are maintained. In this case, the operation's implementation decide on the state the operation terminates in, while the interpreter may further branch the control flow based on the *time* and *count* attributes of the return state.

Finally, the *ModelResource*s represent the materialized models and megamodels used in the runtime environment from a technical point of view. The *URI* of a resource is the uniform resource identifier pointing to a location, from which the model or megamodel can be loaded.

Concerning the execution semantics of EUREMA models, we distinguish between the behavioral FLDs and structural LDs. An FLD specifies the behavior of a feedback loop as a megamodel that contains a flow of operations working on runtime models. For a specific feedback loop instance, the megamodel is instantiated once and this instance is reused if the feedback loop instance is executed multiple times. A megamodel instance encapsulated in a *MegamodelModule* is not reentrant and therefore, at most a single thread of control is active within the module. Moreover, all operations of a megamodel instance are executed synchronously and thus, also the invocations of other megamodel instances by complex model operations are synchronous. A trigger of a *MegamodelModule* that refers to events emitted from the adaptable software activates the megamodel instance asynchronously. A synchronous activation requires specific sensors of the adaptable software, which need not to be provided by the adaptable software, and it would block the execution of the running adaptable software, which is usually not desirable. However, a trigger of a megamodel module does not activate the megamodel instance while the instance is currently running. Any events that potentially effect the trigger are queued and processed when the instance has terminated execution. An example for such a trigger is discussed in Section 3.3.

In contrast, a trigger of a megamodel module that refers to events emitted from another megamodel module activates the module synchronously. Thus, the execution of the module that emitted the event is blocked until the module activated by the corresponding event has terminated. This scenario has been discussed for layered feedback loops in Section 5.

Thus, concurrent executions of interrelated megamodel modules or individual operations are not supported. This avoids the need of synchronization mechanisms to ensure consistency by restricting concurrency. Therefore, the EUREMA interpreter executes an FLD instance by stepwise executing its operations and following transitions between operations. The control flow is exactly defined as a sequence of operation and it may only be exclusively branched by decision nodes and conditions. When executing an operation, the interpreter provides the models that are used as input and it maintains the models used as output by the operation. Thus, any model being the output of one operation can be the input of another operation. Overall, this results in simple execution semantics of EUREMA models and in a lightweight interpreter that just has to cope with the general concepts of a megamodel, namely operations and models.

The LDs provided by the EUREMA language are not executable like FLDs since they are structural diagrams and specify the architecture instance of the self-adaptive software. Thus, an LD can be considered as a reflection model of the overall self-adaptive software that focuses on the employed modules and the relationships between modules. The EUREMA interpreter maintains the LD of the corresponding self-adaptive software at runtime to observe and adapt the adaptation engine by means of off-line adaptation (cf. Section 6). Thus, the EUREMA interpreter realizes a causal connection between the LD and the adaptation engine. This is similar to model-based adaptation of the adaptable software where an architectural runtime model of the software is used as the basis for reconfiguration.

7.2 Metamodel and Interpreter Implementation

The EUREMA metamodel and its interpreter have been developed with the *Eclipse Modeling Framework* (EMF). Figure 7.1 shows the complete metamodel we use in the implementation. The stereotypes of operation and model elements in FLDs, the labels of model usage elements in FLDs, and the labels of module elements in LDs are not directly supported by the metamodel because they do not influence the execution semantics. Thus, they are not relevant for the EUREMA interpreter. However, they are introduced in the EUREMA language by using a profile mechanism. The language for expression conditions to exclusively branch the control flow in FLDs is defined by a grammar and implemented with the *Java Compiler Compiler* (JavaCC).

Our interpreter implementation only relies on EMF and it may run standalone and decoupled from the Eclipse workbench. Nevertheless, the interpreter provides full support for user-defined, EMF-based runtime models used within feedback loops. For example, the interpreter manages the handling of runtime models as input or output of model operation executions. Concerning EUREMA models, the interpreter currently provides full support for executing FLDs and partial support for maintaining an LD. The partial support enables the execution of multiple and layered feedback loops as defined in an LD, but so far, it does not support the dynamic adaptation of the layered architecture by means of changing the LD at runtime. Currently, we are addressing this aspect by integrating the LD as an explicit runtime model into the interpreter in order to completely support off-line adaptation.

Chapter 8

Discussion and Evaluation

To evaluate the EUREMA approach for executable runtime megamodels specifying and executing adaptation engines for self-adaptive software, we first discuss EUREMA's coverage of the requirements we identified in Section 2. Then, we outline the benefits of EUREMA that are leveraged by the application of MDE techniques and we apply the EUREMA language to model example approaches from the literature. The latter demonstrates that the language is expressive enough to capture state-of-the-art approaches to self-adaptive software and their specific variants of feedback loops and runtime models. Finally, we report about experiences concerning the runtime characteristics of the EUREMA interpreter.

8.1 Requirements Coverage

In this section, we discuss the extent EUREMA fulfills the requirements for self-adaptive software that we presented in Section 2. Thereby, we also discuss details concerning related work for modeling languages that could not have been addressed in Section 1.1 due to the missing introduction of the EUREMA concepts.

By the FLDs, EUREMA supports almost all of the requirements for feedback loops that we identified in Section 2.1. The feedback loops are explicitly modeled according to the abstract scheme established by the *MAPE-K* blueprint that has been refined to model individual adaptation activities and runtime models. This makes the feedback loops visible in the design and analysis of self-adaptive software, which enables the modeling and evaluation of alternative designs as, e.g., discussed for variability of feedback loops in Section 3.4. FLDs further support the modeling of *intra-loop coordination* by defining the control flow between adaptation activities and the usage of shared runtime models by the activities. To execute feedback loops as specified by FLDs, triggers are specified and assigned to concrete feedback loops in LDs. Thus, when modeling the architecture of a concrete self-adaptive software in an LD, EUREMA addresses *triggering conditions* as these condition might be specific to the concrete adaptable software and employed feedback loops.

Moreover, multiple feedback loops and the *inter-loop coordination* of these loops are supported by EUREMA. Individual feedback loops and their coordination are modeled by FLDs, while the LDs reflect the abstract sense, effect, and use relationships between these loops. EUREMA addresses the *execution* of feedback loops by providing an executable modeling language and an interpreter for this language (cf. Section 7), while supporting reflective feedback loops (cf. Section 5).

However, we are currently not addressing *concurrency* of feedback loops and their adaptation activities, and the *incremental* execution of adaptation activities in the sense that all activities of a loop continuously process a stream of events running through the loop. Thus, loops and activities are executed sequentially and stepwise, i.e., one after the other. Moreover, EUREMA is currently restricted to non-distributed adaptation engines while the engine may run on a different node than the adaptable software. Thus, the requirement of *distributing* feedback loops is not fulfilled by EUREMA.

EUREMA models as visualized by FLDs and LDs do not realize themselves the runtime models used as knowledge within feedback loops, but they support the related requirements identified in Section 2.2. The FLDs capture the different runtime models employed in a feedback loop and how they are used by adaptation activities. Thereby, the EUREMA language does not restrict the kinds of runtime models by means of their metamodels as well as purpose. For example, the purpose of runtime models is represented by stereotypes that are not part of the core EUREMA language as defined by the metamodel in Section 7 but they are part of an additional, customizable profile. Moreover, EUREMA models are kept alive at runtime. Thus, they become runtime models that can be used as reflection models in FLDs. Moreover, EUREMA runtime models encapsulated in megamodel modules and the relationships between these modules are captured in LDs. In particular, the use of an EUREMA model as a reflection model is captured when layering feedback loops (cf. Section 5).

Concerning the requirements for sensors and effectors as well as the monitor and execute activities discussed in Section 2.3, which are relevant for connecting a feedback loop to an adaptable subsystem, EUREMA abstracts from sensor and effector details to avoid its coupling to a specific technology, platform, or type of the adaptable software. Therefore, EUREMA proposes the explicit modeling of monitor and execute activities, whose implementations have to cope with the sensor and effector details. Sensors details are only revealed by sensor events used for specifying triggering conditions of feedback loops. Thus, EUREMA models and their adaptability support static as well as dynamic instrumentation if this is properly reflected by the monitor and execute activities, their runtime models, and implementations. The EUREMA approach supports both, *parameter* and *structural adaptation* of the adaptable software and feedback loops. For the adaptable software, adaptation is implemented by appropriate adaptation activities and their runtime models, while EUREMA manages the coordination of activities and models to form feedback loops. For adaptable feedback loops, adaptation refers to the concepts of the EUREMA language when using EUREMA models as reflection models. Thus, by changing the structure or parameters of EUREMA models, adaptation of feedback loops is realized (cf. Section 5).

Adaptable feedback loops are crucial for layered architectures (cf. Section 2.4), where feedback loops operate at different layers. EUREMA supports the individual specification of feedback loops at different layers by FLDs, while an LD defines the overall architecture by structuring the feedback loops in different layers. Moreover, EUREMA supports *declarative and procedural reflection* of feedback loops, either by employing user-defined reflection models (cf. Section 5.1) or EUREMA models as visualized by FLDs (cf. Section 5.2) as reflection models. Overall, this enables adaptive control schemes and hierarchical control architectures without restricting the number of layers or feedback loops.

Finally, based on a layered architecture, EUREMA supports *off-line adaptation* in co-existence to self-adaptation as required in Section 2.5. EUREMA models are used to specify the enactment of adaptation that has been analyzed and planned off-line. Then, the concepts of the layered architecture enables the integration of these models to the runtime environment, where these models are executed by the interpreter to enact the adaptation (cf. Section 6).

To summarize, EUREMA covers almost all of the requirements identified in Section 2 by providing an MDE solution that employs executable runtime megamodels for specifying and executing feedback loops and for reflecting and adapting feedback loops in layered architectures or by off-line adaptation. Thereby, the megamodels explicitly capture the runtime models used in the adaptation engine.

In particular, the language design of EUREMA leverages support for the requirements. We briefly discuss major language design decisions in EUREMA with respect to standard modeling languages. The

EUREMA language is specific for the development of adaptation engines for self-adaptive software, but it is based on general modeling concepts for structural and behavioral diagrams. The EUREMA language for FLDs shares characteristics with *Activities* of the *UML* [Object Management Group 2011]. Both languages provide behavioral diagrams with respect to modeling flows of actions (in UML) or operations (in EUREMA). However, in contrast to EUREMA, UML does not provide megamodel concepts as first class entities, like a model being itself an element in another model. This is required to explicitly capture the runtime models used within a feedback loop and in particular to capture the nesting of EUREMA models, i.e., the use of EUREMA models as reflection models within another EUREMA model.

Moreover, the EUREMA language for LDs borrows concepts from UML concerning structural diagrams, in particular *UML Packages* and *UML Objects* [Object Management Group 2011]. However, EUREMA employs concepts specific for self-adaptive software similar to domain-specific languages in general. Thus, an LD in EUREMA describes all modules within an adaptation engine and how they are related to each other and to the adaptable software by means of use, sense, and effect relationships. Finally, the language for LDs has similarities with the UML profile we proposed in [Hebig et al. 2010]. This UML profile focuses on capturing interferences between feedback loops. In contrast, LDs specify the operational use, sense, and effect relationships between modules, while interferences between modules are addressed by explicitly modeling inter-loop coordination.

8.2 Application of MDE Techniques in EUREMA

By adopting an MDE approach, EUREMA aims at leveraging benefits of MDE to the runtime environment for self-adaptation. On the one hand, as discussed in the previous section, EUREMA exploits MDE principles by means of its modeling language, runtime megamodels, and interpreter to address the requirements for self-adaptive software.

On the other hand, EUREMA makes the runtime models used as knowledge within feedback loops explicit. This additionally leverages MDE techniques at runtime to perform individual adaptation activities within a feedback loop. Moreover, the EUREMA modeling language targets a reasonable abstraction level similar to the level of software architectures. Thus, adaptation activities are considered as abstract model operations, which enables the integration and reuse of existing MDE techniques and implementations for realizing and performing these operations resp. adaptation activities.

For example, in our previous work [Vogel et al. 2009, 2010; Vogel and Giese 2010], we employed an existing model synchronization engine to maintain an architectural runtime model reflecting the adaptable software, and an *Object Constraint Language* (OCL) engine to check architectural constraints on this model. Such engines can be considered as reusable implementations for adaptation activities. Thus, the monitor and execute activities may utilize a model synchronization engine at runtime, and the analyze activity an OCL engine. This exemplifies that the development efforts for implementing adaptation activities can be reduced by integrating and reusing existing engines from MDE. Moreover, since such engines are generic and they completely externalize the user-defined inputs in models, like OCL expressions, these models become runtime models, which are then made explicit in FLDs and amenable for adaptation by EUREMA. For example, the OCL expressions can be dynamically adapted without having to change the OCL engine. This potentially simplifies the development of adaptable feedback loops.

Summing up, EUREMA directly exploits MDE principles for specifying, executing, and adapting feedback loops, while it enables engineers to exploit MDE principles for implementing individual adaptation activities that are modeled as model operations in FLDs.

8.3 Application of the EUREMA Language

In the following, we describe three approaches from literature using the EUREMA language. The first approach, *Rainbow* [Garlan et al. 2004], is based on architecture description language (ADL) techniques, and the second one, *DiVA* [Morin et al. 2009a] on MDE techniques. Nevertheless, our language can capture both approaches and the techniques they apply. Both approaches employ exactly one feedback loop, while the third approach, *PLASMA* [Tajalli et al. 2010], supports layered feedback loops.

8.3.1 Rainbow

The *Rainbow* framework [Garlan et al. 2004; Cheng 2008] is an architecture-based approach to self-adaptation. Its major goal is the cost-effective development of self-adaptive software by providing a reusable infrastructure for adaptation engines. The infrastructure provides several customization options to address specifics of the adaptable software when developing the engine. However, it prescribes the architecture of an adaptation engine by supporting exactly one feedback loop as shown in the LD in Figure 8.2 and whose adaptation activities are structured in a predefined way. We modeled this predefined feedback loop in the FLD depicted in Figure 8.1. In contrast to this diagram, the models describing Rainbow in [Garlan et al. 2004; Cheng 2008] provide architectural views that do not make the runtime models used within the framework explicit.

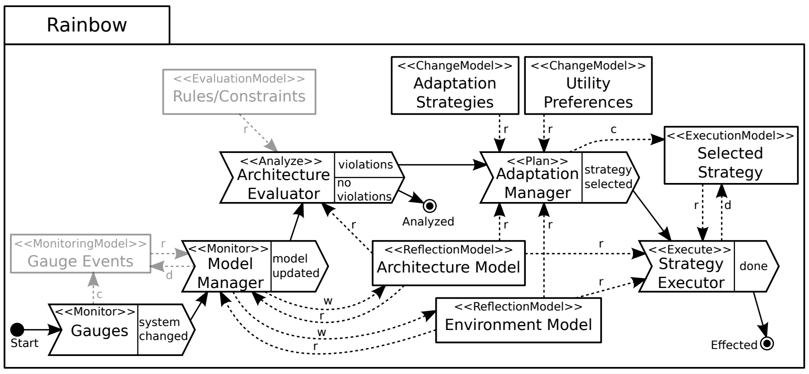


Fig. 8.1: FLD for *Rainbow*

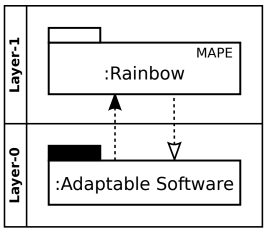


Fig. 8.2: LD for *Rainbow*

For monitoring, *Gauges* abstract from the sensors of the adaptable software to collect data, like average response times, relevant for concerns that should be handled by the self-adaptation mechanism. Gauges notify the *Model Manager* about changes in the adaptable 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 running system or the resource utilization have changed, or the environment model if resources are added or removed from the system. In Figure 8.1, the gauge events are reflected by a gray-shaded element since Rainbow does not explicitly reflect them in a model.

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 these rules and constraints visible in the feedback loop's structure, we depict them by a gray-shaded element. Rules or constraints check, e.g., 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 adaptation. Therefore, based on given *Utility Preferences* among multiple concerns and the current reflection models, a *Selected Strategy* from the repertoire *Adaptation Strategies* is chosen. Strategies are similar to event-condition-action 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 model and the environment model to identify software elements or resources, respectively, which are referenced by the strategy to be executed. Changes of the adaptable software caused by executing the strategy are reflected in the architecture model by the next run of the monitoring activity when the feedback loop is executed again.

The Rainbow framework is based on the ADL *Acme* to describe the architecture and environment models. Rules and constraints as part of the architecture model are specified by *Acme* predicates in a first-order predicate logic. Adaptation strategies and utility preferences are defined in *Stitch* [Cheng and Garlan 2012]. 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 ADL and Stitch models explicit in the architectural design.

8.3.2 DiVA

The goal of the *DiVA* (Dynamic Variability in complex, Adaptive systems) project is to manage dynamic variability in adaptive systems. Therefore, a model-driven approach to specify and execute self-adaptive software has been proposed [Morin et al. 2008, 2009a,b]. MDE techniques and runtime models drive DiVA's feedback loop, which makes this approach an interesting candidate for our modeling approach. Likewise to Rainbow, DiVA employs a single feedback loop as modeled in the LD shown in Figure 8.4. We modeled this feedback loop by the FLD as depicted in Figure 8.3.

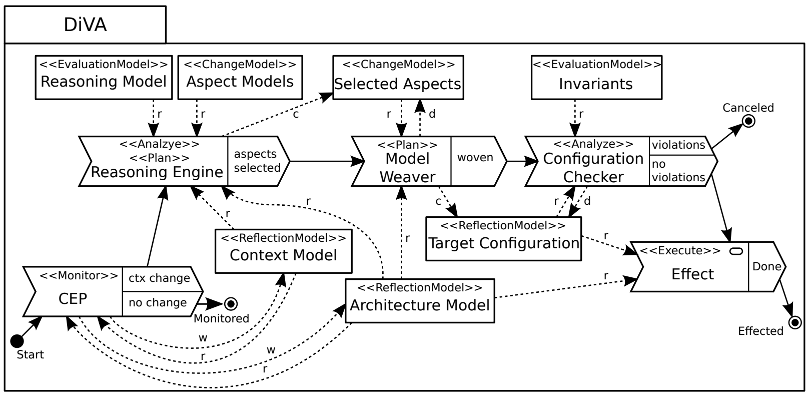

DiVA
<<EvaluationModel>> Reasoning Model
<<ChangeModel>> Aspect Models
<<ChangeModel>> Selected Aspects
<<EvaluationModel>> Invariants
<<Analzye>> <<Plan>> Reasoning Engine
aspects selected
<<Plan>> Model Weaver
woven
<<Analyze>> Configuration Checker
violations
no violations
Canceled
<<ReflectionModel>> Context Model
<<ReflectionModel>> Target Configuration
<<ReflectionModel>> Architecture Model
<<Execute>> Effect
Done
Effected
<<Monitor>> CEP
ctx change
no change
Monitored
Start
r
c
d
w

Fig. 8.3: FLD for *DiVA*

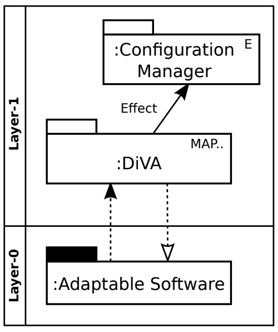


Fig. 8.4: LD for *DiVA*

In DiVA, event-driven sensors are used to monitor the adaptable software and its operational context. A complex event processor (*CEP*) analyzes and filters sensor events to update the *Architecture Model* and the *Context Model* that reflect the adaptable software and the context, respectively. Since 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 by means of features. The reasoning model specifies the rule- or goal-based analysis mechanism to determine which aspects should be activated or de-activated on the current configuration. These *Selected Aspects* are woven or removed by the *Model Weaver* from the architecture model to actually obtain the *Target Configuration* described in a newly created model. Before enacting this target configuration, the *Configuration Checker* validates it by evaluating *Invariants*. 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* as defined in the

FLD shown in Figure 8.5. This invocation executes the adaptation by moving the adaptable software from the current configuration reflected by the architecture model to the target configuration.

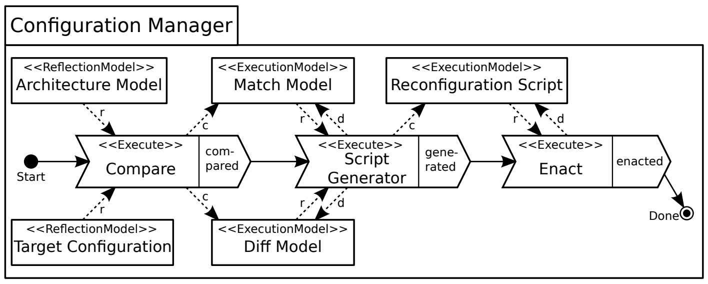

Configuration Manager
<<ReflectionModel>> Architecture Model
<<ExecutionModel>> Match Model
<<ExecutionModel>> Reconfiguration Script
<<Execute>> Compare
compared
<<Execute>> Script Generator
generated
<<Execute>> Enact
enacted
Start
Done
<<ReflectionModel>> Target Configuration
<<ExecutionModel>> Diff Model
r
c
d

Fig. 8.5: FLD for the *Configuration Manager* of DiVA

To enact the target configuration on the adaptable software, the *Compare* activity compares the architecture model with the target configuration to obtain a *Match Model* and a *Diff Model* (cf. Figure 8.5). These model 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 adaptable software.

Modeling DiVA with the EUREMA language shows that the language is able to capture feedback loops that are driven by MDE principles and techniques in contrast to feedback loops based on ADL techniques, like in Rainbow. Since both approaches just employ a single feedback loop, we model in the following an approach for layered feedback loops.

8.3.3 PLASMA

The *PLASMA* approach [Tajalli et al. 2010] proposes a three-layer architecture for plan-based adaptation. The corresponding paper focuses on the generation of plans for adapting software applications, while the behavior of the employed feedback loops and runtime models are often not clearly discussed. This complicated the modeling of PLASMA's feedback loops with the EUREMA language and resulted in the following diagrams. The LD shown in Figure 8.7 defines the layered architecture with the adaptable application at the lowest layer. The feedback loop in the middle layer adapts the application and the highest-layer feedback loop (re)generates plans to be executed by the two lower layers.

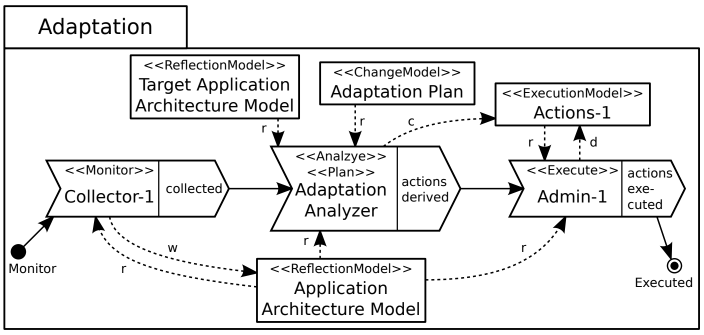

Adaptation
<<ReflectionModel>> Target Application Architecture Model
<<ChangeModel>> Adaptation Plan
<<ExecutionModel>> Actions-1
<<Monitor>> Collector-1
collected
<<Analzye>> <<Plan>> Adaptation Analyzer
actions derived
<<Execute>> Admin-1
actions executed
Monitor
Executed
<<ReflectionModel>> Application Architecture Model
r
c
d
w

Fig. 8.6: FLD for the *Adaptation* layer in PLASMA

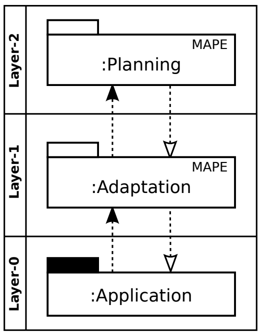


Fig. 8.7: LD for PLASMA

The middle-layer feedback loop, called *Adaptation*, is defined by the FLD in Figure 8.6. The *Collector-1* operation monitors the adaptable application and maintains the *Application Architecture Model* reflecting the application. This reflection model is used by the *Adaptation Analyzer* that executes the

Adaptation Plan provided by the higher-layer feedback loop. This plan defines the adaptation to move the current application architecture to the target architecture as defined in the *Target Application Architecture Model* by the higher-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 executed by the *Admin-1* operation on the adaptable application.

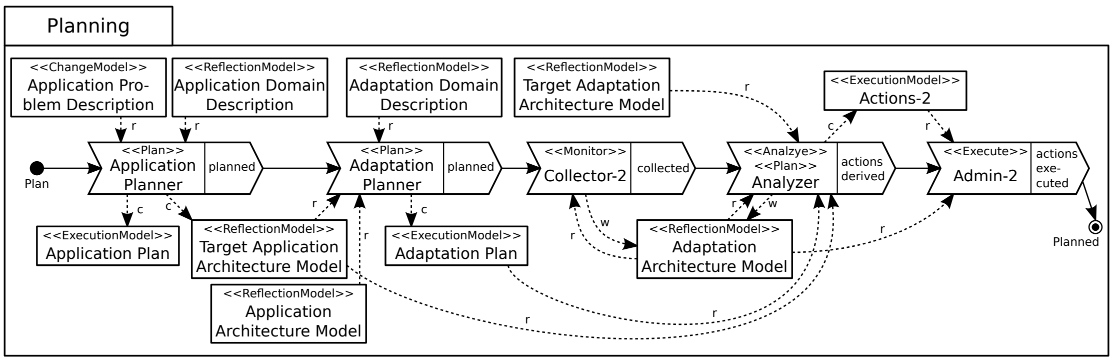

Planning
<<ChangeModel>> Application Problem Description
<<ReflectionModel>> Application Domain Description
<<ReflectionModel>> Adaptation Domain Description
<<ReflectionModel>> Target Adaptation Architecture Model
<<ExecutionModel>> Actions-2
Plan
<<Plan>> Application Planner
planned
<<Plan>> Adaptation Planner
planned
<<Monitor>> Collector-2
collected
<<Analzye>> <<Plan>> Analyzer
actions derived
<<Execute>> Admin-2
actions exe-cuted
Planned
<<ExecutionModel>> Application Plan
<<ReflectionModel>> Target Application Architecture Model
<<ExecutionModel>> Adaptation Plan
<<ReflectionModel>> Adaptation Architecture Model
<<ReflectionModel>> Application Architecture Model
r
c
w

Fig. 8.8: FLD for the *Planning* layer of PLASMA

The higher-layer *Planning* feedback loop defined by the FLD depicted in Figure 8.8 is executed when plans are generated initially or when replanning is required due to changing goals. The *Application Planner* uses a domain model of the application (*Application Domain Description*) and the initial and goal states of the application (*Application Problem Description*), which are both provided by the architect and which are manually changed if system goals change. The planner creates an *Application Plan* to be executed by the application located at the lowest layer and the *Target Application Architecture Model* prescribing the application architecture that is able to execute the application plan.

Moreover, the architect provides the *Target Adaptation Architecture Model* defining the target architecture of the middle-layer feedback loop, which is also reflected in the *Adaptation Domain Description* used by the *Adaptation Planner*. This planner additionally uses the *Application Architecture Model* maintained by the middle-layer feedback loop and the newly created *Target Adaptation Architecture Model* to derive an *Adaptation Plan* for moving the current architecture of the adaptable application to the target architecture. Then, the following operations adapt the middle-layer feedback loop to enable the execution of the generated adaptation plan. The *Collector-2* operation updates the *Adaptation Architecture Model* reflecting the middle-layer feedback loop by monitoring. The *Analyzer* adds the *Adaptation Plan* and the *Target Adaptation Architecture Model* to this model in order to provide them for the middle-layer feedback loop. Moreover, based on the current and target architectures of the middle-layer loop, reconfiguration commands (*Action-2*) are generated to adapt this feedback loop, e.g., to replace the *Adaptation Analyzer* (cf. Figure 8.6) with a version that is able to execute the new adaptation plan. Finally, the *Admin-2* operation adapts the middle-layer feedback loop by executing the reconfiguration commands and by providing the new adaptation plan and target application architecture model to this feedback loop.

Overall, the EUREMA language is able to capture PLASMA's layered architecture and its feedback loops. However, the proper modeling of PLASMA is hard to assess as the runtime models are only implicitly maintained by the employed middleware and their handling is not made explicit by PLASMA. Thus, we derived as far as possible the different runtime models and their usage by operations from informal descriptions in [Tajalli et al. 2010].

8.3.4 Discussion

The results of modeling Rainbow, DiVA, and PLASMA demonstrate that the EUREMA language is expressive enough to capture these state-of-the-art approaches to self-adaptive software and different variants of feedback loops. Thereby, we covered cases where feedback loops are driven by ADL or MDE techniques and where single as well as layered feedback loops are employed. Furthermore, the obtained EUREMA models clearly characterize the feedback loops, adaptation activities, and runtime models of the approaches, which is typically neglected. However, the evidence that the EUREMA language is expressive enough to specify arbitrary feedback loops is limited since we only covered three exemplary approaches. Thus, these findings cannot be generalized to any self-adaptive software.

Moreover, modeling these three frameworks with EUREMA is the initial step for re-engineering and migrating them toward a flexible EUREMA-based solution. Typically, all of these frameworks have a modular design, which makes it straightforward to extract these modules as EUREMA model operations. However, the runtime models are often only implicitly maintained, which makes it difficult to make them explicit as proposed by EUREMA. Nevertheless, if this is feasible, the interplay of the modules and runtime models can be flexibly addressed by EUREMA instead of predefining it by a framework. This potentially leverages EUREMA's benefits for these framework-based approaches.

8.4 Runtime Characteristics of the EUREMA Interpreter

Finally, we evaluate the EUREMA interpreter by discussing its runtime characteristics. Therefore, we conducted experiments to quantify the load and overhead of the interpreter compared to a code-based solution to execute the self-repair feedback loop as defined by the FLDs depicted in Figures 3.8 and 3.6. For the experiments, we considered the case of the self-repair feedback loop, in which each run of the feedback loop always identifies failures. Moreover, a warm-up phase taking place before the actual measurements executes the feedback loop instance more than five times, such that the condition branching the control flow in the FLD depicted in Figure 3.6 is always fulfilled and the *Deep check for failures* operation is executed in each run. Thus, for the measurements, all five operations of the self-repair feedback loop (namely, *Update*, *Check for failures*, *Deep check for failures*, *Repair*, and *Effect*) are executed in each run of the loop instance.

As implementations for these model 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 model 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 period of time, for which it generates load to simulate computations of the model operations.

To evaluate the runtime characteristics 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 as visualized by the FLDs for the self-repair feedback loop. In contrast, the execution is hard-coded 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 experiments we conducted are configured by two parameters. First, the period of time assigned to the mocks defines the internal computation time of the model operations. The same period of time is assigned to all mocks for one experiment and they vary for the different experiments. This results in four groups of experiments, either assigning a period of 0ms, 5ms, 10ms, or 20ms to all mocks. Since the self-repair feedback loop has five model operations, this constitutes a total computation time of either 0ms, 25ms, 50ms, or 100ms for one run of the feedback loop instance. The second parameter is the frequency of consecutive runs of the feedback loop instance, which determines the execution rate of the instance. The frequency is defined by its reciprocal, i.e., the period of time between two consecutive activations of the same feedback loop instance. For each of the four groups of experiments, we varied the period starting from 15ms and doubling it until 960ms. For example, a period of 15ms means that the feedback loop instance is executed every 15ms, which is only feasible if the total computation time of the feedback loop plus the overhead of the code-based solution or the EUREMA interpreter is below 15ms.

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. The results of the experiments[1] are depicted in Figures 8.9 and 8.10 and discussed in the following.

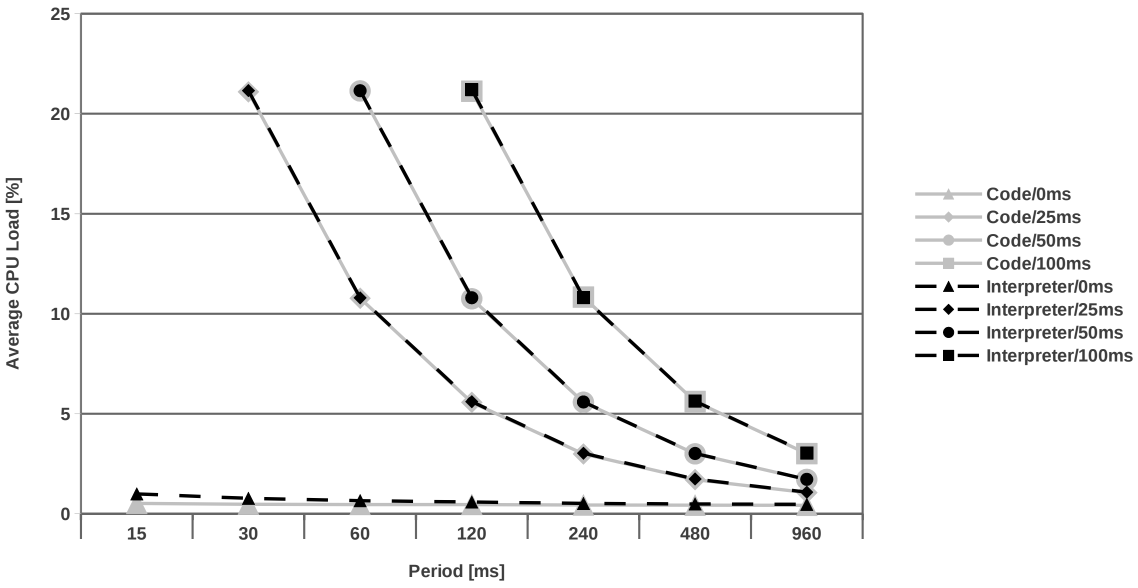


Fig. 8.9: Average CPU Load of the code-based solution (**Code**) and EUREMA (**Interpreter**)

Figure 8.9 visualizes the average 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 running a feedback loop less frequently is supposed to cause less load.

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 0ms (cf. graphs *Code/0ms* and *Interpreter/0ms* in Figure 8.9). However, for the other cases of the computation time (25ms, 50ms, and 100ms), there is no apparent differences between the loads of the code-based solution and the interpreter, and the corresponding graphs for these three cases overlap. Thus, the overhead of the interpreter is noticeable for the hypothetical case that the feedback loop does not perform any computations (computation time is 0ms) and therefore, does not cause any load.

[1] 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 Eclipse Modeling Framework (EMF) Runtime and Tools 2.7.2. The CPU load has been measured by the monitoring capabilities of *Java VisualVM* provided with the Java Development Kit 6 (1.6.0_31).

To further investigate the overhead of interpreting EUREMA models in contrast to the code-based solution, we calculated the overhead as the difference between the average loads of the interpreter and the average loads of the code-based solution. This is depicted in Figure 8.10 for the frequency periods from 60ms to 960ms. We may observe that for all cases the overhead of the EUREMA interpreter with respect to the code based solution is always below 0.2% and tends to decrease with increasing frequency periods. This assumptions is supported by the overhead we predicted (cf. *prediction* graph in Figure 8.10), which is the average overhead based on all measurements for all frequencies and computation times, and normalized for the frequencies.

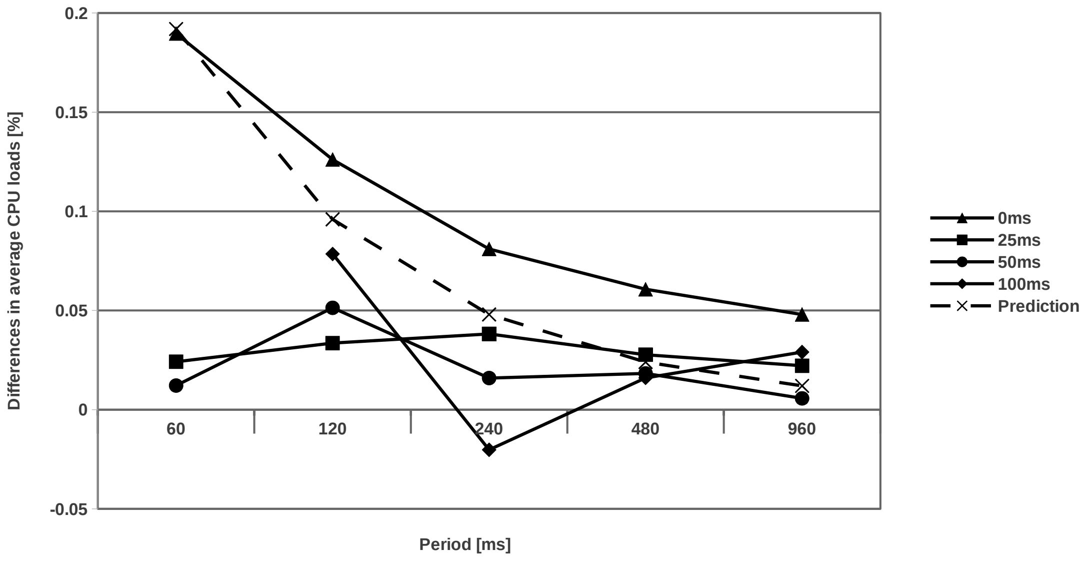


Fig. 8.10: Interpreter overhead by means of the differences in average CPU loads

Summing up, the experiments of quantifying the runtime characteristics of the EUREMA interpreter show 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 EUREMA interpreter. The average of this pure load was for all experiments below 1% (cf. Figure 8.9). Thus, in absolute terms, the EUREMA interpreter works efficiently for the considered case of executing the EUREMA models specifying the self-repair feedback loop. Moreover, employing the EUREMA interpreter and accepting its overhead provides the flexibility to dynamically adapt feedback loops as discussed in Sections 5 and 6.

The validity of the experiments is threatened since we implemented the alternative, code-based solution, such that the comparison between the code-based solution and the EUREMA interpreter needs further investigations. However, we have shown that the interpreter works efficiently in absolute terms by causing a negligible average load (cf. previous paragraph). Another threat of validity is the use of the specific feedback loop as defined in the FLDs depicted in Figures 3.8 and 3.6. Nevertheless, we think that this specific feedback loop is a typical feedback loop since it follows the MAPE principle like the example approaches we modeled in Section 8.3. Moreover, the complexity of the specific feedback loop by means of numbers of model operations and runtime models can be questioned and how the interpreter behaves for larger EUREMA models. Thus, the scalability of the interpreter needs to be further investigated. However, state-of-the-art approaches (cf. Section 8.3) do not employ significantly more complex feedback loops than the self-repair feedback loop with respect to the different sizes of the FLDs.

Chapter 9

Conclusion and Future Work

In this article, we presented EUREMA, a model-driven approach for engineering adaptation engines of self-adaptive software. By *executable runtime megamodels*, EUREMA provides a domain-specific modeling language to specify feedback loops and an interpreter to execute the loops. Thereby, the EUREMA language supports the explicit specification of individual adaptation activities and runtime models as well as complete feedback loops by modular megamodels. Moreover, the interplay and coordination between multiple feedback loops can be captured and layers of feedback loops can be specified and realized to dynamically adjust the adaptation engine. Additionally, the co-existence of off-line adaptation and self-adaptation is supported by uploading megamodel modules, which specify adaptation activities or complete feedback loops, into the adaptation engine and by activating them to execute the off-line adaptation. We evaluated EUREMA by discussing requirements for adaptation engines, the language by modeling state-of-the-art approaches to self-adaptive software, and the interpreter by investigating its runtime characteristics.

In EUREMA, the megamodels specifying feedback loops are kept alive at runtime and they are executed by an interpreter. This provides the required flexibility to cope with changes of the megamodels at runtime when dynamically adjusting the adaptation engine. This is necessary for stacking feedback loops in layered architectures or for executing off-line adaptation. In this context, EUREMA directly supports reflective feedback loops by using its megamodels directly as reflection models. This avoids the development of specific sensors, effectors, and operations to obtain and maintain reflective views of feedback loops. Nevertheless, an engineer may take this effort in order to utilize user-defined reflection models of feedback loops in EUREMA.

In contrast to existing work on self-adaptive software, EUREMA is a seamless approach that covers the specification as well as the execution of adaptation engines by an executable modeling language, corresponding runtime models, and an interpreter. Related approaches on modeling languages for self-adaptive software provide no runtime support for their models, while related work on frameworks for self-adaptive software does not support the explicit modeling of feedback loops. Moreover, frameworks do not provide the flexibility for their users in structuring user-defined adaptation activities to form single or multiple feedback loops in an arbitrary number of layers. In this context, EUREMA does not impose any restriction on engineers when developing adaptation engines.

As future work, we plan to further elaborate the modeling language and the interpreter. With respect to EUREMA models and their execution, we want to study the requirements for adaptation engines that are currently not addressed by EUREMA. Thus, we want to investigate an event-driven, incremental processing of adaptation activities and concurrency in general, however, without tackling the distribution of adaptation engines. Furthermore, we want to investigate the integration of model-based analysis techniques to ease the modularity and reuse of model fragments, like model operations, and for analyzing specifications of feedback loops. Finally, the executability of the EUREMA models makes them amenable for simulation in order to validate and test adaptation engines during the development of self-adaptive software. Simulation may, for example, help in developing and refining feedback loop designs, like adaptation strategies for reconfiguring the adaptable software.

Bibliography

2002. *Special Issue on Adaptive middleware*. Communications of the ACM Series, vol. 45, 6. ACM Press.

Amoui, M., Derakhshanmanesh, M., Ebert, J., and Tahvildari, L. 2012. Achieving dynamic adaptation via management and interpretation of runtime models. *Journal of Systems and Software 85,* 12, 2720–2737.

Andersson, J., Baresi, L., Bencomo, N., de Lemos, R., Gorla, A., Inverardi, P., and Vogel, T. 2013. Software Engineering Processes for Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. LNCS Series, vol. 7475. Springer, 51–75.

Andersson, J., de Lemos, R., Malek, S., and Weyns, D. 2009. Reflecting on self-adaptive software systems. In *Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Computer Society, 38–47.

Astrom, K. J. and Wittenmark, B. 1994. *Adaptive Control* 2nd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Barbero, M., Fabro, M. D., and Bézivin, J. 2007. Traceability and Provenance Issues in Global Model Management. In *Proc. of 3rd Workshop on Traceability (ECMDA-TW)*. 47–55.

Bencomo, N. and Blair, G. 2009. Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems. In *Software Engineering for Self-Adaptive Systems*, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. LNCS Series, vol. 5525. Springer, 183–200.

Bézivin, J., Gerard, S., Muller, P.-A., and Rioux, L. 2003. MDA components: Challenges and Opportunities. In *First Intl. Workshop on Metamodelling for MDA*. 23–41.

Bézivin, J., Jouault, F., and Valduriez, P. 2004. On the Need for Megamodels. In *Proc. of the Workshop on Best Practices for Model-Driven Software Development*.

Blair, G., Bencomo, N., and France, R. B. 2009. Models@run.time. *Computer 42,* 10, 22–27.

Brazier, F. M. T., Kephart, J. O., Parunak, H. V. D., and Huhns, M. N. 2009. Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda. *IEEE Internet Computing 13,* 3, 82–87.

Bruhn, J., Niklaus, C., Vogel, T., and Wirtz, G. 2008. Comprehensive support for management of Enterprise Applications. In *Proc. of the 6th ACS/IEEE Intl. Conference on Computer Systems and Applications (AICCSA)*. IEEE Computer Society, 755–762.

Bruhn, J. and Wirtz, G. 2007. MKernel: a manageable kernel for EJB-based systems. In *Proc. of the 1st Intl. Conference on Autonomic Computing and Communication Systems (Autonomics)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 1–10.

Brun, Y., Serugendo, G. D. M., Gacek, C., Giese, H. M., Kienle, H. M., Litoiu, M., Müller, H. A., Pezzè, M., and Shaw, M. 2009. Engineering Self-Adaptive Systems through Feedback Loops. In *Software Engineering for Self-Adaptive Systems*, B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. LNCS Series, vol. 5525. Springer, 48–70.

Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., and Scheideler, P. 2008. Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. *International Journal on Software Tools for Technology Transfer (STTT) 10,* 3, 207–222.

Burmester, S., Giese, H., and Oberschelp, O. 2004. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In *Proc. of 1st Intl. Conference on Informatics in Control, Automation and Robotics (ICINCO)*, H. Araujo, A. Vieira, J. Braz, B. Encarnacao, and M. Carvalho, Eds. INSTICC Press, 222–229.

Cheng, B. H., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G. D. M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. LNCS Series, vol. 5525. Springer, 1–26.

Cheng, S.-W. 2008. Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA.

Cheng, S.-W. and Garlan, D. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software 85,* 12, 2860–2875.

Cheng, S.-W., Huang, A.-C., Garlan, D., Schmerl, B., and Steenkiste, P. 2004. An Architecture for Coordinating Multiple Self-Management Systems. In *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE Computer Society, 243–252.

de Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Goeschka, K., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Smith, D. B., Sousa, J. P., Tahvildari, L., Wong, K., and Wuttke, J. 2013. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. LNCS Series, vol. 7475. Springer, 1–32.

de Oliveira, F. A., Sharrock, R., and Ledoux, T. 2012. Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing. In *Proc. of the 14th Intl. Conference on Coordination Models and Languages (COORDINATION)*, M. Sirjani, Ed. LNCS Series, vol. 7274. Springer, 29–43.

Ehlers, J. and Hasselbring, W. 2011. A Self-adaptive Monitoring Framework for Component-Based Software Systems. In *Proc. of the 5th European Conference on Software Architecture (ECSA)*, I. Crnkovic, V. Gruhn, and M. Book, Eds. LNCS Series, vol. 6903. Springer, 278–286.

Favre, J.-M. 2005. 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*. Number 04101 in Dagstuhl Seminar Proc. IBFI.

Findeisen, W., Bailey, F., Brdys, M., Malinowski, K., Tatjewski, P., and Wozniak, A. 1980. *Control and Coordination in Hierarchical Systems*. International series on applied systems analysis. J. Wiley.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjorven, E. 2006. Using Architecture Models for Runtime Adaptability. *IEEE Software 23,* 2, 62–70.

France, R. and Rumpe, B. 2007. Model-driven Development of Complex Software: A Research Roadmap. In *Proc. of the ICSE Workshop on the Future of Software Engineering (FOSE)*. IEEE Computer Society, 37–54.

Frey, S., Diaconescu, A., and Demeure, I. M. 2012. Architectural Integration Patterns for Autonomic Management Systems. In *Proc. of the 9th IEEE Intl. Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe)*.

Gacek, C., Giese, H., and Hadar, E. 2008. Friends or Foes? – A Conceptual Analysis of Self-Adaptation and IT Change Management. In *Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 121–128.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer 37,* 10, 46–54.

Gat, E. 1997. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. P. Bonasso, and R. Murphy, Eds. MIT/AAAI Press.

Georgas, J. C., Hoek, A., and Taylor, R. N. 2009. Using Architectural Models to Manage and Visualize Runtime Adaptation. *Computer 42,* 10, 52–60.

Giese, H. and Wagner, R. 2009. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling 8,* 1, 21–43.

Gueye, S. M. K., De Palma, N., and Rutten, E. 2012. Coordinating Energy-aware Administration Loops Using Discrete Control. In *Proc. of the 8th Intl. Conference on Autonomic and Autonomous Systems (ICAS)*. IARIA, 99–106.

Heaven, W., Sykes, D., Magee, J., and Kramer, J. 2009. A Case Study in Goal-Driven Architectural Adaptation. In *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. LNCS Series, vol. 5525. Springer, 109–127.

Hebig, R., Giese, H., and Becker, B. 2010. Making Control Loops Explicit When Architecting Self-Adaptive Systems. In *Proc. of the 2nd Intl. Workshop on Self-Organizing Architectures (SOAR)*. ACM, 21–28.

Hellerstein, J. L., Diao, Y., Parekh, S., and Tilbury, D. M. 2004. *Feedback Control of Computing Systems*. John Wiley & Sons.

Hestermeyer, T., Oberschelp, O., and Giese, H. 2004. Structured Information Processing For Self-optimizing Mechatronic Systems. In *Proc. of 1st Intl. Conference on Informatics in Control, Automation and Robotics (ICINCO)*, H. Araujo, A. Vieira, J. Braz, B. Encarnacao, and M. Carvalho, Eds. INSTICC Press, 230–237.

Isermann, R., Lachmann, K.-H., and Matko, D. 1992. *Adaptive control systems*. Prentice Hall International series in systems and control engineering. Prentice Hall, New York.

Issarny, V., Caporuscio, M., and Georgantas, N. 2007. A Perspective on the Future of Middleware-based Software Engineering. In *Proc. of the ICSE Workshop on the Future of Software Engineering (FOSE)*. IEEE Computer Society, 244–258.

Kephart, J. O., Chan, H., Das, R., Levine, D. W., Tesauro, G., Rawson, F., and Lefurgy, C. 2007. Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs. In *Proc. of the 4th Intl. Conference on Autonomic Computing (ICAC)*. IEEE Computer Society, 24–33.

Kephart, J. O. and Chess, D. 2003. The Vision of Autonomic Computing. *Computer 36,* 1, 41–50.

Kokar, M. M., Baclawski, K., and Eracar, Y. A. 1999. Control Theory-Based Foundations of Self-Controlling Software. *Intelligent Systems and their Applications 14,* 3, 37–45.

Kramer, J. and Magee, J. 2007. Self-Managed Systems: an Architectural Challenge. In *Proc. of the ICSE Workshop on the Future of Software Engineering (FOSE)*. IEEE, 259–268.

Maes, P. 1987. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA)*. ACM, 147–155.

McKinley, P., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. 2004. Composing Adaptive Software. *IEEE Computer 37,* 7, 56–64.

Morin, B., Barais, O., Jézéquel, J.-M., Fleurey, F., and Solberg, A. 2009a. Models@ Run.time to Support Dynamic Adaptation. *Computer 42,* 10, 44–51.

Morin, B., Barais, O., Nain, G., and Jézéquel, J.-M. 2009b. Taming Dynamically Adaptive Systems using models and aspects. In *Proc. of the 31st Intl. Conference on Software Engineering (ICSE)*. IEEE Computer Society, 122–132.

Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., and Blair, G. 2008. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *Proc. of the 11th Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds. LNCS Series, vol. 5301. Springer, 782–796.

Morin, B., Ledoux, T., Hassine, M. B., Chauvel, F., Barais, O., and Jézéquel, J.-M. 2009c. Unifying Runtime Adaptation and Design Evolution. In *Proc. of the 9th IEEE Intl. Conference on Computer and Information Technology (CIT) - Volume 02*. IEEE Computer Society, 104–109.

Müller, H. A., Pezzè, M., and Shaw, M. 2008. Visibility of control in adaptive systems. In *Proc. of the 2nd Intl. Workshop on Ultra-large-scale Software-intensive Systems (ULSSIS)*. ACM, 23–26.

Object Management Group. 2011. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1.

Oreizy, P., Medvidovic, N., and Taylor, R. N. 1998. Architecture-based runtime software evolution. In *Proc. of the 20th Intl. Conference on Software Engineering (ICSE)*. IEEE Computer Society, 177–186.

Oreizy, P., Medvidovic, N., and Taylor, R. N. 2008. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th Intl. Conference on Software Engineering (ICSE)*. ACM, 899–910.

Ramirez, A. J., Cheng, B. H., and McKinley, P. 2010. Adaptive Monitoring of Software Requirements. In *Proc. of the 1st Intl. Workshop on Requirements@Run.Time (RE@RunTime)*. IEEE Computer Society, 41–50.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., and Scholz, U. 2009. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In *Software Engineering for Self-Adaptive Systems*, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. LNCS Series, vol. 5525. Springer, 164–182.

Salehie, M. and Tahvildari, L. 2009. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst. 4,* 2, 1–42.

Schmidt, D., White, J., and Gokhale, A. 2008. Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation. *Software and Systems Modeling 7,* 1, 3–23.

Shaw, M. 1995. Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes 20,* 1, 27–38.

Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., and Mei, H. 2011. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software 84,* 5, 711 – 723.

Sykes, D., Heaven, W., Magee, J., and Kramer, J. 2008. From goals to components: a combined approach to self-management. In *Proc. of the Intl. Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*. ACM, 1–8.

Tajalli, H., Garcia, J., Edwards, G., and Medvidovic, N. 2010. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *Proc. of the IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*. ACM, 467–476.

Villegas, N. and Müller, H. A. 2010. Managing Dynamic Context to Optimize Smart Interactions and Services. In *The Smart Internet - Current Research and Future Applications*, M. Chignell, J. Cordy, J. Ng, and Y. Yesha, Eds. LNCS Series, vol. 6400. Springer, 289–318.

Villegas, N. M., Tamura, G., Müller, H. A., Duchien, L., and Casallas, R. 2013. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems II*, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. LNCS Series, vol. 7475. Springer, 265–293.

Vogel, T. and Giese, H. 2010. Adaptation and Abstract Runtime Models. In *Proc. of the 5th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 39–48.

Vogel, T. and Giese, H. 2012a. A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *Proc. of the 7th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE Computer Society, 129–138.

Vogel, T. and Giese, H. 2012b. Requirements and Assessment of Languages and Frameworks for Adaptation Models. In *Models in Software Engineering (MoDELS 2011 Workshops)*. LNCS Series, vol. 7167. Springer, 167–182.

Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., and Becker, B. 2009. Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems. In *Proc. of the 6th IEEE/ACM Intl. Conference on Autonomic Computing and Communications (ICAC)*. ACM, 67–68.

Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., and Becker, B. 2010. Incremental Model Synchronization for Efficient Run-Time Monitoring. In *Models in Software Engineering (MoDELS 2009 Workshops)*. LNCS Series, vol. 6002. Springer, 124–139.

Vogel, T., Seibel, A., and Giese, H. 2011. The Role of Models and Megamodels at Runtime. In *Models in Software Engineering (MoDELS 2010 Workshops)*. LNCS Series, vol. 6627. Springer, 224–238.

Vromant, P., Weyns, D., Malek, S., and Andersson, J. 2011. On interacting control loops in self-adaptive systems. In *Proc. of the 6th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 202–207.

Weyns, D., Malek, S., and Andersson, J. 2012. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst. 7,* 1, 8:1–8:61.

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., and Goeschka, K. 2013. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. LNCS Series, vol. 7475. Springer, 76–107.

ISBN 978-3-86956-227-8
ISSN 1613-5652