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

Incremental Self-Adaptation of Dynamic Architectures Attaining Optimality and Scalability

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

in der Wissenschaftsdisziplin
Praktische Informatik

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

von
Sona Ghahremani

March 2023

Sona Ghahremani:
Incremental Self-Adaptation of Dynamic Architectures Attaining Optimality and Scalability
March 2023

Advisor:
Prof. Dr. Holger Giese

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-62423
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-624232

ABSTRACT

The landscape of software self-adaptation is shaped in accordance with the need to cost-effectively achieve and maintain (software) quality at runtime and in the face of dynamic operation conditions. Optimization-based solutions perform an exhaustive search in the adaptation space, thus they may provide quality guarantees. However, these solutions render the attainment of optimal adaptation plans time-intensive, thereby hindering scalability. Conversely, deterministic rule-based solutions yield only sub-optimal adaptation decisions, as they are typically bound by design-time assumptions, yet they offer efficient processing and implementation, readability, expressivity of individual rules supporting early verification. Addressing the quality-cost trade-off requires solutions that simultaneously exhibit the scalability and cost-efficiency of rule-based policy formalism and the optimality of optimization-based policy formalism as explicit artifacts for adaptation. Utility functions, i.e., high-level specifications that capture system objectives, support the explicit treatment of quality-cost trade-off. Nevertheless, non-linearities, complex dynamic architectures, black-box models, and runtime uncertainty that makes the prior knowledge obsolete are a few of the sources of uncertainty and subjectivity that render the elicitation of utility non-trivial.

This thesis proposes a twofold solution for incremental self-adaptation of dynamic architectures. First, we introduce Venus, a solution that combines in its design a rule- and an optimization-based formalism enabling optimal and scalable adaptation of dynamic architectures. Venus incorporates rule-like constructs and relies on utility theory for decision-making. Using a graph-based representation of the architecture, Venus captures rules as graph patterns that represent architectural fragments, thus enabling runtime extensibility and, in turn, support for dynamic architectures; the architecture is evaluated by assigning utility values to fragments; pattern-based definition of rules and utility enables incremental computation of changes on the utility that result from rule executions, rather than evaluating the complete architecture, which supports scalability. Second, we introduce HypeZon, a hybrid solution for runtime coordination of multiple off-the-shelf adaptation policies, which typically offer only partial satisfaction of the quality and cost requirements. Realized based on meta-self-aware architectures, HypeZon complements Venus by re-using existing policies at runtime for balancing the quality-cost trade-off.

The twofold solution of this thesis is integrated in an adaptation engine that leverages state- and event-based principles for incremental execution, therefore, is scalable for large and dynamic software architectures with growing size and complexity. The utility elicitation challenge is resolved by defining a methodology to train utility-change prediction models. The thesis addresses the quality-cost trade-off in adaptation of dynamic software architectures via design-time combination (Venus) and runtime coordination (HypeZon) of rule- and optimization-based policy formalisms, while offering supporting mechanisms for optimal, cost-effective, scalable, and robust adaptation. The solutions are evaluated according to a methodology that is obtained based on our systematic literature review of evaluation in self-healing systems; the applicability and effectiveness of the contributions are demonstrated to go beyond the state-of-the-art in coverage of a wide spectrum of the problem space for software self-adaptation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADL	Architecture Description Language
aDT	Average Deployment Time
AI	Artificial Intelligence
ATL	ATLAS Transformation Language
CBR	Case-based Reasoning
CF	Critical Failure
CG	Correspondence Graph
DCG	Discounted Cumulative Gain
DSL	Domain-specific Language
ECA	Event-Condition-Action
EMF	Eclipse Modeling Framework
EU	Expected Utility
FET	Failure Exposure Time
FGS	Failure Group Size
FOL	First Order Logic
GA	Genetic Algorithm
GBM	Gradient Boosting Models

GT	Graph Transformation
IAT	Inter Arrival Time
IID	Independent and Identically Distributed
KPI	Key Performance Indicator
LHS	Left-hand Side
LRA-M	Model-based Learning, Reasoning, and Acting
MADP	Mean Absolute Deviation Percent
MDE	Model-driven Engineering
MDP	Markov Decision Process
MEU	Maximum Expected Utility
MIMO	Multi-input, Multi-output
MPC	Model Predictive Control
mRUBiS	Modular Rice University Bidding System
MTBF	Mean Time Between Failures
NAC	Negative Application Condition
OCL	Object Constraint Language
PI	Performance Issue
pmml	predictive model markup language
POMDP	Partially Observable Markov Decision Processes
QoS	Quality of Service
QVT	Query/View/Transformation
RF	Random Forest
RHS	Right-hand Side
RL	Reinforcement Learning
RMSE	Root Mean Square Error
RRS	Recursive Random Sampling
RTM	Runtime Model
SAM	Similarity Aggregation Metric
SD	Story Diagram
SDM	Story-driven Modeling
SEAMS	Symposium on Software Engineering for Adaptive and Self-Managing Systems
SLA	Service Level Agreement
SLR	Systematic Literature Review
SP	Story Pattern
TGG	Triple Graph Grammar
UML	Unified Modeling Language
XGB	Extreme Gradient Boosting Trees

1 INTRODUCTION

1.1 MOTIVATION

Modern, complex software systems are increasingly required to continue operating in highly dynamic environments, accommodate rapidly changing requirements, and cope with unpredictable operation conditions [112, 367]. While manual oversight and maintenance [see 258, 336] benefits from domain expertise, global problem contexts, and flexible policies, human operators are costly and error-prone. Embedded and low-level mechanisms [see 151, 160] however, are effective and timely for error recovery, but suffer from covering only local scopes. Such solutions are application specific which makes reusability infeasible. In addition, embedded mechanisms to change software are costly to modify in the face of dynamic adaptation objectives, thus integration in contexts that are not known a priori becomes challenging.

The vision of *autonomic computing* and *self-adaptation* emerged as the complexity of the computing systems approached human limits for manual maintenance, development, deployment, management, and evolution [110, 158]. Autonomic computing and self-adaptation seek the reduction of the human involvement in the maintenance/evolution of the computing systems. Both visions share the objectives of automating the adaptation process and shifting the responsibility for performing the adaptation from human to the system itself. More specifically, autonomic computing is generally concerned with automating the management of computing systems based on the high-level objectives obtained from domain experts. Self-adaptation, on the other hand, refers to automating the modification of system behavior and/or structure based on the perception of the system of itself, its environment, and its goals [89]. The aforementioned modifications can be realized from the software engineering perspective. While self-adaptive systems are used in a number of different areas, this thesis focuses only on their application in the software domain, called *self-adaptive software*. Leveraging domain expertise, an automated process of system adaptation provides an end-to-end system perspective allowing for modification of the target system at runtime [252]. The promise of coping with the growing costs, complexity, and diversity of evolving systems via realization of self-adaptation has led to continuously growing research trends on autonomic computing and self-adaptation during the last two decades [275, 293, 332, 342, 426].

The *architecture model* of a software system is an abstract representation of the system as a composition of computational elements and their interconnections [381]. *Architecture-based self-adaptation* of software systems uses *external* mechanisms where the self-adaptive software comprises the adaptable software, the adaptation engine, and an architecture model of the target system to close the control loop of the automated adaptation—see [167, 367]. The state-of-the-art has evolved to support variability and adaptation modeling; this goal is achieved by making use of architecture models at runtime to monitor and drive runtime adaptation at a desired level of abstraction, e.g., [108, 144, 146, 198, 254], as architecture models allow for explicitly capturing the knowledge required to manage software quality. Architectural models are widely recognized as providing separation of concerns between the system behavior and that of its constituent components that renders architecture models appealing for self-adaptation of software systems [93, 426]. Specification of the adaptation logic at the model level yields separation of the logic from the code and benefits the loose coupling of the adaptation policies and the main system functionalities [323].

The use of software architecture as the basis of a control model for self-adaptation holds a number of advantages; a rich body of work on architecture trade-off analysis techniques, used at design-time, facilitates runtime self-adaptation. As an abstract model, an architecture model exposes important system properties and constraints, provides end-to-end problem contexts, and allows principled and automated adaptations [321, 381]. Considering high-level system design decisions, the model makes system integrity constraints explicit, thereby helping to ensure the validity of a change. An architecture-based solution for self-adaptation cultivates the benefits of an overall software architecture specification so that, despite the changes to the system, the system will remain well-formed with respect to its desired specification, thus the system may preserve the structural and behavioral properties captured by its specification [170].

1.2 PROBLEM STATEMENT

In the following, we first present the scope of the problem that concerns this thesis. We state the elements of the problem as requirements ($\mathcal{R}$) for architecture-based self-adaptation of software systems.The requirements are derived from the literature and summarized in Table 1.1. Next, we present state-of-the-art policy formalisms for software self-adaptation mechanisms and discuss their limitations.

1.2.1 *Requirements for Architecture-based Self-adaption of Software Systems*

The *quality-cost trade-off* in adaptation is a multi-faceted challenge in engineering self-adaptive systems [184]. A self-adaptive system is expected to simultaneously balance multiple software quality objectives while complying to the cost-induced restrictions, e.g., security vs. performance vs. time. The severity of the conflicting objectives however varies for different application domains. In safety-critical systems, for instance, it is essential to maintain certain levels of quality at any expense, as otherwise the system might endangers lives [84, 260]. In other domains, e.g., commercial systems such as *Amazon Web Services* (AWS), the *Service Level Agreement* (SLA) demands a monthly up-time percentage of at least 99.99%[1] in combination with cost minimization. In order to avoid violation of the SLAs, system availability must be obtained constantly. Thus, eliminating runtime anomalies that affect system availability in a timely manner is important. Streaming media services like Netflix are another example where minimizing the streaming latency is critical to satisfying user experience; on the other hand, reducing the maintenance and operation cost enables lower subscription prices which is an influential factor of client experience [319].

Today's complex and dynamic software systems demand adaptations that *fulfill system quality objectives* ($\mathcal{R}1$) in a *cost-effective manner* ($\mathcal{R}2$) [see 93]. In essence, planning an adaptation is a search and optimization process performed over the space of pos-

1 `https://aws.amazon.com/compute/sla/`– accessed 18 March 2023.

sible solutions [404]. Exhaustive search in the possible adaptation space can provide quality guarantees but renders attaining optimal adaptation plans time-intensive [361]. Challenges in addressing the trade-off include provisioning of efficient and *scalable* solutions ($\mathcal{R}3$) to cope with the complexity caused by the possible combinatorial explosions of adaptive system artifacts such as configurations, variant dependencies, and adaptation options [145].

The imbalance between quality and cost objectives increases significantly as software systems grow in size and complexity, creating a relatively larger search space for adaptation solutions that satisfy multiple quality objectives [72]. To design a self-adaptive software that is capable of maintaining multiple quality objectives, cost-effectively and at runtime, models that explicitly capture system objectives and requirements, as well as the knowledge representing system properties and its relevant context are required [183]. The models must be kept representative at runtime. In order to enhance runtime reasoning of multiple objectives during self-adaptation, *trade-off policies must be made explicit* ($\mathcal{R}4$) [see 351].

Software systems with *dynamic architectures* allow for modifications of their architecture during their execution [303]. Designing architectures that exhibit a good trade-off between multiple quality attributes is effortful [53, 381]; adapting dynamic software architectures at runtime, on the other hand, imposes further challenges in the face of growing software complexity, changing requirements, highly dynamic environments, and unpredictable operating conditions [146, 396]. Additionally, monetary, time, and resource constraints further perplex the adaptation, yielding possibly conflicting objectives for the system [93]. An adaptation mechanism is required to *manage dynamic software architectures* ($\mathcal{R}5$) [53, 214]. This implies that the self-adapting system should be able to dynamically create and operationalize adaptation plans that satisfy system objectives.

As stated in ($\mathcal{R}4$), the trade-off policies should be made explicit to enable dynamic adaptation [93]. *Utility functions*, i.e., high-level specifications that capture system objectives [146], support the explicit treatment of quality-cost trade-off. Specifying utility functions however is non-trivial and may require several iterations [143, 259]. Constructing a utility function that represents system preferences is a challenging task due to sources of uncertainty and subjectivity, e.g., non-linearities, complex dynamic architectures, and black-box system models [254, 383]. Addressing these challenges requires specialized domain knowledge [333], which makes utility elicitation burdensome [79]. Additionally, highly dynamic operation conditions of software systems rapidly renders the prior knowledge obsolete [117].

The state-of-the-art proposes the common practice of construing the system utility by hand-picking the features of the architecture [see 232] or reducing the number of the features by exploring inter-feature relationships [129]. The runtime uncertainty hinders the expectation to have an omniscient decision-maker that knows user/system[2] preferences at any given time and under different operation conditions [117]. An ideal solution should be able to *address the problem of initially unknown runtime knowledge to support adaptation* ($\mathcal{R}6$). In other words, the approach should preclude the designers from having a detailed knowledge of the user or system preferences for every specific adaptation decisions necessary to restore the system to the desired states. Systematic

2 A user of a system is another system (physical or human) which interacts with the former [281].

Table 1.1: Requirements for architecture-based self-adaption of software systems.

Req.	Description
$\mathcal{R}1$	Solution satisfies system quality objectives at a desirable level
$\mathcal{R}2$	Solution is cost-effective
$\mathcal{R}3$	Solution is scalable
$\mathcal{R}4$	Solution explicitly captures quality-cost trade-off policies
$\mathcal{R}5$	Solution supports dynamic architectures
$\mathcal{R}6$	Solution addresses problem of initially unknown runtime knowledge of user and system preferences

acquisition of utility prediction models for black-box systems where prior knowledge is missing or becomes obsolete at runtime should be supported.

All the requirements discussed in this section in provision of the architecture-based self-adaptation solutions are listed in Table 1.1. We will use them in this thesis to discuss our approach to engineering self-adaptive software and contrast it to related work.

1.2.2 *State-of-the-art Policy Formalisms for Self-adaptation*

Adaptation policies govern the decision-making process during planning and are used to express and operationalize the high-level objectives of the system [214]. Two main families of formalism have been proposed to capture adaptation policies [see 253]: deterministic *Event-Condition-Action* (ECA) rules [244] and optimization-based approaches leveraging utility functions or goal models [254].

Rule-based solutions. Self-adaptation solutions that use ECA rules to formalize adaptation policies relate context events to reconfiguration actions via rules [108, 163]. The main strength of the rule-based approaches are twofold: the readability and expressivity of each individual rule and efficient processing and implementation [243]. These approaches benefit from using well-known policy definition formalism, allowing them to be efficiently implemented, while supporting early verification [144].

Limitation of rule-based solutions. Designing a self-adaptive system where quality trade-off policies are deterministically defined makes rule-based policies implicit artifacts of system design [241, 321]. While adaptation solutions defined based on ECA rules may explicitly consider quality objectives during software design, the resulting trade-off decisions are not exposed as first-class (operational) entities in software functionality. The implicit nature of hard-wired policies complicates the reasoning about satisfying objectives, for there is often no explicit treatment of the objectives. Moreover, adding new quality of concern may be difficult since it requires modifications that affect many aspects of the system. Rule-based policies are bound by design-time assumptions, thus have a single context of use which may become obsolete when system preferences change due to evolving business needs [see 90, 113, 278]. While reactive, condition-based solutions for adaptation deliver adaptation plans timely, i.e., in a cost-effective manner (see $\mathcal{R}2$ in Table 1.1), they often fail to find the optimal solutions, thus cannot guarantee achievement of $\mathcal{R}1$. Additionally, rule-based solutions may encounter scalability problems related to the management and validation of large sets of rules when context and variability spaces grow, thus may not satisfy $\mathcal{R}3$.

Optimization-based solutions. In optimization-based approaches, the adaptation policy is expressed as high-level system objectives through utility functions or goal models [see 146, 198]. Utility function policies can be viewed as generalizations of goal policies [253]. A utility function evaluates each system configuration in terms of satisfying system objectives. The optimization aims at identifying configurations yielding the highest utility values. The main benefit of the optimization-based approaches is the abstraction they provide through properties allowing use of relatively simpler adaptation actions. In addition, utility functions are an efficient way to determine how well-suited a configuration is depending on the runtime context.

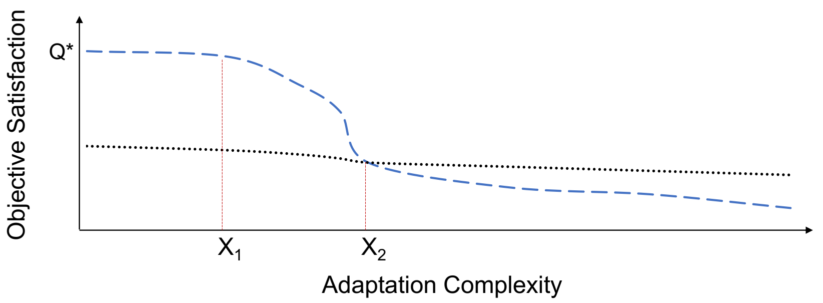
Limitation of optimization-based solutions. Optimization-based policies do not explicitly describe all the possible configurations of the system a priori, instead, they often perform an exhaustive search in the possible adaptation space. Thus, the optimization process hinders scalability for large configuration spaces. In the presence of multiple system objectives, scalability is further impeded by employing complex utility functions, as used in constraint solver-based approaches [145]—see $\mathcal{R}3$ in Table 1.1. These solutions solve an optimization problem before every adaptation which may render attaining optimal adaptation plans time-intensive [361]. As stated in $\mathcal{R}2$ (see Table 1.1), a solution for self-adaptation should be capable of coping with the continuous change in a cost-effective manner, i.e., scale with the complexity of the adaptation space. Furthermore, as discussed in the context of $\mathcal{R}6$, due to non-linearities, complex dynamic architectures, black-box models, and runtime uncertainty, elicitation of utility functions is non-trivial.

Hybrid solutions. The state-of-the-art offers paradigm for compromise, namely hybrid adaptation, where an ensemble of multiple adaptation policies, combined at different levels, steer the adaptation—see [62, 90, 326, 351]. *Hybrid planning* refers to solving a planning problem by combining multiple planning approaches/algorithms to benefit from their combined strengths. However, the end result inherits the properties of its constituents including their limitations.

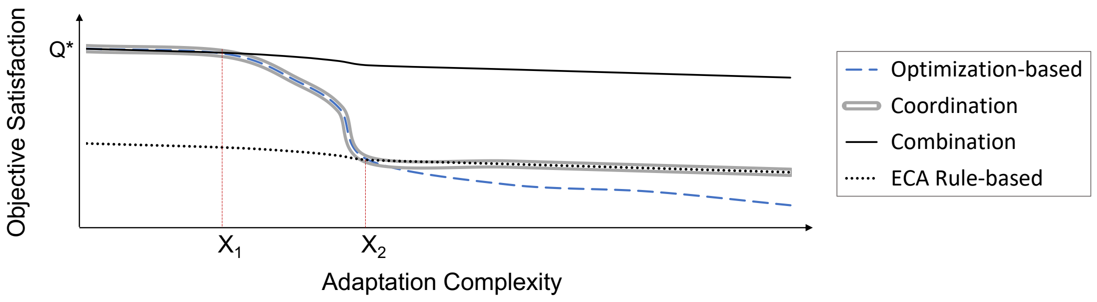
Summary. Various research provide tangible evidence that constructing an individual or hybrid automated adaptation mechanism that systematically recognize and handle cost-effective adaptations of large, dynamic software systems with growing complexity remains an open challenge where current approaches offer only partial solutions with limited applicability [8, 137, 297, 326, 365, 367, 426]. Existing approaches are restricted in being concerned with fixed set of quality attributes, captured either as ECA rules or hand-crafted objective (utility) functions, for which desired values are defined a priori. They lack flexibility to efficiently cater to dynamic architectures and offer only partial support for trade-off across multiple objectives and varying contexts. A systematic treatment in addressing the increasing complexity of the adaptation space for dynamic, large, and highly-configurable systems is missing.

1.3 overview of proposed solutions

For self-adaptation of software systems that an individual policy cannot simultaneously satisfy $\mathcal{R}1$ and $\mathcal{R}2$, a compromise between two ends of the spectrum, i.e., rule- and optimization-based approaches, is beneficial. Consequently, a trade-off for large-scale dynamic systems while addressing the complexity-induced combinatorial explosion of the number of context and configuration changes can be achieved [145]. This thesis comprises a twofold solution for software self-adaptation based on (i) design-time *combination* (Venus) and (ii) runtime *coordination* (HypeZon) of rule- and optimization-based formalisms of adaptation policies. The solution contributes to a segment of software self-adaptation problem space where the quality and cost requirements for a solution exceed the ability of an individual adaptation mechanism, to solely, fulfill the objectives. A customized *combination-based* solution may combine policy formalism of the individual solutions in its design to construct a new policy. The resulting solution may leverage the strength of all the constituents, thus outperforming them in terms of cost minimization and quality maximization. On the other hand, the quality-cost trade-offs can be addressed by a *hybrid* self-adaptation mechanism that *coordinates* multiple off-the-shelf approaches that individually perform either cost-effectively ($\mathcal{R}2$) or with high objective satisfaction ($\mathcal{R}1$). The quality and cost of the adaptation carried out by the hybrid solution, is bounded by the ones of its constituents, i.e., a coordination-based solution performs only as good as its best performing constituent policies.



(a) Individual adaptation policies.



(b) Individual adaptation policies against their coordination and combination.

Figure 1.1: A notional representation of a space for self-adaptation solutions in domains with growing complexity (Q* denotes optimal objective satisfaction).

Figure 1.1 shows a notional representation of the solution space for self-adaptations versus growing complexity of the adaptation space. An optimization-based and a rule-based policy constitute the set of individual solutions (top) for whom a coordination-based and a combination-based solution is sketched (bottom). The parts of the x-axis before X_1 represent adaptation complexities where an individual solution, i.e., the optimization-based solution, performs desirably (obtains Q^*). X_1 marks the point in the adaptation space where the individual solutions cannot desirably satisfy the adaptation objectives (below Q^*). X_2 represents a point where the complexity of the adaptation space prevents the optimization-based solution from out-performing the rule-based alternative. The coordination approach is bounded by the performance margin of its constituent solutions (see the switch at X_2), while the combination approach may prevent the performance degradation of the optimization-based solution and exhibit the steady pattern of the rule-based solution. Adaptable software with dynamic architectures is prone to evolution during system execution, thus adaptation complexity is subject to change. This might rapidly render a well-performing solution insufficient—see X_2 in Figure 1.1 where a slight change in the complexity of the adaptation space disqualifies the, until then, best-performing optimization-based solution in comparison to the alternative rule-based solution.

Capturing the complexity of the multi-dimensional adaptation space as points across the x-axis is an oversimplification of the phenomenon. Note that Figure 1.1 is used only for conceptual illustration purposes and does not correspond to any empirical measurements. We replace this chart by an alternative[3] that is based on quantitative experiments in Chapter 9 where we report on a series of experiments during which we maintain all the dimensions of the adaptation space constant and emulate the increasing complexity of the adaptation by step-wise increasing the complexity of the adaptation setup, e.g., increasing system load or architecture size.

The twofold solution of this thesis for self-adaptation of software systems, i.e., VENUS, and HYPEZON, is integrated in an external adaptation engine that implements the *Monitor, Analyze, Plan, Execute, and Knowledge* (MAPE-K) loop—a standard blueprint of a control feedback loop from IBM for an automated cycle of the four activities operating on a shared Knowledge base [215]. The adaptation engine employs utility functions to incrementally evaluate an architectural *Runtime Model* (RTM) [322] of the adaptable software as well as the adaptation plans. We present the technical contributions of this thesis as three main building blocks that collectively constitute our solution for architecture-based adaptation: (i) utility functions for dynamic software architectures; (ii) a solution that combines utility- and rule-based policies to engineer the Analyze and Plan activities of the adaptation loop (VENUS); (iii) a coordination-based hybrid solution for planning software adaptation (HYPEZON)—see Figure 1.2 for an overview.

1.3.1 *Utility Functions for Dynamic Software Architectures*

Leveraging graphs and a graph-based formalism, we capture the software architecture as a graph. Dynamic architectures are evaluated by assigning utility values to fragments of the architecture, i.e., *graph patterns*. We refer to this as *pattern-based utility*. Desired and undesired patterns in the architecture are distinguished by assigning positive and negative values, respectively, introducing the notion of *positive* and *negative architectural utility patterns* that capture the impact of the fragments of the architecture on the utility. Utility functions in general can be obtained in two ways: (i) analytically engineered from domain knowledge or (ii) gradually learned from the observations. In this thesis,

3 See Figure 9.4.

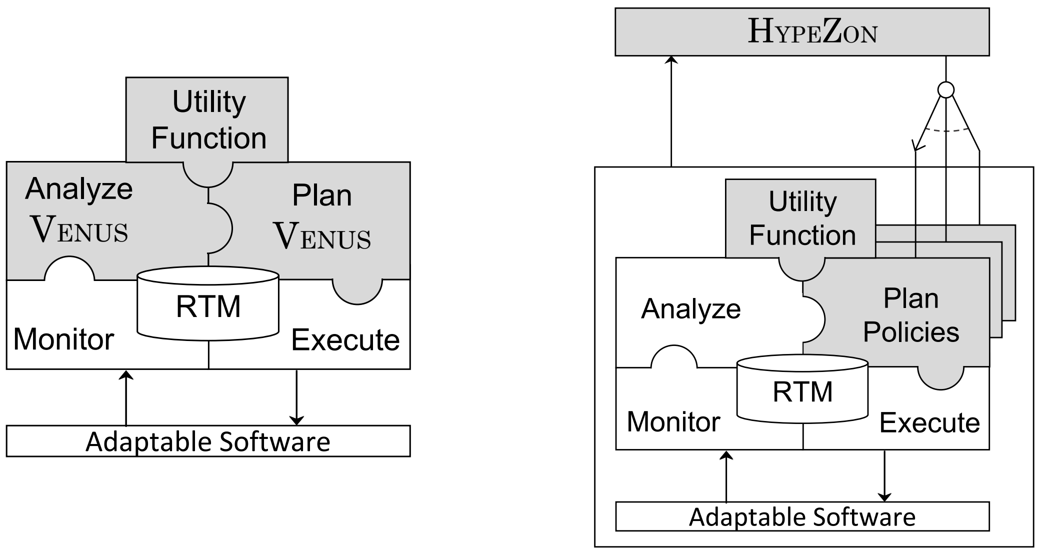


Figure 1.2: Overview of proposed solutions (gray blocks are novel contributions of this thesis).

we present our approach to attain both practices for utility elicitation in the context of dynamic software architectures.

In addition to employing analytically defined utility functions obtained from domain knowledge, this thesis provides a systematic methodology to acquire utility prediction models for black-box software systems or application domains where specialized domain knowledge is not available. We extend the standard machine learning process to systematically train *utility-change* prediction models for rule-based self-adaptive software. Utility-change indicates the changes in utility during an adaptation. The methodology employs multiple learners to empirically construct models for predicting the effects of the adaptation rules on the system utility, when sufficient prior knowledge is missing. The prediction models are trained offline and when the adaptable software is executed under simulated operation conditions. The methodology calls for co-existence of offline training and online execution of the prediction models to maintain accurate predictions of rule applications impact on the system utility.

1.3.2 VENUS*: Combining Utility- and Rule-based Policies*

VENUS is a utility-driVEN rULe-based scheme for engineering software self-adaptation. The scheme combines the common ECA rule- and optimization-based formalisms in its design. Consequently, VENUS exhibits the benefits of its constituents, collectively, while, inevitably, inheriting certain limitations. VENUS employs ECA rules to realize the adaptation loop activities. In addition, the scheme uses utility functions to capture system objectives and steer the adaptation towards optimizing the utility function. VENUS primarily targets the architecture-based self-adaptation of large, dynamic software systems, i.e., resolving runtime issues by dynamically adapting the system architecture. *Model-driven Engineering* (MDE) principles are heavily exploited to support creation and runtime evolution of *causally connected* architectural RTMs—see [148]. Being integrated in a MAPE-K feedback loop, the scheme operates on a causally connected architectural RTM of the system that is captured as a graph. Consequently, *adaptation issues*, i.e., phenomena that trigger adaptation, as well as the condition and action parts of the ECA rules are realized as graph patterns in the architectural RTMs—see Figure 1.2 for an overview.

VENUS is *reactive* and targets a class of self-adaptation problems that are usually identified by symptoms, also known as conditions, e.g., self-healing systems [see 81, 185, 343], where adaptation is only needed if runtime failures occur. The scheme exploits these symptoms, leveraging *event-based principles*, i.e., event-based detection and processing of the changes. VENUS also supports state-based execution of the adaptation loop as it holds a global view on the adaptable software through maintaining an architectural RTM that represents the state. While state-based realization of the adaptable software supports decisions with global impacts, event-based processing of the changes enables incremental execution of the feedback loop. Scalability in VENUS is achieved through the incremental processing of adaptation activities and leveraging the locality information of the event-based changes affecting the self-adaptive software. VENUS is scalable as its complexity is independent of the size of the system architecture and is only influenced by the number of adaptation issues.

The pattern-based realization of the utility functions as well as the adaptation rules jointly enable mapping the impact of the rule executions to the corresponding utility values in VENUS. Based on the expected impact of the adaptation rules on the overall system utility, and based on the estimated costs of the adaptation rule executions, VENUS aims to obtain optimal adaptation decisions that maximize the system utility cost-effectively. In this thesis, we demonstrate that, for the targeted class of self-adaptive systems, VENUS is *robust*, *scalable*, *timely*, and *optimal*—in terms of system utility. The optimality claim in VENUS is restricted to the class of self-adaptation problems that satisfy the *greedy choice property* [see 121], where a globally optimal solution can be reached by making a locally optimal (greedy) choice at each step [6].

1.3.3 HYPEZON*: Coordinating Off-the-shelf Policies*

The second part of our proposed solution is HYPEZON, a coordination-based HYbrid PlannEr for self-adaptation employing receding horiZON [see 298]. VENUS constitutes the main contribution for engineering self-adaptive software. HYPEZON serves as a complementary solution with a relatively low development and deployment effort. The scheme proposes a coordination-based hybrid adaptation for problems that a runtime coordination between multiple off-the-shelf policies could satisfy the cost and quality objectives, thereby rendering the cost-intensive development of custom solutions such as VENUS dispensable. In short, VENUS might provide over-engineered[4] solutions to a problem that a simpler process, in terms of development-effort, is adequate. Furthermore, HYPEZON is beneficial when the problem at hand does not satisfy the greedy choice property, thus VENUS cannot provide optimality guarantees.

HYPEZON exploits the notion of meta-self-awareness [see 284] to provide the hybrid planner, realized in an additional, higher-level control loop with increased awareness. This way, HYPEZON holds a global view of the system and the adaptation process that allows for observing phenomena with global scope. The scheme enables the self-adaptive software to observe its own behavior (in combination with the adaptation feedback con-

4 "The act of designing a product or providing a solution to a problem in an elaborate or complicated manner, where a simpler solution can be demonstrated to exist with the same efficiency and effectiveness as that of the original design" [324].

trol loop in terms of objective satisfaction), reason about changing trade-offs during its lifetime, and explore coordination of multiple off-the-shelf policies at runtime. HypeZon exhibits the characteristics of a *generic* solution for hybrid adaptation since it is designed to consider the employed adaptation policies as black-box and can coordinate arbitrary policies—see Figure 1.2 for an overview.

HypeZon considers the *coordinator unit* as an additional, conceptual entity and implements it as a controller with receding horizon. Similar to the benefits of the external adaptation approaches over the internal alternatives [367], the external realization of HypeZon in the design of the control loops supports explicit separation of concerns at the architecture level, yielding reusability, easier maintenance, and independent evolution of each level [265, 427]. Conforming to the well-established practice of explicitly capturing control loops in the architecture of the system [208, 367], HypeZon adopts the hierarchical arrangement of the control loops from adaptive control in its design [380].

The control design and architecture of HypeZon build on control theory as a prominent base, and moreover, extend the involvement scope of the higher-level control loops in the lower-level entities. In adaptive control, the controller may change its own control regime by having adjustable parameters [see 280]. The adaptive control theory restricts the scope of controllers to calculating set-points and prescribing required changes in the system input parameters accordingly. In HypeZon however, adaptive control is perceived as reasoning about the adaptation logic [see 140] that requires advanced self-reflective properties through, e.g., meta-self-awareness [285].

1.4 CONTRIBUTIONS

In brief, this thesis advances the state-of-the-art in software architecture-based self-adaptation by providing the following contributions:

C1 - Incremental execution of the adaptation loop, thus attaining scalability
In order to perpetuate a global-scope view, the adaptation engine maintains a causally connected architectural RTM that captures the state of the adaptable software and its context. The MDE principles are explored to engineer the elements of a MAPE-K feedback loop. Dynamic architectures are supported via realizing the architectural RTMs as a graph ($\mathcal{R}5$) where the change events, the condition, and the action parts of the ECA rules are then captured as graph patterns. Using an RTM avoids complicated events that carry the context of a state change because the model represents the state of the adaptable software, therefore, the context of the change. Modifications to the dynamic architecture during software execution are realized and implemented via graph transformation rules.

Scalability in Venus is achieved via fully exploiting the opportunities for incrementality as majority of the runtime changes, while occurring frequently, are by their nature incremental, i.e., limited number of phenomena change at each time point [183]. The RTM provides for capturing change events that typically contain a fraction of the state corresponding to the context of the change. Event-based execution of the adaptation loop allows for the loop activities only processing the change events rather than the whole state, thus supporting cost-effective execution of the adaptation ($\mathcal{R}2$). In this thesis, enabled by model-driven adaptation, we employ incremental monitoring, analysis, planning, and execution to improve the runtime performance of the adaptation loop contributing to a more scalable solution, thus supporting $\mathcal{R}3$—see Table 1.1.

C2 - Defining pattern-based utility functions for dynamic architectures
This thesis uses utility theory to capture the high-level system objectives and potential trade-offs among them in the form of utility functions and preferences over different quality attributes, thus supporting $\mathcal{R}4$. When available, we use domain knowledge to engineer analytically defined utility functions. We explore a wide range of complexity classes for utility functions that are characterized at the level of software architecture. Representing the software architecture as a graph allows for pattern-based definition of utility functions. Therefore, similar to the adaptation issues and ECA rules (see **C1**) we define utility values for dynamic architectural fragments, i.e., graph patterns (supporting $\mathcal{R}5$). Pattern-based definition of utility functions allows for incremental utility calculation that is in harmony with the incremental execution of the adaptation loop activities and provides for scalability (supporting $\mathcal{R}3$).

The majority of state-of-the-art in utility-based adaptation either employ search-based optimization in the solution space hindering scalability, e.g., [129, 359], or characterize utility functions over pre-defined, and hence, bounded configuration spaces—see [91, 146]. In this thesis, we advance the state-of-the-art in utility-driven self-adaptation by defining utility functions that exploit beyond the linear utility functions and cover a wide range of mathematical complexity. Moreover, utility functions are then associated with architectural graph patterns, thus extensible as software architecture evolves at runtime, thereby supporting evaluation of dynamic architectures—see $\mathcal{R}5$ in Table 1.1.

C3 - Training utility-change prediction models for rule-based self-adaptive software
In addition to the analytically designed utility functions, we propose a methodology to systematically train prediction models for utility-changes in rule-based self-adaptive software systems (supporting $\mathcal{R}6$). This is beneficial for engineering systems with a black-box model with initially unknown knowledge of system inner-working, e.g., feature dependencies, or runtime preferences, or when the prior knowledge becomes obsolete. Moreover, large and complex architectures make the manual acquisition of a utility function challenging. The proposed methodology extends and modifies the standard machine learning process to contemplate the class of self-adaptive systems with dynamic architectures employing rule-based adaptation. The methodology learns prediction models for multiple variants of a system employing different utility models that vary in the complexity and number of attributes. The learned prediction models approximate an analytically-defined optimum with an acceptable error margin.

C4 - Combining rule- and optimization-based formalism of adaptation policies at design time
Venus is designed based on a combination of ECA rule-based constructs and utility theory. The pattern-based characterization of the ECA adaptation rules as well as the utility functions allows for mapping the utility values to the adaptation rules and therefore, predicting the impact of each rule application on the overall utility. Based on the predictions for the impact of the adaptation rules and leveraging the knowledge about the cost of each rule application, rules that offer an acceptable utility-cost trade-off are chosen to carry out the adaptation, thus supporting $\mathcal{R}4$ in Table 1.1. As a result, Venus brings together the benefits of the rule- and optimization-based approaches.

C5 - An optimal solution for self-adaptation with negligible runtime overhead
Venus aims for optimality with respect to the system quality objective satisfaction via pursuing a greedy algorithm before each adaptation, thus the optimality claim is restricted to the class of self-adaptation problems that satisfy the greedy choice property where a globally optimal solution can be reached by making a locally optimal (greedy) choice at each step. Mapping utility values to adaptation rule applications allows for assessing the expected utility-increase of each rule application before the adaptation, and choosing a rule that maximizes the overall utility ($\mathcal{R}1$). Capturing the adaptation decisions via rule-based formalism, pattern-based characterization of the adaptation rules as well as of the utility values, and finally, incremental execution of the adaptation loop collectively contribute to engineering an adaptation process that introduces negligible runtime overhead. Moreover, the solution is scalable for large and dynamic software architectures, thus supports cost ($\mathcal{R}2$) and scalability ($\mathcal{R}3$) requirements.

We demonstrate that the computation complexity in Venus is only affected by the number of change events, hence independent of the size of the architecture and the configuration space. Venus computes the utility for each possible adaptation option incrementally and at runtime, taking into account the change events and their contexts. Due to the incremental computation, the scheme is scalable while achieving optimal adaptation decisions in terms of utility and *reward*, i.e., accumulated utility over time.

C6 - A robust solution
The IEEE standard [227] delineates the robustness of a software system as the degree to which the system operates correctly in the presence of exceptional inputs or stressful environmental conditions. We demonstrate the robustness of Venus, as is customary [see 306], via bombarding the self-adaptive software with valid and exceptional inputs and verify the success criteria, i.e., *if it does not crash or hang, then it is robust*. To this end, we operate a self-adaptive software equipped with Venus together with a diverse collection of input traces covering a large and representative spectrum of the excepted input space for the system. Moreover, we demonstrate the robustness of the scheme through executing it outside its intended operation condition, i.e., when certain validity assumptions are violated.

C7 - A generic scheme for hybrid self-adaptation
As a complementary solution to Venus, we propose HypeZon. HypeZon leverages control-theoretic principles to systematically engineer a generic hybrid adaptation mechanism to support $\mathcal{R}1$ and $\mathcal{R}2$. The scheme coordinates multiple off-the-shelf adaptation policies and chooses at runtime the one that best suits the operation conditions. HypeZon is generic, as its design is independent of its constituent policies. The scheme advances the state-of-the-art for hybrid adaptation of software systems by providing explicit architectural design and explicit separation of concerns, i.e., adaptation and policy coordination, at the architecture level. Moreover, it allows for reusability, easier maintenance, and independent evolution of each level, i.e., for adaptation and policy coordination.

C8 - Coverage of a wide spectrum of self-adaptation problem space beyond the state-of-the-art

In light of our *Systematic Literature Review* (SLR) of state-of-the-art in evaluation of self-healing systems [see 176, 177], we identified a set of required improvements to support the claims in self-healing systems via evaluation. The identified improvements, developed into an evaluation methodology, are implemented in thesis. The applicability and effectiveness of our solutions as well as the credibility of our contributions are investigated according to said methodology via thorough evaluation. Figure 1.1 shows an example of how evaluating an approach against only a single (or a non-representative set of) data points in the input space of the self-adaptive software yields inconclusive results—see how the best performing solutions in Figure 1.1a change as complexity increases.

1.5 thesis evaluation plan

To evaluate this thesis, we apply both Venus and HypeZon on two different application examples based on two different architectural styles. For each application, we employ multiple utility functions that vary in complexity and number of attributes and capture different system objectives. To ensure a conclusive, robust, and reliable evaluation outcome, each approach-application combination is evaluated via multiple reproducible, controlled experiments over a wide spectrum of the adaptation input space providing for sensitivity analysis of the results—as suggested by our evaluation methodology in **C**8 (see Section 1.4). In addition, we implement two alternative solutions for self-adaptation: a deterministic, rule-based scheme and an optimization-based scheme that are used in a comparative study with Venus. We use the two solutions also as off-the-shelf adaptation policies in the evaluation of HypeZon.

We evaluate the impact of incremental execution of the proposed adaptation engine that encapsulates Venus and HypeZon in a comparative study with an instantiation of a non-incremental alternative. We quantitatively compare the solutions via a set of experiments across different architecture sizes and input traces.

In the context of Venus, we comprehensively evaluate the scheme with respect to the quality attributes of optimality, scalability, runtime performance (timeliness), and robustness. We investigate the *optimality* of Venus in terms of business goal satisfaction that is quantified via utility functions; we compare the reward of the scheme to an optimization-based solution that employs a constraint solver to plan the adaptations. The *performance* and *timeliness* of Venus is compared to a deterministic, rule-based adaptation policy that employs design-time preferences with no runtime overhead. The *scalability* of Venus is evaluated via controlled evaluation scenarios while systematically increasing the size of the system architecture and the number of the adaptation issues. The timeliness, optimality, and effectiveness of Venus are studied for *robustness* during multiple evaluation scenarios which simulate operational environment with different characteristics in terms of the magnitude and pace of the changes resulting in adaptation issues. Moreover, we evaluate the effectiveness and correctness of Venus when certain validity assumptions are violated.

We evaluate our methodology to train utility-change prediction models concerning model accuracy, performance, and runtime effort. We conduct the evaluation for prediction models across a breadth of mathematical complexities and number of attributes in the utility functions. We analyze the design of the methodology by comparing it to the standard machine learning process and discussing the design guidelines. We investigate the application of the methodology to a black-box software system. We study the model accuracy and runtime effort of different learners employed to train prediction models. For this purpose, we execute the prediction models on a running system to estimate the impact of the adaptation rules on the overall system utility. The performance of the self-adaptive system is then compared to a white-box equivalent of the system where the ground truth, i.e., the analytically defined utility function, is available. We assess the benefits of an aggregation metric to guide the choice of the best fitting prediction model at runtime without requiring the models to be deployed on a real system.

In the context of HypeZon, we first discuss its implementation that is captured through two different designs. We then compare the two designs with respect to their impact on the utility and timeliness of the adaptations via multiple evaluation scenarios on two different application examples. The benefits of using receding horizon in HypeZon is evaluated through a comparison to an alternative solution for hybrid adaptation that does not support runtime adjustments of its control parameters and exploits deterministic conditions for policy invocation. Finally, we investigate the cost and benefit of hybrid adaptation versus employing the individual adaptation policies to solely carry out the adaptation.

1.6 DOCUMENT ROADMAP

The remainder of this document is structured into three parts. In Part I, we describe the preliminaries of this work and introduce the running example (Chapter 2). Part II presents the technical contributions of this thesis. We outline how to define, manually construct, and learn architectural pattern-based utility functions (Chapter 3) and introduce Venus in Chapter 4. Chapter 5 presents HypeZon. Finally, Part III introduces the application examples and their corresponding input traces in Chapter 6, presents the implementation of Venus and evaluates it via a set of qualitative and quantitative experiments in Chapter 7, investigates the methodology to train prediction models for utility-changes in Chapter 8, and evaluates HypeZon in Chapter 9. Moreover, Part III includes discussion of the related work (Chapter 10), concludes the thesis, and presents an outlook on the future work (Chapter 11).

Part I

PRELIMINARIES

This part presents the scientific background and foundations for this work as well as the running example that is used through out the thesis. In detail, we discuss the concepts of control theory, utility theory, self-adaptive software, model-driven engineering, and graph transformation.

2 FOUNDATION

2.1 CONTROL THEORY

Control theory is concerned with engineering disciplines to influence the behavior of *dynamical systems*. A dynamical system is a system whose behavior changes overtime, often in response to external stimulation or forcing [19]. The design of a controller is based on a mathematical model of the system behavior. This model is usually a dynamic model defined based on differential or difference equations. The system model formalizes the relationships between time, system state, control variables, i.e., system inputs, and the controlled variables, i.e., system output. The objective is to develop a model or algorithm to govern system inputs such that the desired output is achieved, while minimizing any delay, overshoot, or steady-state error and ensuring a level of control stability; often with the aim to achieve a degree of optimality.

Closed-loop control is one of the several process control paradigms that is extensively studied and applied in various disciplines concerned with the control of dynamical systems. Closed-loop control uses a *controllable process* that usually includes the environment, a controlled variable or measured output, and control action(s). As shown in Figure 2.1, the *target system* is controlled by *control actions* form the *controller*. The *reference* input—also known as the *set-point*—is the desired value of the system's *measured output*. The objective of the closed-loop control is to continuously adjust the control input to the target system such that the measured output matches the reference input within some error margin despite the *disturbance* [46]. For this purpose, the difference between the measurement and reference, i.e., *error*, is fed back to the controller which determines the control actions that are required to achieve the reference input. The *disturbance* captures any change that affects the way in which the control input influences the measured output.

An alternative design for control mechanisms is the *open-loop control*, also referred to as *feedforward* which is a technique that avoids using the measured output to adjust the control input. Thus, the control actions are independent of the system output. While feedback control is reactive, i.e., it is invoked by perturbations in the target system to perform corrective actions, feedforward control is predictive as it can anticipate changes to the measured output. Monitoring only the control input, feedforward controller determines control actions. Under certain circumstances, it is possible to measure a disturbance before it affects the system and use this information to take corrective actions

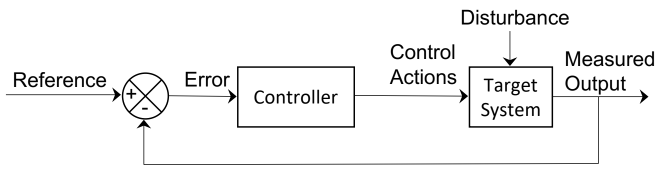


Figure 2.1: Block diagram of a feedback control loop.

before the disturbance influences the system. This way, the effect of the disturbance is reduced by measuring it and generating a control signal that counteracts it through the feedforward control.

An open-loop controller assumes stable conditions where the control result is approximately adequate under normal conditions without the need for feedback. In this design, an accurate model of the target system is required to determine the setting of the control input. The control input is then a deterministic function of the reference (and/or disturbance) input. Open-loop control supports reduction in component count and complexity, however, it cannot make any corrections on potential control errors or adjustments for disturbances.

Applying controllers to computing systems often requires consideration of several feedback control properties; A control system is *stable* if for any bounded input the output is also bounded. Stability refers to the system reaching a *steady-state* equilibrium, that is, the observed property converges to a specific value, ideally the set-point, and stays inside a previously defined stability margin around this convergence point. The control system is *accurate* if the measured output converges (or becomes sufficiently close) to the reference input. Typically, we do not quantify accuracy. Rather, we measure *inaccuracy*. For a system in steady-state, its inaccuracy, or steady-state error measures how far from the set-point the system converges, i.e., the steady-state difference between the set-point and the observed property. The *settling time* is the time required for the system to reach the steady-state equilibrium. And finally, *overshoot* refers to an output exceeding its final, steady-state value. The control system should achieve its objectives in a manner that does not overshoot.

Computing systems often have multiple inputs, e.g., settings of configuration parameters, and multiple outputs, e.g., response time and throughput. Thus, controlling a software system can be viewed as *Multi-input, Multi-output* (MIMO) control problem, however, the measured output and control input are often *discrete-time* rather than *continuous-time* as in traditional closed-loop process control. Moreover, discrete-time is consistent with the way that measurements are naturally obtained from computing systems. Provided that the measured output and control input can be properly identified in a software system, control theory also supports the control of discrete variables.

2.1.1 *Adaptive Control*

While feedback control has established mathematically grounded and practical frameworks for managing complex systems [140], it restricts the scope of the controllers to calculating set-points and prescribing required changes in the system input parameters accordingly [137]. The black-box-oriented scheme of feedback control further extends towards *adaptive control*, where the controller may change its own control regime. An adaptive controller is a controller with adjustable parameters and a mechanism for adjusting the parameters. In general, adaptive control is concerned with a set of techniques which provide a systematic approach for automatic adjustment of the controllers in real time in order to achieve or maintain a desired level of control system performance, when the parameters of the target system are unknown and/or vary over time [280]. This requires the controllers to have adjustable parameters. Moreover, an *adjustment mechanism* needs to be in place to oversee the parameter tuning via layered arrangements of the control loops, where the lower-level controller is controlled by the immediate higher-

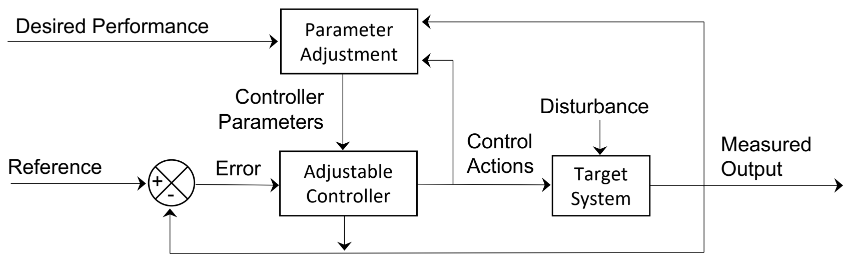


Figure 2.2: Block diagram of an adaptive control system.

level loop [113]. The tuning of the controller is done in real-time based on the real-time data collected from the system. Figure 2.2 shows a block diagram of an adaptive control system. The controller in this case is nonlinear because the parameters of the controller depends upon measurements of the system variables through the *adaptation loop*, i.e., the higher-level, parameter adjustment loop.

An adaptive control system measures a certain performance index of the control system using the inputs, the states, the outputs, and the known disturbances. Via comparing the measured performance index and a set of desired inputs, the adaptation mechanism modifies the parameters of the adjustable controller and/or generates an auxiliary control in order to maintain the performance index of the control system close to the *desired performance*. Note that the control system here (Figure 2.2) is an adjustable, dynamic system in the sense that its performance can be adjusted by modifying the parameters of the controller or of the control signal. The above definition can be extended for adaptive systems in general in a straightforward manner [279].

An adaptive control system, in addition to a basic closed-loop feedback control with adjustable parameters, contains a supplementary loop that takes explicit measures to compensate for variations in the system dynamics or for variations in the disturbances, in order to maintain the optimal performance and robustness of the system. In brief, an adaptive control system can be thought of as two loop; on is a conventional feedback control loop with the target system and the controller; the other loop is the parameter adjustment loop [20].

2.1.2 *Model Predictive Control*

A specific form of closed-loop control called *Model Predictive Control* (MPC) is particularly well suited for MIMO control problems with inequality constraints between manipulated inputs and outputs [192]. A block diagram of an MPC system is shown in Figure 2.3. Provided an accurate dynamic model of the target system, model and current measurements can be used to predict future values of the system outputs. The difference between the actual and the predicted outputs, i.e., *error*, serves as the feedback signal to a *prediction* block. The appropriate changes to the input variables are then calculated based on both the predictions and the measurements. The predictions are used to generate *set-points*. Set-points define the target values for *control calculations*. Set-point calculations and control calculations are performed at each *sampling interval*. In essence, the changes in the individual input variables are coordinated after considering the input-output relationships represented by the system model [68].

MPC formulates a multi-variable optimization function for set-point calculations. Typical optimization objectives include maximizing a profit function (e.g. a utility function),

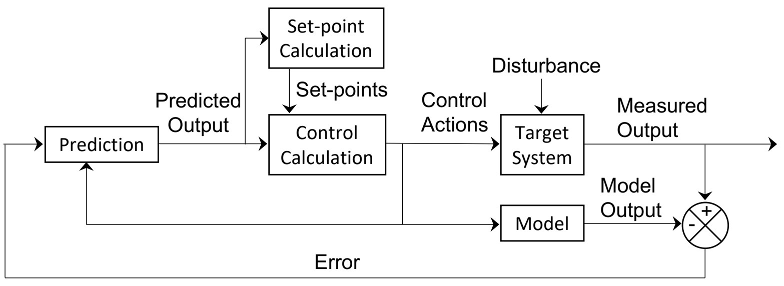


Figure 2.3: Block diagram for model predictive control.

minimizing a cost function, or maximizing a production rate. The MPC calculations are based on current measurements as well as the predictions of the future values of the outputs. The objective of control calculation is to determine a sequence of M control actions so that the predicted output moves towards the set-points in an optimal manner. Control calculations determine a sequence of M control actions (i.e., *control horizon*) such that the *predicted output* moves towards the set-points over a finite *prediction horizon* P—see Figure 2.4 for basic concepts of MPC from [374].

The advantages of MPC are manifold: the technique systematically captures the constraints on inputs and outputs. Moreover, the target system model captures dynamic and static interactions between input, output, and disturbance variables. Set-point calculations and control calculations can be coordinated because the set-points define the target values for control calculations. Finally, leveraging accurate model predictions provides early warnings of potential problems. The success of MPC depends on the accuracy of the system model.

A distinguishing feature of MPC is its *receding horizon control*. It suggests that although a sequence of several control actions is calculated after each sampling interval, only the *first* action is executed. A new sequence is calculated in the next sampling interval and after new observations become available. Similarly, only the first action for the new sequence is executed. This procedure is repeated at each sampling interval. Employing a receding horizon of size one supports the case where the variables available for the control calculations change from one execution time to the next. If the control structure changes from one control execution time to another but the MPC controller does not recalculate the parameters, the subsequent control calculations may become *ill-conditioned* [374]

This thesis draws concepts and techniques of feedabck and MPC, adaptive control, receding horizon control, stability, and steady-state from the control theory domain to

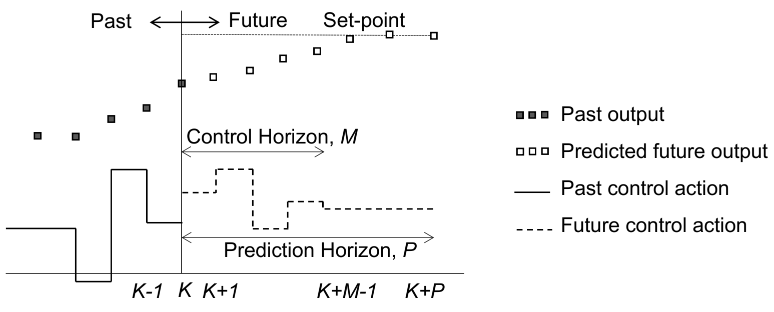


Figure 2.4: Basic concept for MPC from [374].

control software systems. Unlike the traditional control systems, however, this thesis targets discrete, state-based computing systems [see 208] which require a different type of system model, e.g., a software architecture model.

2.2 UTILITY THEORY FOR DECISION-MAKING

Decision theory in its essence is the interdisciplinary study of choice. Given two or more incompatible *options*, decision theory is concerned with how optimal decisions can be reached [152]. A core concern of decision theory is to study choice under uncertainty [261]. The choice becomes challenging between incommensurable commodities [54]. Numerous techniques for choice under uncertainty have been developed; decision trees and decision tables to evaluate desirability of decisions via tests on attributes [420], *Expected Utility* (EU) theory to attribute values to decisions [127, 142], and *Markov Decision Process* (MDP) to solve for an optimal decision [131].

Following the expected utility theory for decision-making, when presented with options, each outcome is assigned a subjective value, i.e., utility, that reflects the strength of the preference for that outcome. This way, decision-making is about choosing among options based on the expected desirability of their *immediate* outcomes [361]. For this purpose, the numerical representation of preference orderings is important. The numerical measures in question are known as *utility functions*. The principle of *Maximum Expected Utility* (MEU) suggests that a rational decision-maker should make a choice that maximizes the expected utility.

Von Neumann and Morgenstern [419] formulated four axioms for utility theory as constraints on rational preferences[1]. The axioms of utility theory imply the existence of utility functions. As stated by Keeney [245, 248] and DeGroot [109], if the preferences have been determined consistent with the axioms, then it can be shown that a numerical utility function *always* exists. Russel and Norvig [361] refer to this inference from utility axioms as the *existence of utility function*, i.e., *"if an agent's preferences obey the axioms of utility, then there exists a function* U *such that* $U(A) > U(B)$ *if and only if A is preferred to B and* $U(A) = U(B)$ *if and only if the agent is indifferent between* A *and* B"—see [361, p. 613].

2.2.1 *Analytics of Preferences: Engineering and Learning*

Constructing a utility-theoretic decision-maker requires obtaining a utility function that captures the preferences. This process is often referred to as *preference elicitation* or equivalently, *utility elicitation* [79, 152]. Preference elicitation approaches involve collecting the system's *user* preferences and using them to suggest progressively better recommendations until a satisfactory option is obtained [338]. A user is another system—physical or human—which interacts with the former [281]. Preference information may be acquired in either *direct* or *indirect* manner [249]. The former relies on interactive user/

1 The *completeness* axiom states that for any two possible outcomes, either one outcomes is preferred or the individual is indifferent. The *transitivity* axiom (assumes that preferences are consistent across any three options). The *continuity* axiom states that given three subjectively ordered outcomes, a decision maker will be indifferent between the middle outcome and a probabilistic combination of the two other outcomes. Finally, *independence* states that preferences between two options should not change when altering both options equally by mixing them with a common outcome– we refer the interested reader for the mathematical formalization of the axioms to [419].

stakeholder querying while latter can be obtained indirectly from preference statements such as SLA, user critiques, observations of user's clicking behavior, etc.

2.2.1.1 *Engineering utility function for software systems*

The burden of utility elicitation can be lessened considerably if the decision-maker is presented only with a restricted set of scenarios. This allows for making assumptions about the preferences based on the available domain knowledge, provided by users or domain experts. Leveraging domain knowledge to capture preferences allows for deriving functional form(s) of the utility function that satisfy the assumptions [51, 261, 333]. If the assumptions are verified, the functional form can be used to simplify the requisite assessments needed to specify the utility function.

Traditional elicitation methods rely on the domain knowledge, e.g., via querying users or stakeholder about the desired behavior of the utility function or the relative importance of every outcome in terms of each decision criterion [245]. Once presented with the preferences, the engineer/designer leverages the knowledge for *direct utility assessment*, i.e., to program simple standard mappings between outcomes and utility values [90]. Alternatively, the knowledge can be utilized for *qualitative structuring of preferences*, i.e., postulating various assumptions about the preference attitudes of the decision-maker and deriving functional forms of the multi-attribute utility function consistent with these assumptions that ascribe proper utility values to outcomes—see [249].

While the concept of utility is theoretically sound, specifying reasonable procedures for obtaining multi-attributed utility functions is challenging [248]. The elicitation of preference and utility functions is complicated by the fact that utility functions are very difficult for users to assess [338], more specifically when multiple attributes capturing different dimensions are involved.

In decision-making under uncertainty, a common operational method for assessing multi-attributed preferences is the use of *additive utility function* [230, 340]. In n dimensions, the additive utility function may be written as $u(x_1, x_2, ..., x_n) = \sum_1^n u_i(x_i)$, where $u_i(x_i)$ is a utility function defined over the X_i attribute—see [246]. The additive utility decomposition is applicable only when the *mutual utility independence* between different attributes hold. Utility independence concerns situations where the levels of some attributes are deterministically fixed. The assumption then requires that preferences between prospects over the remaining attributes to be independent of the fixed deterministic levels [48]. The assumption of independence allows for the reduction of the number of outcomes for consideration and the construction of less complicated and thus more manageable utility functions. The main advantage of the additive utility function is its relative simplicity. The assessment of the $n-$dimensional utility function is reduced to the assessment of n one-dimensional utility functions and $n-1$ scaling constant. A major shortcoming of employing additive decomposition of utility is the restrictiveness of the necessary assumption [247].

Keeney [247] shows that when the utility independence assumption holds, the utility function is either additive or *multiplicative*, i.e., the utility of any given outcome can be broken down to the multiplication of the individual attributes [248]. In this thesis, we consider additive utility functions, thus we exclude systems where attributes are preferentially dependent and assumptions of additive decomposability do not hold.

2.2.1.2 *AI for preference learning*

While the classic preference elicitation protocols are well-founded, their applicability is restricted in variety of circumstances, e.g., when sufficient prior domain knowledege is not available. Initiating from the operations research community, several researchers started to deal with the problem of eliciting a utility function when only incomplete information is available—see [24, 230]. More recently, researchers in *Artificial Intelligence* (AI) [see 50, 52, 79, 410, 423] have developed elicitation techniques for utility elicitation with the goal of relaxing the requirement for sufficient prior information to manually craft a utility function and mitigating the cognitive cost for the user/designer.

As AI in its core is concerned with developing agents that act rationally on behalf of the user, eliciting the preferences of the user in an effective way is therefore crucial. Preference elicitation from the AI perspective is the process of interactively learning a model that captures the preferences with the goal of providing a high-quality recommendation with minimal cognitive effort for the designer [287, 338]. Some of the AI-based techniques to automatically learn or extract a model capturing the preferences include but are not limited to supervised learning, optimization, planning, and *Reinforcement Learning* (RL) [54, 261].

Supervised learning methods for utility elicitation, mainly applied to classification [78, 286] and regression [56] problems, provide a set of training example in the form of input–output pairs. The employed automated learning algorithm then infers a function that maps from input to output. Once the learning has converged, the learner may generalize a preference model from the provided examples that allows the algorithm to correctly determine the class labels for unseen instances [30, 64]. The success of this technique is measured through its *generalization error* that strictly depends on the size and the characteristics of the training data sets as well as the complexity of the *true* function.

Optimization methods for utility elicitation demand specification of the possible decision space as well as the attribute measures to be maximized. The utility of a decision is quantified through several simulations on the dynamic model of the problem at hand while employing the decision. The optimization algorithm then performs a search in the decision space for the optimal decision. The success of the optimization methods depend on the size of the decision space as well as the distribution of the utility measures across the space. If the decision space is relatively low dimensional and the utility measures do not have many local optima, then various local or global search strategies become relevant, e.g., hill-climbing [237], simulated annealing [257], and WalkSAT [173]. Although the optimization techniques assume a knowledge of a dynamic model to run the simulations, they do not require a model to guide the search for the optimal decision which is essential in complex problems.

Planning technique for automated utility elicitation is a branch of the optimization technique that requires a model of the problem dynamics to guide the search in the decision space. Majority of the state-of-the-art in this context focus on deterministic problems where a deterministic model suffices. While deterministic models support application of methods that more easily scale to high-dimensional problems, they fail to account for future uncertainty [242, 262]. reinforcement *RL*-based methods use model-free learning algorithms where the learner is given only a numerical performance score as guidance [119, 264]. This technique eliminates the model requisite of the planning methods to guide the search in the decision space [330]. As a result, the desired decision-making strategy is learned while the decision-maker interacts with its context. RL-based methods for utility elicitation only demand the designer to provide attribute measures to optimize, the learner then optimizes the behavior of the decision-maker towards making optimal decisions. RL-based methods however cannot always provide guarantees for convergence.

2.3 SELF-ADAPTIVE SOFTWARE

2.3.1 *Vision*

The vision of *autonomic computing* and *self-adaptation* emerged as the computing systems' complexity approached human limits for manual maintenance, development, deployment, management, and evolution [112, 252]. Autonomic and self-adaptive systems have been used interchangeably in the literature [112, 214], however, there are slight differences in the perspectives from which these systems are being considered. Autonomic computing takes a system administrator perspective and refers to"computing systems that can manage themselves given high-level objectives from administrators" [252, p. 41]. In contrast, self-adaptation considers "systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals". Thus, takes a software engineer perspective [112, p. 1]. While self-adaptive systems are used in a number of different areas, this thesis focuses only on their application in the software domain, that is, *self-adaptive software*.

In a layered model for a software-intensive system consisting of hardware, operating system, network, middleware, and application, self-adaptive software primarily covers the application and middleware layers [302] while its coverage fades in the lower layers; autonomic computing addresses the lower layers as well. Autonomic computing has emerged in a relatively broader context while self-adaptive software domain is assumed to be relatively limited and falls under the umbrella of autonomic computing [367]. Autonomic computing and self-adaptation seek reducing the human involvement in the maintenance/evolution of the computing systems. Both visions share the objectives of automating the adaptation process and shifting the responsibility for performing the adaptation from human to the system itself. This way, software has a continuous life cycle where the traditional development-time activities of software engineers are shifted towards runtime [26]. This resulted in efforts towards runtime adaptation of software systems, i.e., changing a running system without pausing or stopping its life cycle [302].

Both visions build upon the *self*- and *context*-awareness of the system to mitigate the growing costs, complexity, and diversity of maintaining, operating, and generally evolving software systems. Self represents the whole body of the system and context encompasses everything in the operating environment that affects the system's properties and its behavior [111].

The self-* properties [see 22] associated with autonomic computing systems originate from the IBM manifesto [211]. Kephart and Chess [252] define the four properties of the self-managing and self-adaptive systems that serve as the de facto standard in this domain as follows: *self-configuration*, that is the system's ability to automatically install, integrate, configure, and compose/decompose (parts of) itself; *self-optimization*, that is, automatically seeking opportunities to improve performance and efficiency of the system; *self-healing*, that is, automatically detecting, diagnosing, and repairing errors, faults, and failures in the system; *self-protection*, that is, automatically defending itself against security breaches, malicious attacks, and cascading failures. Selehie and Tahvildari [367] base the above four major capabilities on two primitive ones: *self-awareness*, that is, the system is aware of itself, its own state, and behavior; *context-awareness*, that is, the system is aware of its operational environment and execution conditions.

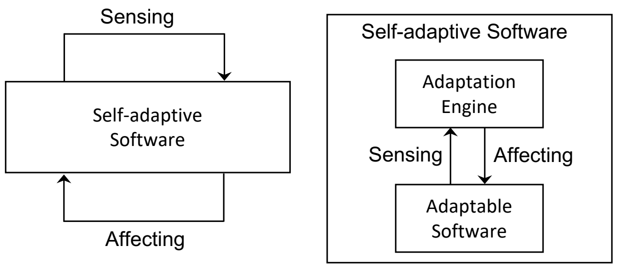
2.3.2 *Realization*

This section briefly introduces the key concepts to realize self-adaptation in software systems.

2.3.2.1 *Feedback control loop for software adaptation*

In general, it is software that realizes the self-adaptation by means of a feedback loop [see 60]. A distinct feature in engineering self-adaptive software is shifting certain design-time activities towards runtime where the system uses runtime feedback to adapt itself. Feedback loops from control theory provide a prominent base to engineer generic mechanisms for self-adaptation [60]. The adaptable software can be considered a *controllable process*, i.e., the target system, in the feedback control loop in Figure 2.1; the control feedback loop observes the system output in sampling intervals and adapts the system towards its set-points to prevent violation of system requirements and goals; the reference input in the feedback loop (see Figure 2.1) specifies the set of values and corresponding types that characterize the desired target state for the adaptable software; the adaptation is responsible for achieving and maintaining the desired target state(s) under changing conditions during system execution, e.g., disturbance; the measured output represents the set of values and corresponding types that are measured in the adaptable software; the measurements are compared to the reference inputs to evaluate whether the desired state is reached.

In order to enable self-adaptation properties in a software system, the software is equipped with an *adaptation mechanism* that monitors the software (self) and its operation environment (*context*) to detect changes and take appropriate actions to adapt the system accordingly [92]. An adaptation mechanism realizes and controls the adaptation process via adding a feedback loop to the software system resulting in a closed-loop system with feedback from the self and the context. As a result, the closed loop system exhibits degrees of variability provided by its self-* properties through which the self-adaptive system may automatically *configure*, *optimize*, *heal*, and *protect* itself at runtime according to its operational context.



(a) Internal approach. (b) External approach.

Figure 2.5: Internal and external approaches for building self-adaptive software.

Salehie and Tahvildari [367] distinguish between the two categories for incorporating adaptability into software systems: *internal* and *external*. As depicted in Figure 2.5a, the *internal* approach to subsume adaptability in a software system entangles the adaptable software and the *adaptation logic*, i.e., the feedback loop. In this approach, the whole set of sensors, effectors, and adaptation processes are mixed with the application code, which often leads to poor scalability and maintainability.

In contrast, the *external* approach as shown in Figure 2.5b separates the *adaptation engine*—implementing the adaptation logic—and the adaptable software. In this approach, the external adaptation engine controls, i.e., senses and affects, the adaptable software. The separation of the adaptation engine and the adaptable software allows engineers to use application-independent adaptation mechanisms and supports reusability of the adaptation engine or parts of it across different applications. The external approach addresses the drawbacks of the internal alternative in various aspects: it supports scalability because one adaptation logic can manage various resources; it increases maintainability through modularization [146, 275]; the external approach supports the attainment of a global view on the system and offers reusability of the adaptation logic. As stated by Salehie and Tahvildari [367] and confirmed in the survey conduced by Krupitzer et al. [275], the choice of the external approach in the design of self-adaptive software is relatively more prominent. In the following, we focus on the external approach for engineering software systems with self-* properties.

2.3.2.2 *MAPE-K reference model*

MAPE-K, depicted in Figure 2.6, is a common reference model that implements the external approach for realizing self-adaptation in a software system [252]. In this model, the control feedback loop is explicitly realized via a *MAPE-K* loop that comprises of components and interfaces for decomposing and managing the feedback loop. The adaptation engine then is implemented via the four *Monitor*, *Analyze*, *Plan*, and *Execute activities*. *MAPE-K* loop Monitors and Analyzes the system and if needed, Plans and Executes an adaptation of the system, which is all based on Knowledge [252]. In the context of autonomic computing, this cycle is referred to as the Collect/ Analyze/ Decide/ Act cycle [118]. The Knowledge component in the MAPE-K reference model is shared among the four activities and provides information about the adaptable software and its context; Knowledge can be an architectural model of the adaptable software [118, 146], implementation of a registry, a dictionary, a database or a repository [215]. In general, Knowledge consists of particular types of data with architected syntax and semantics,

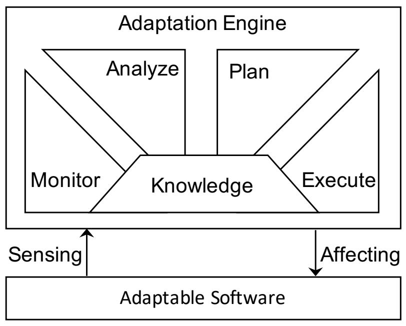


Figure 2.6: MAPE-K reference model for self-adaptive software.

such as symptoms of undesired situations, adaptation policies, change requests, and change plans [214].

2.3.2.3 *Parameter and structural adaptation*

Software adaptation can be generally conducted in two ways; *parameter* adaptation modifies variables of a program that determine the adaptive behavior. In contrast, *structural* adaptation, also referred to as compositional adaptation, exchanges algorithmic or structural system components, e.g., modifying system architecture. The software architecture is modified by adding, removing, or reconfiguring components and connections among them [302]. Structural adaptation enables a broader adaptation scope that is beyond the provisioned adaptability provided by parameter adaptation. In this thesis, we consider structural adaptation of software systems where software components, connectors, and their attributes define the points of variability in the *configuration space* for the self-adaptive software [35].

2.3.2.4 *State of adaptable software*

The architectural features of the self-adaptive software constitute the configuration space of the system that can be identified as a vector of *configuration attributes* setting $\bar{c}$. Thus, the system is adapted via changing the current configuration $\bar{c}$ to the new configuration $\bar{c}'$. The *state* of the adaptable software is comprised of a collection of parameters that characterize or model the system; the state is affected by the operational environment, i.e., context, and the internal configuration of the software systems; the state can be described as a vector of attributes that are either measured directly via sensors or are synthesized based on sensor measurements [421]. The *observable attributes* are set of features that can be observed and reflect the impact of the environment on the system or its context. An observation is a vector of observation values $\bar{o}$; the current observations $\bar{o}$ may change to new observations $\bar{o}'$ independently of the configuration changes. A collection of the configuration and observations attributes, $(\bar{c}, \bar{o})$, constitute the *state* of the adaptable software at each point in time [59].

2.3.2.5 *Adaptation policy*

The *Plan* activity in the MAPE-K loop takes into account the monitoring data from the sensors to produce a series of changes, i.e., an *adaptation plan*, to be executed on the adaptable software. The execution of the plan adapts the software system to a new state. Adaptation *policies* govern the decision making process during planning and are used to express and operationalize the high-level objectives of the system [214]. Numerous and varied definitions of policy [see 31, 104, 393] have been put forward in recent years and in their core they define a *high-level* policy as a declarative statement of what the end user wants. *Low-level* policy representations are generally procedural and specify the logic of how to achieve a goal, e.g. as rules or program control flows that are evaluated to determine a sequence of actions that should be taken [104]. As stated by Kephart and Walsh [253] and Bearden et al. [31], any single formal definition of policy tends to be restrictive in covering the broad spectrum of behavioral guidance that autonomic computing systems might require. AI is an appropriate core discipline from which to borrow concepts and techniques for autonomic computing and self-adaptive software systems, for automated decision-making is the central focus in AI [253]. As outlined by

Russell and Norvig [361], the notion of policy in general refers to a form of guidance used to determine decisions and actions. Three main types of policies for expressing and operationalizing the self-adaption goals are considered [214, 253]: Rule/ Action Policy, Goal Policy, and Utility Function Policy.

2.3.2.6 *Rule/ action policy*

ECA rules directly map specific event combinations to adaptation plans, i.e., actions. The semantics of ECA rules are straightforward: when the event occurs, evaluate the condition; if the condition is satisfied, execute the action [214, 300]. Rule policies operate in a *stateless* manner, thus offer limited applicability; in a stateless solution, the adaptation engine keeps no information on the state of the system and relies solely on the current sensor data, i.e., events, for analysis and planning. Keeping state information that can be updated progressively through sensor data and reasoned about, however, is beneficial; Kephart and Walsh [253] propose to facilitate state-based executions of rule policies via considering a state-based view of the rule policy referred to as *action policy*. An action policy governs the choice of actions that should be taken when the system is in a given state; this takes the form of IF(Condition) THEN(Action). The condition specifies either a specific state or a set of states that satisfy the condition.

2.3.2.7 *Goal policy*

Goal policies are a higher-level form of behavioral specifications that establish performance objectives, leaving the system to determine the actions required to achieve those objectives [421]. They specify either a single desired state, or one or more criteria that characterize a set of desired states for the adaptable software. Goal polices cannot express preference in choosing adaptation actions. In other words, goals describe the desired system state but not how to reach the desired state. Therefore, the adaptation engine is responsible for computing an action or a sequence of actions, i.e., a plan, that adapts the system from the current state to the desired state(s). Rather than relying on a human to explicitly encode rational behavior, as in action policies, the adaptation engine may generate adaptation plans from the goal policy. This provides more flexibility and eliminates the necessity of knowing low-level details of system function at the cost of requiring reasonably sophisticated planning or modeling algorithms—see [88, 387, 422].

2.3.2.8 *Utility function policy*

Utility function policies quantify the desirability of each possible state of the adaptable software; they require objective functions that map system states to scalar values [421]. In a *utility-driven* adaptation mechanism, the goal is to adapt the system to a feasible state that optimizes the objective function. Via assigning desirability values to states, utility function policies generalize the binary state classification of the goal polices to only desired and undesired states. Compared to the goal and action policies, utility function policies provide more fine-grained and flexible specifications of the adaptive behavior [421]. Utility functions permit on-the-fly determination of a *best* feasible state while goal policies place the system in any state that is both feasible and acceptable, with no drive towards further improvement. In general, utility function policies allow for decision making by specifying the appropriate trade-off. However, utility functions policies require multi-dimensional set of preferences, which is non-trivial to obtain [248]—see Section 2.2.1. Utility function policies can be viewed as generalizations of goal policies; conceptually, a utility function can be defined by specifying a complete set of disjoint goals and assigning values to them—this is not generally feasible when there is a large state space; and is impossible if the state space is continuous; in these cases, more compact functional expressions of utility functions should be considered. Although action policies are computed by optimizing utility function policies, there is no meaningful sense in which utility function policies can be derived from action policies because action policies are defined over the *current* state space and utility function policies are defined over the *desired* state space [see 253].

As discussed by Kephart and Walsh [253], among the three main mechanisms (policies) to capture the adaptation logic discussed above, action-/rule-based policies can serve as the foundation for more sophisticated goal- and utility-based policies as adaptation actions or adaptation rules, henceforth used interchangeably, capture the fine-grained units of change and collectively constitute the repertoire of ways in which an adaptable software can be modified.

2.3.2.9 *Static and dynamic policy*

From a temporal perspective, Salehie and Tahvildari [367] categorize the process of decision-making in the adaptation engine into *static* and *dynamic* decision-making. The former stipulates decisions woven to the adaptation engine based on design-time conditions while the latter is concerned with the runtime assessment of the conditions affecting the adaptable software. Static decision-making operates based on design-time hard-coded decisions where adaptation actions are mapped to states (conditions) according to design-time estimates for the resulting state [49]. Adaptation plans synthesized by static decision-making processes are likely to yield sub-optimal behavior at runtime because the behavior is hardwired into the process and cannot be changed dynamically and at runtime. The estimations for the possible adaptation outcome, e.g., the desirability of the resulting states, are agnostic to runtime conditions. However, due to minimal runtime activity, static solutions for adaptation do not introduce considerable runtime computation overhead during planning [322].

In contrast, a more flexible approach to compositional adaptation implements it at runtime via realizing *dynamic* decision-making [302]. Dynamic decision-making mechanisms can modify the design-time decisions based on runtime conditions during system execution without halting or restarting the adaptable software. Thus, leveraging runtime observations available during system execution to define the adaptive behavior. In dynamic decision-making, adaptation policies [253], adaptation actions [289] or adaption goals [190] may be externally defined or managed, allowing them to be modified during system execution. This provides for more flexibility in the adaptive behavior of the system regarding both functional and non-functional software requirements [367]. However, as discussed by McKinley et al. [302], dynamic solutions to runtime adaptation, in addition to the necessity of deciding at runtime and dynamically, face additional challenges such as finding optimal or partially optimal solutions for decision-making problems, dealing with uncertainty and incompleteness of the observations available at runtime, and addressing the scalability and fault-proneness of the decision-making mechanisms.

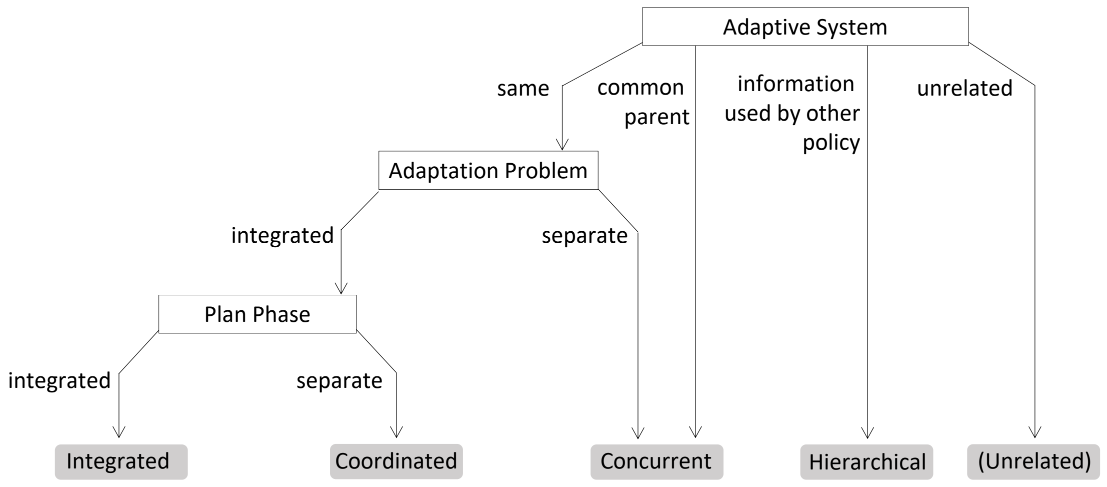


Figure 2.7: Classification of ways to create hybrid policies from [405].

2.3.3 *Hybrid Planning for Self-adaptive Software*

Hybrid planning for self-adaptive software in general refers to combing two or more adaptation policies to plan an adaptation. As discussed by Pandey [326], the notion of hybrid planning is inspired by the research field of *hyper-heuristics* that relies on the idea that as different heuristics yield different strengths and weaknesses, thus it is beneficial to combine them in a way that they compensate the weaknesses of each other [63]. The field in general is concerned with combining multiple lower-level heuristics and developing search algorithms or learning mechanisms for selecting or generating heuristics to solve computational search problems [63, 335, 356].

Based on a structured literature analysis on approaches for engineering adaptation policies, Trollmann et al. in [405] suggest a classification of ways to create hybrid solutions from multiple adaptation policies; the classification considers how the combined policies relate with respect to their corresponding adaptive systems, adaptation problem, and their planning phase. Figure 2.7 summarizes the classification of different hybrid policies based on the taxonomy provided in [405].

Integrated adaptation policies: (i) target the same adaptive system, (ii) are concerned with the same adaptation problem, and (iii) share the same planning phase; Therefore, employing an *Integrating Hybrid Planner*, the adaptation engine uses information from all the constituent policies during planning. [218, see].

A *Coordinating Hybrid Planner* combines adaptation policies that: (i) target the same adaptable system, (ii) are concerned with the same adaptation problem, but (iii) maintain separate planning phases. The planning phase of an adaptation engine employing *coordinated policies* can be subdivided into the original planning phases of the constituent policies. The hybrid planning approach by Pandey et al. [327] is an example for coordinated policies.

In a *Concurrent Hybrid Planner*, the combined policies: (i) solve different adaptation problems, (ii) are operated in separate MAPE-K loops, i.e., different plan phases, but (iii) target the same adaptive system directly or indirectly—see Figure 2.7. If the policies operate on the same adaptive system, they directly share the same adaptive system. However, the distinction between what is interpreted as the system's dynamic behavior and the adaptation carried out by other policies may be different. For instance, the adaptation performed by one policy may be considered as the dynamic behavior of the system by the other policy. In the indirect case, the involved policies operate on different (possibly overlapping) subsystems of the adaptive system that are all parts of a common, overall adaptive system. The approach of Vogel and Giese in [415] is an example of a concurrent self-adaptation approach in which a model of an adaptive system is synchronized with multiple views on that model and each partial model has its own adaptation mechanism.

In a *Hierarchical* combination of adaptation policies, the adaptive system that is managed by one policy is part of the information that is used by the other policy. The subsequent adaptation policies in the hierarchy are not concerned with adapting the same adaptive system but to modify, i.e., adapt, the way in which their precedent policy adapts the system. Sharifloo et al. in [379] use a hierarchical combination where one policy adapts the other policy.

2.4 MDE FOR RUNTIME ADAPTABILITY

The wide conceptual gap between the problem and the solution domains makes development of complex software systems that should, among others, run in distributed and embedded environments, target various devices and platforms, and operate dependably, a nontrivial task [375]. The problem domain refers to the concepts in the application domains such as cyber physical systems, IoT, or health, while the solution space refers to the space of computing and implementation technologies where the solution is developed in the form of components, classes, or methods [42]. Employing solutions that require extensive handcrafting of implementations to bridge the gap have been shown to increase the accidental complexities [see 57], hence, give rise to the cost and complexity of software development. Consequently, efforts towards industrializing the software development have been put forward [148].

MDE is primarily concerned with raising the level of abstraction in software engineering, hence automating the development process via using concepts that are closer to the problem domain rather than the ones offered by the programming languages [377]. To this end, MDE leverages technologies that support systematic transformation of problem-level abstractions to software implementations [148]. According to France and Rumpe [148], in MDE-based development of software systems, models describing the system at multiple levels of abstractions constitute the primary artifacts. Models at different levels of abstraction provide vertical views of the system [377], e.g., models describing the system's goals, architecture, or deployment—see [35, 88]. Models that describe specific system aspects, e.g., performance or security [see 251] provide horizontal views of the system [377].

The MDE vision of software development is generally realized via creation and use of *development models* that are models of software at levels of abstraction above the code level [148]. MDE employs *modeling languages* to express the various models that reflect concern-specific views of the system [251]. The abstract syntax of a modeling language is specified by a *metamodel* that defines the concepts of the language and how these concepts can be combined to create a model [375, 376]. Bézivin [41] describes the relation between a model and its metamodel similar to the one between a program and the programming language in which it is written. A model created with a modeling language represents an *instance* of the corresponding metamodel [21].

Applying *models@run.time* extends the use of MDE principles from development-time to runtime environments. In contrast to the development-time models, RTMs are used to reason about the operating environment and runtime behavior via providing aspect-specific views of an executing system and thus abstractions of runtime phenomena [18, 35]. Leveraging the benefits of MED at runtime such as abstraction, automation, and analysis, allows for model-driven management and adaptation of software during its execution.

As discussed by Blair et al. [47], RTMs can be used to fix the design errors via modifying the initial design or to support controlled ongoing design of a running system via enacting new design decisions. In the light of this vision for RTMs, Blair et al. [47, p. 23] define a *Runtime Model* (RTM) 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". The definition can be further contrasted with respect to the notion of *computational reflection* [294].

Computational reflection enables a system to reveal selected details of its implementation [294]; any computational system is concerned with a domain, which corresponds to the type of the application domain (problem space) addressed by the system [9]. The domain is realized by the system via structures representing the domain, i.e., the domain model. An important property of any computational system is the *causal connection* between the domain and its representing model [9]. The causal connection requires that the model and the corresponding domain are linked in a manner that changes on one side lead to a corresponding effect upon the other [294]. A *reflective* computational system is a system which incorporates structures representing (aspects of) itself; the structures are then referred to as *self-representation*s of the system; the self-representation is causally-connected to (aspects of) the system it represents, thus the system has an accurate representation of itself. Additionally, the status and computation of the system are always in compliance with the self-representation. This means that a reflective system can actually bring modifications to itself by virtue of its own computation. As a result, a reflective system is defined as a computational system which is about itself in a causally connected way [100].

Maes [294] describes two approaches for computational reflection: *procedural* reflection that considers the implementation of the system directly as self-representation implying that computations about the system are performed at the abstraction level of the system implementation [267]; *declarative* reflection, conversely, considers a standalone and independent entity as an explicit self-representation of the system which is different from the implementation of the system and can be considered as an RTM. These two cases represent the two end-points on a self-representation continuum [9]. Declarative reflection has been adopted by threads of research, particularly, the software architecture [166, 322], model-driven engineering [37], and requirements engineering fields [305], that aim at raising the level of abstraction of runtime representation by considering explicit RTMs [36, 38, 315].

Building on the computational reflection, models@runtime are the causally connected self-representations at the higher levels of abstractions that emphasize the structure, behavior, or goals of the system from a problem space perspective—see the definition by Blair et al. [47]. The definition conforms to declarative reflection defined by Maes [294] that considers abstract statements about the system, e.g., what the system behavior is instead of how the behavior is implemented. The causal connection between the system and its self-representation, i.e., RTM, guarantees that explicit representation of the system and its implicitly obtained behavior are consistent with each other [144, 310]. This implies that, owing to causal connection, RTMs constantly mirror the underlying system and its current state and behavior; any changes in the system is reflected in the RTM and vice versa.

As discussed earlier in Section 2.3.2, adapting a software system at runtime is realized in a multi-dimensional problem space including temporal aspects, structural features, goals, and requirements [323]. The complexity caused by the wealth of information associated with runtime phenomena poses as a particularly challenging problem [18, 310]. Developing adaptation mechanisms that leverage MDE and software models at runtime emerged as a promising solution [47]. Models@runtime that provide abstract information on runtime phenomena during execution enable the development of technologies that automate runtime decision-making and safe adaptation of the runtime behavior or structure.

In a model-driven adaptation scheme, the adaptation mechanism operates on a model of the system and its context. Models@runtime, analogously to the computational reflection, tend to focus on either the *structural* or the *behavioral* aspects of the underlying system [114, 323]. Structural models reflect how the software is currently constructed in terms of its constituents and their state, e.g., components and their connections [135, 147, 172]. In contrast, behavioral models are associated with the dynamics of the system, i.e., how the system executes in terms of flows of events or traces [25, 139].

The causal connection between the system and the RTM allows for modifying the modeled system via model transformation [186] where model queries search for parts of the model that are to be altered via *in-place transformations* which correspond to desired alterations to the system. Owing to the the causal connection, the changes to the RTM are reflected on the system to maintain the two entities, i.e., the RTM and the system, consistent with each other. In-place model transformation is a rule-based modification of a source model resulting in a target model where both the source and target models are typed over the same metamodel [263].

Software adaptation can be generally conducted as parameter adaptation, i.e., modifying variables of a program or as structural adaptation, i.e., changing the software architecture by adding, removing or replacing components and connections among components [302]—see Section 2.3.2. In this context, many researchers advocate that software architecture provides an appropriate abstraction level for realizing software self-adaptation (e.g., [53, 161, 213, 272, 296, 321, 322]) because self-adaptation can be generally achieved by adding, removing, and reconfiguring components as well as connectors among components in the system [302]. An architecture model represents the system architecture as a graph of interacting components[2]. Nodes in the graph, termed components, represent the principal computational elements and data stores of the system: clients, servers, databases, user interfaces, etc. Arcs, termed connectors, represent the pathways of interaction between the components. This is the core architectural representation scheme adopted by a number of *Architecture Description Language* (ADL), such as Acme [164] and xADL [106].

The architecture supports *weak* and *strong* adaptation [367]; the former refers to changing the parameters of the architectural elements and the latter is concerned with structural changes of the architecture [302]. Similar to the structural properties of the

2 Although there are different views of architecture [96], in this thesis we are primarily interested in the component-connector view as it characterizes the abstract state and behavior of the system at runtime to enable reasoning about problems and courses of adaptation.

software system, the behavioral aspects of the software can also be represented at the higher level of abstraction [25]. For this purpose, the adaptation mechanism maintains a causally connected *architectural* RTM as part of its knowledge to represent the architecture of the adaptable system. A recent survey by Bencomo et al. [36] on the state-of-the-art in models@runtime revealed that the most common use of models@runtime includes structural RTMs at the architectural level, e.g., [73, 146, 165, 166, 207, 310].

The architectural model used to capture system commonalities does not describe a precise configuration of the components and the connectors to whom the adaptable software must conform, but rather a set of *constraints* on the way components may be composed [162, 163]. Supporting a broad class of adaptive changes at the architectural level implies simultaneous changing of components, connectors, and the topology in a reliable manner. This requires distinctive mechanisms and architectural formalisms [322]. The architectural models are often expressed in ADLs [303] and, therefore, in different languages as used for the implementation of the system. Examples of commonly used ADLs are Darwin [170], C2/xADL [107, 321], or Acme/ ABLE [163, 166].

2.5 RUNNING EXAMPLE: MRUBIS

Throughout this thesis we use the *Modular Rice University Bidding System* (mRUBiS) [414] as a running example. mRUBiS is an online marketplace on which users sell or auction products. It is derived from RUBiS, an open source benchmark to evaluate control theoretic adaptation as well as self-adaptive software systems with performance concerns ([see 332]) and is widely used to evaluate research ideas [120, 156, 226, 307, 308, 346]. mRUBiS extends RUBiS by adding new functionalities and modularizing its monolithic structure; the modularization enables the architectural adaptation of mRUBiS. The exemplar hosts arbitrary number of shops; each shop belongs to a tenant and can be configured differently and runs isolated from the other shops; thus the architecture of each shop is isolated from the architectures of the other shops; a shop in mRUBiS consists of 18 components that are individually deployed; all shops share the same component types but each have their own individually configured components.

The business objective of mRUBiS is to achieve high sales volumes for its tenants. Thus, the mRUBiS goal model includes high availability of the services and low response times for the customers. In order to satisfy these two goals, we employ architectural self-healing capabilities to automatically repair runtime failures that cause disruptions in mRUBiS services; this allows us to consider repair actions that adapt the architectural configuration of mRUBiS. For instance, to mitigate faulty components, they can be restarted, redeployed, or replaced with alternatives. In addition self-optimization may be employed to improve the performance of the shops by reconfiguring the system; we have extended mRUBiS in previous work [175] to support self-optimization options such as adding replicas for system components.

In order to add self-healing and self-optimization properties to mRUBiS, the software is equipped with a MAPE-K feedback loop that uses an architectural RTM of mRUBiS. Specifically, the model represents the runtime architecture of mRUBiS according to the deployment of mRUBiS on an application server. Figure 2.8 shows a simplified metamodel of mRUBiS (based on the widespread ECORE syntax [391]) which defines valid model instances [44]. The metamodel of the RTM captures the mRUBiS Architecture with a set of ComponentTypes that require and provide InterfaceTypes. For each Tenant, the same

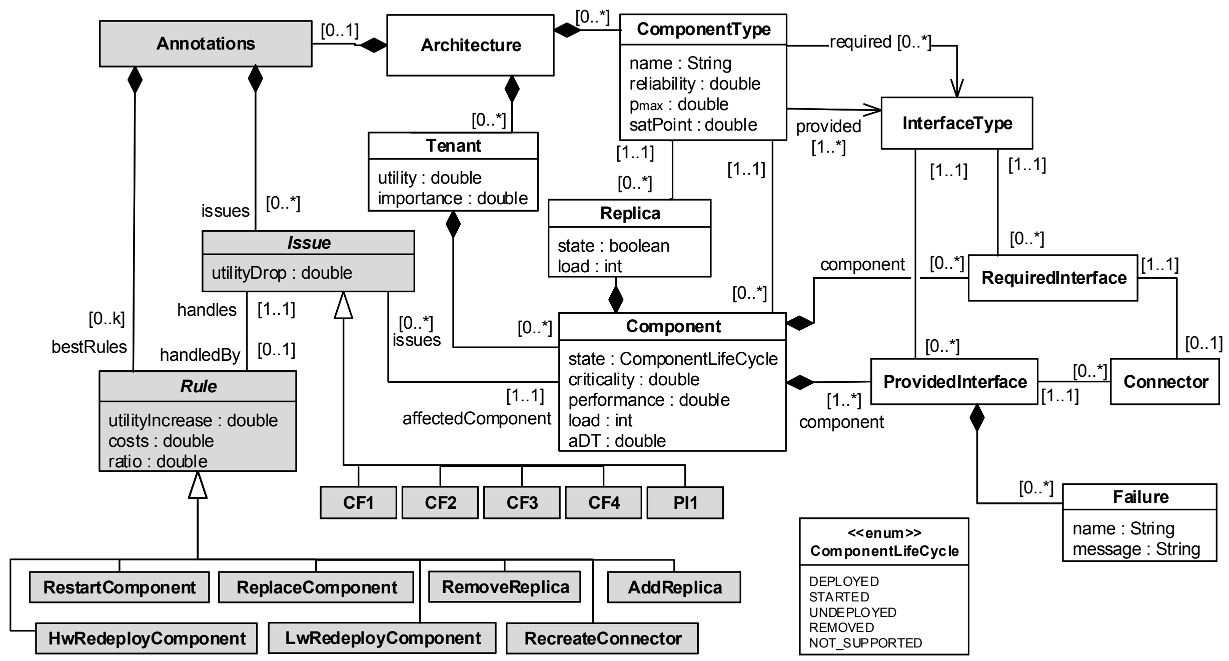


Figure 2.8: Simplified metamodel of mRUBiS.

component types are instantiated to Components with their Provided- and RequiredInterfaces. Components conforming to ComponentTypes constitute concrete configurations of mRUBiS. Components are parameterized by setting the value attribute of properties, e.g., criticality. A Component provides at least one interface and can require interfaces by means of functionality from other components. A Connector links a required and a provided interface if both are of the same InterfaceType. Using a ProvidedInterface of a component may cause Failures in terms of exceptions. A Component can have one or more Replicas that are of the same ComponentType as the original Component. The ComponentLifeCycle defines the state of a Component. These elements allow us to describe the runtime architecture of mRUBiS and the runtime issues, e.g., exceptions, failures, or sub-optimal configurations. The elements colored gray are relevant for self-adaptation and described later.

In this thesis, we use the self-healing and self-optimization properties of mRUBiS to discuss VENUS. To achieve high availability for the tenant in mRUBiS, self-healing aims at repairing architectural failures that disrupt the operation of mRUBiS. For the mRUBiS architecture, four classes of *Critical Failure* (CF) considered as introduced by Vogel [414]. They target Components that either crash and enter the UNKNOWN life cycle state (CF1), throw Exceptions exceeding a given threshold (CF2), are destroyed and removed from the architecture (CF3), and Connectors that are lost and removed from the architecture (CF4). To achieve low response times, the self-optimization aims at improving the performance of mRUBiS by architectural reconfiguration. We define one *Performance Issue* (PI), henceforth, PI1, that indicates the performance of a component in mRUBiS is below a threshold. PI1 may occur when the average load of a component changes.

As adaptation options, i.e., rules in Figure 2.8, mRUBiS supports restarting, redeploying, and replacing components, adding and removing service replicas, as well as recreating connectors. For the redeployment, there are two variants. The light-weight variant keeps the latest configuration while the heavy-weight variant resets the configuration parameters of the redeployed component.

2.6 GRAPH TRANSFORMATION

The automated process of taking one or more *source model*s as input and producing one or more *target model*s as output, following a set of *transformation rules*, is referred to as *model transformation* [377]. Model transformation rules are specifications that define a model modification and are expressed via the transformation languages in MDE. Examples of such languages are *Triple Graph Grammar* (TGG) [373], *Query/View/Transformation* (QVT) [320], and *ATLAS Transformation Language* (ATL) [236]. Transformation rules define, at the level of metamodels, how the target models are produced from the source models using the joint concepts of the source and target modeling languages, i.e., metamodels, for the specification of the transformation [189, 408].

Graphs and graph-based formalisims serve as a well-established prominent base for dealing with non-linear data structures [27, 124, 186, 188, 373, 377, 408]. Treating models as graphs where model entities are captured as vertices and connectors between entities as edges, allows for realization of model transformation via established graph operations such as *Graph Transformation* (GT) [377]. To this end, model transformation rules may be derived in the form of GT rules in accordance with the joint source and target contexts [124]. GT provides a pattern- and rule-based manipulation of graph models, which is frequently used in various model transformation tools, e.g., VIATRA [409], GREAT [412], and Henshin [16].

GTs are realized by the application of *GT rules* which are used for the evolution of graph structures [188]. A GT rule, also referred to as *graph rewriting rule*, consists of a graph to *match*, called *Left-hand Side* (LHS), and a replacement graph, commonly referred to as *Right-hand Side* (RHS). The LHS and RHS are *graph patterns* that describe the pre-condition and post-condition of the GT rule respectively. A match identifies an occurrence of the LHS pattern in a *source* graph G and the process of finding matches of the LHS of a GT rule in a graph is called *graph pattern matching*.

The application of a GT rule on a source graph G replaces a match of the LHS in G by RHS. A match for the LHS graph identifies a sub-graph in G that is eligible for the transformation; if a match is found, the corresponding GT rule is fired and results in the matched sub-graph of G being replaced by the RHS graph. Thus, each GT rule application transforms a graph by replacing a part of it by another graph [408]. In more details, this is performed by finding a match of LHS in the source graph G, via graph pattern matching, removing a part of the graph G that can be matched to LHS but not to RHS, and finally, creating RHS by adding the adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the target graph G′ [189].

2.6.1 *Adaptation via Graph Transformation*

RTMs support system adaptation at runtime by providing a means to represent and handle complex information captured in a non-linear format [34, 47, 396]. As discussed in Section 2.6, the rigorously defined formal semantics of graph transformation are well-established to support the complexity of models and model operations [27, 373, 397, 408]. Thus, in the context of self-adaptive systems and runtime adaptivity, the prac-

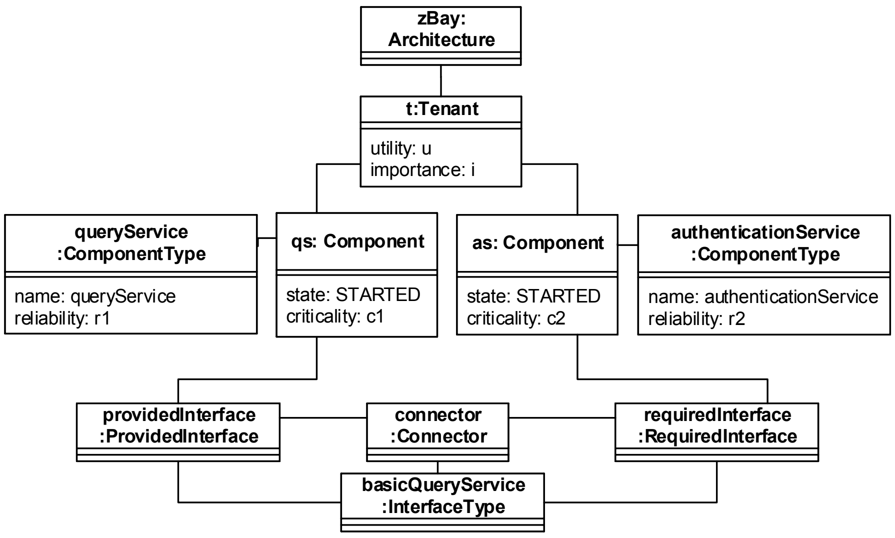


Figure 2.9: Exemplary architectural runtime model of mRUBiS.

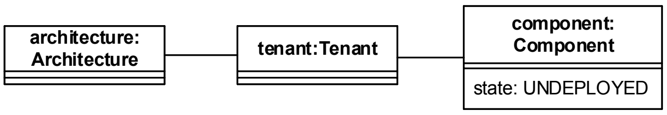


Figure 2.10: LHS of a transformation rule in mRUBiS.

tice of encoding RTMs as graphs has gained attention [32, 186, 415, 417, 418]. RTMs capture the structural as well as the behavioral dynamics of a causally connected system [47, 302]. Thus, most of the standard techniques for the formal description of systems such as automata or Petri nets are only of limited value in capturing structural dynamics [32]. Attributed graphs and graph transformations are in contrast a natural choice to capture structural changes of models [186]. Software architecture can be characterized as a graph of components and connectors [282, 322]. Thus, it is natural to capture the required modeling artifacts for architectural RTMs by graphs and graph transformations [115, 415].

Figure 2.9 depicts an architectural RTM capturing a fragment of the Architecture instance in mRUBiS. The RTM conforms to the metamodel in Figure 2.8, however, only *partially* instantiates the elements of the metamodel that are relevant for the intended context. The RTM includes one Tenant in mRUBiS with two Components in STARTED state. The components correspond to queryService (qs) and authenticationService (as) ComponentTypes respectively and are connected via a connector for basicQueryService InterfaceType. Owing to the causal connection, the RTM mirrors the state of mRUBiS. As discussed earlier, the architectural RTM can be captured as a graph, where the components are mapped to graph vertices and their properties are captured as the vertex attributes. Connectors between the components are modeled as graph edges—see [415].

Adaptation rules, specified either as ECA rules or Condition-Action rules, capture modifications to the adaptable software system and constitute the basic artifact of the adaptation [253]. Using (meta)models and MDE techniques for adaptation, adaptation rules are realized by model transformation rules, i.e., the equivalent model-based notion of GT rules–see Section 2.6. The *condition* in an adaptation rule, specifying a model (graph) *query*, is thus captured by the LHS of a model (graph) transformation rule. To characterize a model query, we use a pattern P describing a sub-graph of the source graph G, i.e., the architectural RTM. A *graph query* is the equivalent graph-based notion of a model query, i.e., a means to explore an existing graph. Typically, a simple graph query searches for a smaller graph, henceforth called a (graph) pattern, in the existing (queried) graph.

Since the architecture is represented by the RTM, we also use G to refer to the model. An occurrence of a pattern P in the model G corresponds to a match m of P in G (we write $G \models_m P$). For instance, a match identifies a failure in the architecture—see Figure 2.10 for an example of an LHS pattern in mRUBiS. The execution of the transformation rule including the specified LHS pattern in Figure 2.10 searches for the instances of class Component belonging to a Tenant in the Architecture that are in an UNDEPLOYED state.

A TGG specification is a declarative definition of a bidirectional model transformation that may be used to specify graph-to-graph transformation [373]. A TGG combines three graph grammars (LHS, CG, RHS) where LHS and RHS represent source and target graph structures respectively and are linked to each other by means of an additional *Correspondence Graph* (CG). The CG stores relationships between the corresponding source and target graphs. In the context of self-adaptive systems and causally connected RTMs, TGG rules capture the formalism for supporting the incremental propagation of changes among two models in both directions. TGG rules specify by means of model transformation rules how the RTM, reflecting the running system, is synchronized with the adaptable software [415, 416].

An *adaptation issue*, that is a phenomenon that triggers an adaptation, can be realized as the condition of an adaptation rule and, consequently, by the LHS of a GT rule. In this contexts, rules query the architectural RTM to identify adaptation issues, e.g., runtime failures. An adaptation rule uses the LHS patterns and their identified matches to localize the adaptation issue. Meanwhile, the RHS of the transformation rule determines how to modify the model enacting the *action* part of the adaptation rule and thus adapts the system through causal connection to resolve the detected issue.

Part II

APPROACH

This part presents the technical contributions. We divide the contributions to three main building blocks that collectively constitute our solution for architecture-based adaptation of software systems—see Figure 1.2 for an overview. First, defining utility functions for dynamic software architectures that enable utility-based adaptations; second, introducing an incremental solution that combines the said utility functions with the adaptation rules to engineer the Analyze and Plan activities of the adaptation loop; third, building on the utility function for dynamic architectures and the incremental analysis, proposing an alternative solution for planning that coordinates existing adaptation policies at runtime. The contributions are embodied in a MAPE-based adaptation engine that is executed incrementally. The incremental Analyze and Plan activities are complemented to build a complete adaptation loop by using an incremental monitoring scheme from the exiting work and including an incremental Execute activity. We leverage utility theory to incrementally evaluate an architectural RTM of the adaptable software as well as the adaptation plans. Utility functions in general can be obtained in two ways; analytically engineered from domain knowledge or gradually learned from the observations. We present our approach to attain both practices for utility elicitation in the context of dynamic software architectures. Having defined utility functions for dynamic software architectures, we discuss Venus, a utility-driVen rUle-based scheme to engineer architecture-based self-adaptive software. Venus uses graph pattern-based characterization of the adaptation issues, adaptation rules, and utility functions as its underlying principles. The scheme embodies ECA rule-based constructs and relies on utility theory for decision-making. Different complexities of software self-adaptation problem space require different solutions in terms of expressiveness, development effort, quality, and cost of the adaptation. Building on the incremental platform for monitoring, analysis, and execution in the adaptation engine, we proposes HypeZon as an alternative solution for planning. HypeZon is a Hybrid PlannEr for self-adaptation employing receding horiZon. HypeZon exploits the notion of meta-self-awareness to provide the hybrid planner, realized in an additional, higher-level control loop with increased awareness. Compared to Venus, the scheme requires less development effort and addresses the quality-cost trade-off by coordinating multiple off-the-shelf adaptation policies at runtime.

3 UTILITY FUNCTIONS FOR DYNAMIC SOFTWARE ARCHITECTURE

In this chapter, we discuss the *Utility Function* building block of the proposed solution for incremental architecture-based self-adaptation of software systems—see Figure 1.2. The utility function contributes to the Analyze and Plan activities of the adaptation engine—see Figure 3.1 for an overview. First, we discuss how to define utility functions for large, dynamic architectures based on architectural patterns Section 3.1. Then, in Section 3.2 we describe our approach to engineer a white-box decision-maker where we analytically construct utility functions based on the domain knowledge. Next, we present in Section 3.3 a methodology to systematically train utility-change prediction models for black-box systems where a detailed knowledge of preferences or system attributes is not available. The proposed methodology addresses the $\mathcal{R}6$ requirement in Table 1.1 for architecture-based self-adaptation regarding the initially unknown runtime knowledge of user and system preferences. Solution addresses problem of initially

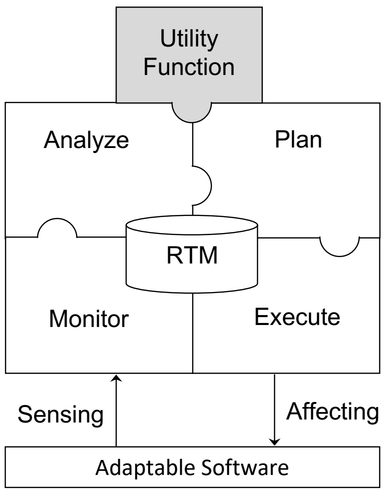


Figure 3.1: Chapter overview: Utility Function for adaptation engine.

unknown runtime knowledge of user and system preferences

3.1 PATTERN-BASED UTILITY

Utility functions can provide a general, principled, and pragmatic basis for utility-driven decision-making in self-adaptive systems [421]. A *utility function* $U(X)$ is an objective function that expresses how well each system state X satisfies the functional and non-functional requirements of the system. The state X can be characterized as a multi-dimensional property $X = \{X_1, X_2, ..., X_n\}$ where X_is represent constituent dimensions of state X. In this thesis, we focus on the architecture-based adaptation of software systems, thus we capture the *state* of the system via a *snapshot* of the architectural configuration of the system. In the context of architecture-based self-adaptation, utility function $U(G)$ assigns a real, scalar value to any system configuration G and has a range of $U(G) \in (-\infty, +\infty)$. The utility value assigned to configuration G indicates the desirability of G according to system objectives—see Section 2.2.1. This quantification allows for comparisons between different architectural configurations; the adaptation then can be steered towards obtaining the configuration yielding highest utility value. Furthermore, the *reward*, that accounts for the accumulated utility over time, supports comparisons of different architecture configurations over time.

Defining a *representative* and *correctly formulated* utility function is essential for the optimization process that is concerned with finding the set of parameters that maximize the said function. In a utility-driven self-adaptation mechanism, it is the utility function—and not the real utility of the system—that is being constantly optimized to obtain an adaptation decision. Utility-driven adaptation can be carried out through an extensive search in the solution space where the impact of the different properties on the overall system utility, resulting in different configuration settings, is evaluated prior to the decision-making; the configuration yielding the highest utility is then chosen—see [146]. In addition to the internal configuration of the system, G may also include set of features that reflect the impact of the environment on the system or its context. Such features however, while having an impact on the utility, cannot be adapted. Hence, the search within the solution space for maximizing the utility for G is limited to what can be adapted.

In the specific context of architecture-based self-adaptation, a typical approach is to use normalized[1], linear[2] utility functions that for each non-functional property, e.g., reliability or content quality, compute the impact of the alternatives options, providing similar functionality at different quality levels, on the overall system utility [90, 91, 146]. Given a concrete architecture with concrete alternative options selected, the utility function then computes the weighted sum of these impact values over all properties where the weights represent the preferences and the result is the utility of the given architecture [12, 39, 421]. However, defining such utility functions is particularly challenging for large, dynamic, and highly configurable software architectures where an extensive search in the solution space before each adaptation decision hinders scalability and renders the optimization process time-intensive.

In this section, we describe our methodology to define utility functions for large, dynamic architectures based on patterns. We introduce the notion of *positive* and *negative architectural utility patterns* indicating the impact of the fragments of the architecture, i.e., patterns, on the overall system utility. For this purpose, we present in the following two requirements that should hold for utility functions that are used to evaluate the software architecture:

Req 3.1 *the optimal architectural configuration that fulfills the system objectives, at the most desirable level, yields the maximum utility.*

Req 3.2 *any violation of system objectives leads to a decrease in the system utility.*

1 A normalized function is one where the integral is equal to one over the entire domain.
2 A linear function is a function whose graph is a straight line, that is, a polynomial function of degree zero or one.

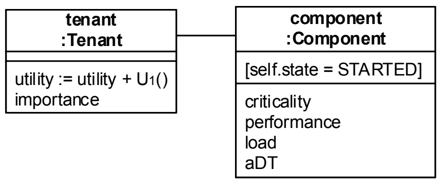


Figure 3.2: Positive architectural utility pattern P_1^+.

3.1.1 *Positive Architectural Utility Patterns*

According to ReQ3.1, we include the impact of *present* architectural fragments in the definition of architecture-based utility. We define the fragments as *positive architectural utility patterns* $\mathcal{P}^+ = \{P_1^+, \ldots, P_k^+\}$ and capture their *positive* impact on the overall system utility by utility sub-functions U_i. Thus, when a positive architectural utility pattern P_i^+ is present in the software architecture, it contributes a value defined by its corresponding utility function U_i to the overall system utility. Recall from Section 2.6.1 that an occurrence of a pattern P in the model, i.e., the architecture G, corresponds to a match m of P in G (we write $G \models_m P$). The obtained utility by the match m of the positive pattern P_i^+ in G is denoted as $U_i(G, m)$.

A positive architectural utility pattern, henceforth, positive pattern, may include fragments of the system architecture containing a single component or multiple components and connectors. Moreover, the patterns may be defined as *generic* or *component-specific* positive patterns. The impact of a pattern P_i^+ is defined by U_i and, depending on the specific *context* for individual occurrences of the pattern, may vary for *each* match of the pattern in the architecture. Thus, the utility sub-functions U_i are context-dependent and may ascribe different utility values to identical patterns with different context. In general, any information that is observable at runtime and is represented in the RTM of the software architecture can serve as the context of an architectural fragment and be used for computing the utility values. Note that the context for utility calculation is not necessarily required to be part of the original pattern. At runtime, the context of a matched pattern is dynamically obtained from the RTM when evaluating the corresponding utility sub-functions U_i.

We define the overall system utility, i.e., the utility of the system architecture G, based on the principles of additive utility functions—see Section 2.2.1.1. In order to apply the concept of the additive utility, the architecture G should be decomposed to fragments that comply with the mutual utility independence. Let $G = \{g_1, g_2, ..., g_n\}$ where g_is represent architectural fragments of G with $\forall i \neq j \implies g_i \neq g_j$. Any decomposition of G to g_is that fulfills the mentioned requirement and conforms to the mutual utility independence can be considered for the calculation of the additive overall utility of G as below:

$$U(G) := \sum_{i=1}^{n} U_i(g_i) \tag{3.1}$$

Thus, conforming to the additive utility principles, when matching a positive utility pattern P_i^+ in the architecture G, the overall utility of the system is increased by U_i.

Figure 3.2 shows an example of a positive pattern P_1^+ and the corresponding utility sub-function U_1 in mRUBiS. This pattern conforms to the metamodel in Figure 2.8 and includes a component that is associated with a tenant and therefore, contributes to the functionality of the tenant. The pattern targets a single component and is generic as it refers to *any* started component. When matching this pattern for one component in the architecture, the utility of the associated tenant increases by U_1. For any match m of the positive pattern P_1^+ in Figure 3.2, let the utility sub-function U_1 be defined as $U_1(G, m)$:= criticality of the component $\times$ reliability of the corresponding component type $\times$ connectivity of the component—see the metamodel of mRUBiS in Figure 2.8 for reference.

While the pattern in Figure 3.2 involves a Tenant and a Component in STARTED state, in addition to the information directly observable by the match, the context for the computation of the corresponding utility function U_1 involves the reliability of the ComponentType and the number of the ProvidedInterface and RequiredInterface of the matched Component, i.e., connectivity of the Component. Figure 3.3 visualizes (in gray) the required context for utility calculation for a match of P_1^+ in Figure 3.2. Thus, pattern P_1^+ and the corresponding utility sub-function U_1 determine the context of a matched component as: criticality of the component , reliability of the component type, and connectivity of the component. The three attributes define the impact of an occurrence of P_1^+ on the utility of the associated tenant.

In mRUBiS, individual tenants are independent of each other, thus comply with the mutual utility independence. In this context, the mutual utility independence of tenants implies that the utility of each tenant is independent of other tenants. We define the utility of architecture G with l tenants in mRUBiS as :

$$U(G) := \sum_{i=1}^{l} U_i(tenant_i) \tag{3.2}$$

Similarly, for each tenant in mRUBiS, we assume utility independence between it constituent components and define the utility of a tenant as the sum of the utility of its 18 components—see Section 2.5. Once all 18 components of a tenant are matched with the positive pattern described in Figure 3.2, the utility of the tenant is the sum of the corresponding sub-utilities $U_1(G, m)$ for all of the matched components. Finally, the pattern is applied to all the tenants in mRUBiS to obtain the utility for each tenant and then, collectively, for the whole architecture G.

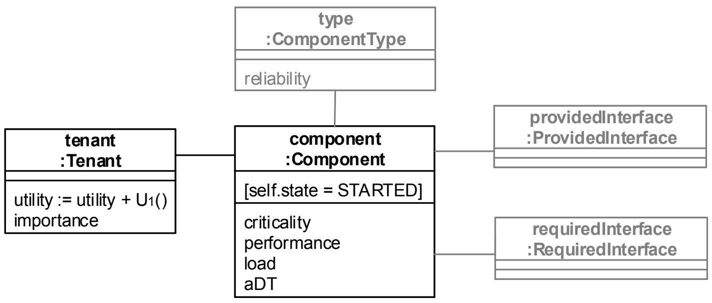


Figure 3.3: Positive architectural utility pattern P_1^+ and relevant context (in gray) for utility calculation.

3.1.2 *Negative Architectural Utility Patterns*

According to ReQ3.2, we include the impact of the undesired architectural fragments in the definition of the architecture-based utility, titled *negative architectural utility patterns* with $\mathcal{P}^- = \{P^-_{k+1}, \ldots, P^-_n\}$. Negative architectural utility patterns, henceforth, negative patterns, negatively affect the system utility and cause a reduction in the overall utility of the architecture G, i.e., $U(G)$, relative to their corresponding utility sub-functions. An occurrence of a negative utility pattern P^-_j in the architecture G is denoted by a match m in G (we write $G \models_m P^-_j$). $U_j(G, m)$ is the impact of a match m of P^-_j defined by its utility sub-function U_j. Examples of such negative patterns are occurrences of runtime failures, e.g., exceptions.

We define the utility sub-function U_j of a negative pattern P^-_j as a negative real value—remember from earlier $U(G) \in (-\infty, +\infty)$. Thus, if occurrence of a match for pattern P^-_j changes the architecture from G_1 to G_2, the system utility changes from $U(G_1)$ to $U(G_2)$. Conforming to the additive utility definition, $U(G_2) = U(G_1) + U_j$ with $U_j < 0$. $U(G_2)$ can be equivalently rephrased as $U(G_2) = U(G_1) - |U_j|$. Consequently, while the positive patterns capture the possible utility gain by the current architectural configuration, the negative patterns represent whether the potential for utility gain can be actually realized. The potential for obtaining the (maximum) possible utility gain is not realized if there is a match for a negative pattern in the architecture that correspondingly decrease the utility—see ReQ3.1. Therefore, considering $M_i(G) = \{m \mid G \models_m P_i\}$ as the set of matches for the pattern P_i in the current architecture G, the overall utility function $U(G)$ accumulates all the effects due to the matches of all n patterns $\mathcal{P} = \{P_1, \ldots, P_n\}$—see Equation 3.3. If we do not have to distinguish between positive and negative patterns, we omit the superscript + and − for the patterns $P \in \mathcal{P}$.

$$U(G) := \sum_{i=1}^{n} \sum_{m \in M_i(G)} U_i(G, m) \tag{3.3}$$

Analogously to the positive patterns, negative patterns may also be defined in a generic manner where they can be matched by any generic entity of the same class indicated by the pattern, e.g., any component or connector. The negative patterns may also be defined in a component-specific manner where they may only be matched to specific instantiation of the entities, e.g., only components of the type query management service. Negative patterns are context-dependent and, based on the specific context in the architecture G, the impact of a negative pattern P^-_j may vary for each individual match of the negative pattern in G. Figure 3.4 shows an example for a negative pattern P^-_2 in mRUBiS. According to P^-_2, the occurrence of five or more failures, e.g., exceptions,

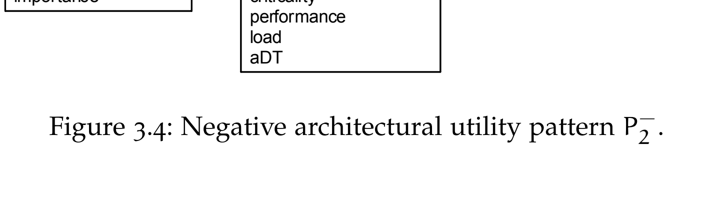


Figure 3.4: Negative architectural utility pattern P^-_2.

in a Component in STARTED state marks a negative utility pattern in the architecture. Each match of the negative pattern reduces the utility of the corresponding Tenant by U_2 where $U_2(G, m)$ is a negative value.

The definition of the *pattern-based utility* takes the context into account. In general, when matching a positive or negative pattern P_i, the concrete context for each match is dynamically identified in the architecture. The context corresponds to a fraction of the architecture that is navigated to obtain the required information for calculating the pattern-based utility at runtime. Each pattern P_i specifies a context that influences the corresponding utility sub-function U_i and thus the increase or decrease of the utility. For instance, each of the patterns P_1^+ in Figure 3.2 and P_2^- in 3.4 specify the criticality of the Component and the associated Tenant as part of their contexts. This context could be extended, for instance, by taking the ComponentType into account—see Figure 3.3—such that the pattern would only match to components of certain type, e.g., components of the type authentication or user reputation. Finally, the individual context of each match of a pattern could cause variations in the final (expected) utility after the adaptation, thus plays an essential role in steering the decision-making mechanism. We discuss this in Section 4.2.

Figure 3.5 shows an excerpt of the mRUBiS architecture; tenant t1 has two matches of the positive pattern P_1^+ introduced in Figure 3.2, i.e., two STARTED component. The first match includes t1 and the reputation component; t1 and the authentication component constitute the second match for P_1^+ in tenant t1. Tenant t2 includes only a single match for the positive pattern P_1^+—the three matches of the positive pattern in the architecture are highlighted in different shades of gray. Each match m for P_1^+ increases the utility of the corresponding tenant by $U_1(G, m)$ and by taking the context of the matched components into account—see Figure 3.3 for the required context to calculate the utility of the matched P_1^+. The utility of a tenant is the sum of $U_1(G, m)$ for all matches of P_1^+ in the tenant, consequently, the utility of the zBay architecture is the sum of the utilities of t1 and t2.

In addition to the matches for the positive pattern P_1^+, t1 in Figure 3.5 also includes a match for the negative pattern P_2^- (see Figure 3.4) caused by the providedInterface of the authentication component that has 6 exceptions. The match for the negative pattern is marked with a dash-lined box in Figure 3.5. The match for the negative pattern reduces the utility of t1 as well as the utility of the architecture. In Figure 3.5, $U_2()$ represents a negative utility associated with the match for P_2^-. Therefore, for the given architecture, the overall utility of tenant t1 comprises of two times $U_1()$, one for each match of the positive pattern P_1^+, and one time $U_2()$ for the negative pattern match.

In this section, we introduced the notion of pattern-based utility and described how *utility values* are assigned to the architectural fragments. We impose no restrictions on the size of the considered architectural fragments, i.e., patterns; a fragment may include only a single component or may be as large as the whole system architecture. By assigning utility values to architectural fragments, we are enabled to evaluate different architectural configurations in terms of utility. The ability to quantify the desirability of a configuration supports the decision-making through out the adaptation process by providing a common basis for comparison between different adaptation options that lead to different next configurations. The pattern-based definition of utility supports dynamic architectures that may change in size or complexity at runtime, for the context of the pattern-matching can also be dynamically extended at runtime.

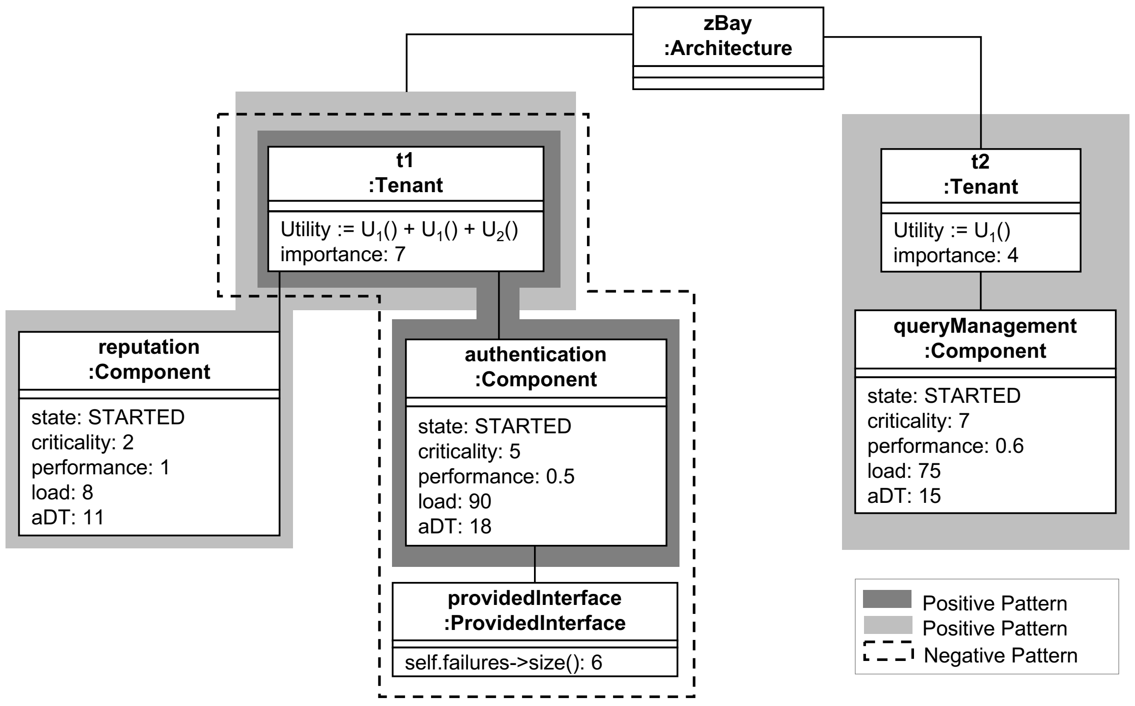


Figure 3.5: Excerpt of mRUBiS architecture including matches for positive and negative patterns.

3.2 ENGINEERING UTILITY FUNCTIONS

This section discusses how to analytically engineer mathematical models[3] that map a set of input parameters to a scalar value, namely, the utility value. In general, for a software system, the utility function U can be obtained from SLAs, through user preference elicitation, or based on templates [see 421]; utility of system describes the degree to which system objectives are satisfied by the input parameters.

3.2.1 *Utility Space*

Defining utility functions for self-adaptive software requires recognizing three types of attributes that construct the *utility space* for architecture-based self-adaptation: *configuration attributes*, *observation attributes*, and *quality attributes*. Configuration attributes can be identified as a vector of configuration settings $\bar{c}$. These attributes typically specify the *structural* characteristics of the architectural elements, i.e., components and connectors, as well as their *descriptive* properties that have no influence on the composition of the system. For instance, the criticality of a component, indicating its importance for system functionality, is not a structural property, however, represents a configuration attribute in this definition. In mRUBiS, the running example through out this thesis—see Section 2.5—examples of configuration attributes include state, criticality, and connectivity of a Component—see Figure 2.8. While state and connectivity, that is the sum of providedInterface and RequiredInterface, are structural properties, criticality is a behavioral property. The

3 Functions are rules that express the dependency of one variable quantity, i.e., output, on one or more variable quantities, i.e., input, and are considered as simple kinds of mathematical models [see 98], thus we use models as a general term here.

architecture-based adaptation of the software then is achieved via changing the current configuration $\bar{c}$ to the new configuration $\bar{c}'$.

In addition to the internal configuration of the software systems captured via configuration attributes, the utility space is also affected by the operational environment of the system. The observation attributes are set of features that can be either measured directly or synthesized from sensor measurements and reflect the impact of the environment on the system or its context [421]. An observation is a vector of observation values $\bar{o}$ that are monitored at runtime. An example of an observation attribute is the component load—see the metamodel of mRUBiS in Figure 2.8. The current observations $\bar{o}$ may change to new observations $\bar{o}'$ independently of the configuration changes. A collection of the configuration and observations attributes, i.e., $(\bar{c}, \bar{o})$, constitute the *state* of the adaptable software at each point in time [59].

Finally, there are quality attributes that represent properties of the *service* delivered by the system to the user. The service delivered by a system is its behavior as it is perceived by its user(s). A user is another system—physical or human—which interacts with the former [281]. Quality attributes capture the evaluation of the system operation and are orthogonal to system functionality and often pervade the structure of the system [191]. Software quality is then recognized as the degree to which software possesses a desired combination of its quality attributes, e.g., reliability, performance, and response time—see [217]. Quality attributes are also recognized as *Key Performance Indicators* (KPIs) [354]. In the context of mRUBiS, instances of quality attributes are reliability of ComponentType, i.e., continuity of service, and performance of Component, i.e., timeliness of the service delivered by the component—see [385]. We denote a vector of quality attributes as $\bar{q}$ which may depend on both internal and external properties captured by the configuration $\bar{c}$ and observation $\bar{o}$ attributes respectively. The set of quality attributes $\bar{q}$ may change to new quality attribute values $\bar{q}'$ in response to changes in the configuration or observation.

Each quality attribute captures a business objective of the system by mapping it to a monitored architectural property. The overall quality of the system, i.e., the overall system utility, is a quantitative measure of its quality attributes. Thus, a utility function is a mathematically formulated function that maps each possible state of an entity, i.e., the whole system, a single component, or a sub-set of system components, into a real scalar value representing its desirability in terms of business objective satisfaction [421]. In this context, we define $U(\bar{c}, \bar{q}, \bar{o})$ with U the utility of the system as a function of configuration attributes $\bar{c}$, quality attributes $\bar{q}$, and observation attributes $\bar{o}$.

Quality attributes are either directly observable from the system, e.g., response time, or can be computed based on a set of relevant configurations $\bar{c}$ and observations $\bar{o}$, e.g., performance. For complex systems however, it is often not trivial to obtain the relevant values for $\bar{c}$, $\bar{q}$, and $\bar{o}$ to compute a well-behaving [4] $U(\bar{c}, \bar{q}, \bar{o})$. The reason is that, it is not trivial to trace the effect of the changes—as a result of the adaptation—on the system quality attributes. Thus, we often do not know which quality attribute values $\bar{q}$ result from which changes. Instead, we may only observe the system after the adaptation, and thus only know the resulting configuration $\bar{c}'$ and the usually stable

4 In this context, a well-behaving utility function is the one that calculates the utility values in a representative manner.

Table 3.1: Data schema for utility space.

Field	Symbol	Example
Configuration attributes	$\bar{c}$	State, connectivity
Observation attributes	$\bar{o}$	Load, disruption
Quality attributes	$\bar{q}$	Performance, response time
Quality model	$S(\bar{c}, \bar{o})$	Performance = $\frac{\text{fullfilled tasks}}{\text{consumed resources}}$
Utility function	$U(\bar{c}, \bar{q}, \bar{o})$	Linear, sigmoid

observations $\bar{o}$. The solution is to calculate the quality attribute values $\bar{q}$ by means of a model S that takes configuration $\bar{c}$ and observations $\bar{o}$ as input—see below:

$$U(\bar{c}, S(\bar{c}, \bar{o}), \bar{o}) = U(\bar{c}, \bar{q}, \bar{o}) \tag{3.4}$$

This way, we obtain the utility function $\hat{U}$ based on model S for the quality attributes as described below:

$$\hat{U}(\bar{c}, \bar{o}) = U(\bar{c}, S(\bar{c}, \bar{o}), \bar{o}) \tag{3.5}$$

Table 3.1 summarizes the artifacts that construct the utility space. Finally, we define the utility space of a software system, comprising of a vector of configuration attributes, a vector of observation attributes, and a vector of quality attributes as Equation 3.6.

$$\begin{aligned} \bar{c} &\equiv \{c_1, c_2, ..., c_l\} \\ \bar{o} &\equiv \{o_1, o_2, ..., o_m\} \\ \bar{q} &\equiv \{q_1, q_2, ..., q_n\} \end{aligned} \tag{3.6}$$

As discussed earlier, $q_i \in \bar{q}$ may be observed directly or calculated via a quality model S with $\bar{c}$ and $\bar{o}$ as inputs; in this case, $\bar{q}$ is defined as:

$$\begin{aligned} q_i &= S_i(\bar{c}, \bar{o}) \\ \bar{q} &\equiv \{S_1(\bar{c}, \bar{o}), S_2(\bar{c}, \bar{o}), ..., S_n(\bar{c}, \bar{o})\} \end{aligned} \tag{3.7}$$

Thus, the utility space to construct the utility function $U(\bar{c}, \bar{q}, \bar{o})$ is a space of $\nu = l \times n \times m$ dimension. In the context of dynamic software architectures, that is, software that modify their architecture and enact the modifications during the system execution, the vectors $\bar{c}$, $\bar{o}$, and $\bar{q}$ cannot be captured by finite sets of l, m, and n elements respectively as shown in Equation 3.6. Therefore, the utility space for dynamic architectures is, by definition, unbounded and cannot be represented via static, finite sets as shown above.

3.2.2 *Utility-change*

A pair of a configuration attributes and the corresponding observations, $(\bar{c}, \bar{o})$, constitute the state of the adaptable software system at each point in time [59]. For a software system at a state $(\bar{c}, \bar{o})$, the associated utility function $U(\bar{c}, \bar{q}, \bar{o})$ is required to ascribe a scalar value describing the desirability of the system state. In the context of self-adaptive software systems, the system is adapted via changing the current configuration $\bar{c}$ to the new configuration $\bar{c}'$—see Section 2.3.2.

The adaptation of a software system changes the current state of the system, captured by $(\bar{c}, \bar{o})$, to a new state $(\bar{c}', \bar{o})$ by execution of an *adaptation action* $a \in A$ with A the set of available actions to adapt the system. We show this as $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$. During this process, changes in the system utility, i.e., *utility-changes*, indicate the differences between the previous and the current utility. The utility-change during $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$ is described as:

$$\hat{U}_{\Delta}(\bar{c}, \bar{o}, a) = \hat{U}(\bar{c}', \bar{o}) - \hat{U}(\bar{c}, \bar{o}) \tag{3.8}$$

By localizing the effects of action executions, we avoid the potentially costly computation of $\hat{U}(\bar{c}', \bar{o})$ for the whole architecture and only, incrementally, compute $\hat{U}_{\Delta}(\bar{c}, \bar{o}, a)$. To further support incremental utility-change calculation in this thesis, when available, we propose to explore the locality of the context for the adaptation action a. Instead of $\hat{U}_{\Delta}(\bar{c}, \bar{o}, a)$, a simpler function $\hat{U}_{\Delta}(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$ can be employed; the reason is that we assume a local effect for action a, i.e., executing action a changes a restricted scope in the architecture. Therefore, we do not need the full architectural configuration $\bar{c}$ and observations $\bar{o}$, instead, we can consider only the sub-vectors for the architectural configuration $\tilde{\bar{c}}$ and observations $\tilde{\bar{o}}$ which are only available if we can assume a local effect for execution of action a. The sub-vectors $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$ can be determined locally, and relative to the application context of the action a and the selection could be specific to each action a. In order to compute $\hat{U}_{\Delta}(\bar{c}, \bar{o}, a)$ in Equation 3.8, we compute $\hat{U}_{\Delta}(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$.

In software architectures where the side effect of executing an action a cannot be locally constrained, i.e., it has a global effect on the whole architecture, sub-vectors $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$ are equal to $\bar{c}$ and $\bar{o}$ respectively. Consequently, the effort for computing the utility-change $\hat{U}_{\Delta}(\bar{c}, \bar{o}, a)$ in Equation 3.8 would be equal to the effort for computing $\hat{U}(\bar{c}', \bar{o})$ since the scope of the change, caused by a, cannot be localized and includes the whole architecture.

3.2.3 *Utility Function Construction for mRUBiS*

We define a multi-objective utility function as an aggregation of multiple quality attributes where each attribute represents a business objective. The ultimate goal of a multi-objective utility function is to support quantitative evaluation and trading off multiple quality attributes to arrive at a better overall system. In this thesis, we start from quantification of individual quality attributes based on the architectural properties of the software and aggregate them to a single metric, that is, the multi-objective utility function. In computing aggregate values, we accumulate units of each measured/ observed attribute in the same manner, so a formal distinction is not needed. We then combine values by the sum or multiplication operators, depending on the employed mathematical model for U and regardless of the attribute types.

The analytical engineering of utility functions, i.e., preference elicitation, for a software system is often driven by a contract, e.g., SLA, or via querying the user(s), and entails mapping of business objectives to quality attributes. For this purpose, the constituent elements of the system utility space—see Table 3.1— are exploited to manually arrive at the best fitting utility function U that represents system objectives. We dis-

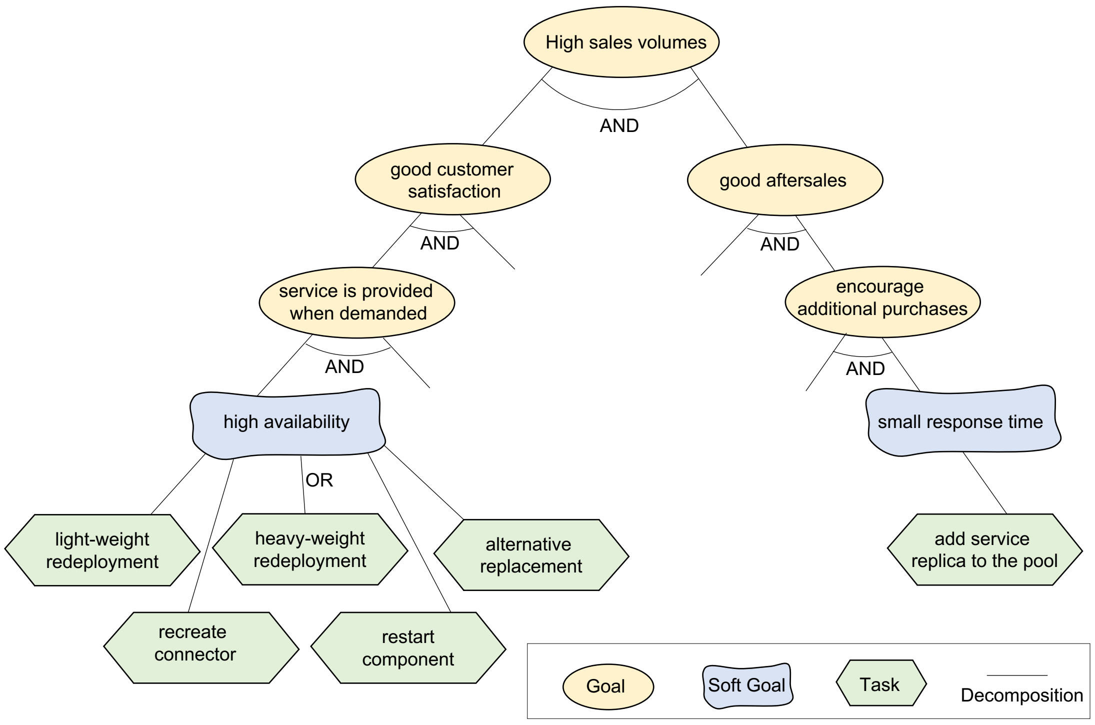


Figure 3.6: Excerpt of mRUBiS goal model.

cussed in Section 3.2.1 that the utility space of dynamic software architectures is a multi-dimensional, unbounded space. Next, based on contracts such as system goal models, SLAs, or informal knowledge of user preferences, a subset of the utility space dimensions are selected to construct the utility function U. At this part of the thesis, the utility construction is illustrated for our running example mRUBiS as system-specific goal models are required. In the following, we present four analytically constructed utility functions for mRUBiS that subscribe to four different mathematical complexity classes.

mRUBiS simulates a marketplace on which users sell or auction items. Similar to any e-commerce system, e.g., eBay and Amazon, the companies that are selling their

Table 3.2: Utility functions for mRUBiS and their complexity class.

U	**Complexity Class**
$\text{Reliability}\times\text{Criticality}\times(\text{providedInterface}+\text{requiredInterface})$	Linear
$\text{Reliability}\times\ \text{Criticality}\times\ P_{max}\times\ \tanh(\alpha\times\frac{\text{replica}}{\text{load}})$ $\times(\text{providedInterface}\ +\ \text{requiredInterface})$	Saturating
$\text{Reliability}\times\text{Criticality}\times(\text{requiredInterface}+1)\times\text{Importance}\times\beta$ $\times(\text{providedInterface})-10\times\text{ADT}$	Discontinuous
$\text{Reliability}\times\text{Criticality}\times\text{Importance}\times\beta\times(\text{providedInterface})\times P_{max}$ $\times\ \tanh(\alpha\times\frac{\text{replica}}{\text{load}})\times(\text{requiredInterface}+1)-10\times\text{ADT}$	Combined

products on mRUBiS aim for *high sales volumes* by achieving customer satisfaction and encouraging customers to additional purchases. Therefore, the e-commerce platform should be *highly available* and exhibit *small response times* [414]. See Figure 3.6 for a excerpt of the goal model of mRUBiS. The soft goals of high availability and small response times in the goal model are fulfilled via the adaptation of mRUBiS. In order to map the business objectives of mRUBiS to a representative utility function, we explored a range of complexity classes for the mathematical formulation of U summarized in Table 3.2.

In the context of mRUBiS, for any match m of the positive pattern P_1^+ in Figure 3.2, the utility sub-function $U_1(G, m)$ can be defined via one of the four variants of the utility functions in Table 3.2; the variants differ in their parameters, complexity, and employed operators. Moreover, having different parameters in the utility function formula implies different required context for utility calculation when pattern P_1^+ is matched. We will discuss this in more details when we explain the utility function variants.

Linear U: The linear variant, i.e., a polynomial of degree one [204], is a multiplication of criticality of the component, reliability of the corresponding component type and the connectivity of the component. This variant is used as the exemplary utility function in Section 3.1. Figure 3.3 illustrates the required context in the mRUBiS architecture to calculate the linear utility function. The connectivity is the sum of the providedInterface and requiredInterface of a component. As described in Section 2.5, components in mRUBiS have a criticality attribute denoting their relevance for a tenant—see the metamodel of mRUBiS in Figure 2.8. For instance, concerning the functionality of e-commerce services in general, the authentication service is considered more critical for the functionality of an online shop than the reputation service since the former is necessarily required by a shop to close a deal while the latter is not. In mRUBiS, the Authentication service is critical since it is required to have the buying and selling functionality while the Reputation component takes care of rating a seller or buyer after a deal which is not a critical matter and can be done later in time. Additionally, each component type has a reliability. Certain functionalities in mRUBiS can be provided by multiple, alternative component types with different reliabilities, e.g., local vs. various third-party authentication services; in these cases, selecting the most reliable alternative results in larger utility improvements. The connectivity of a component as the number of associated connectors indicates the importance and accordingly influences the utility of the component.

Saturating U: The linear variant of the utility function is extended by adding the quality attribute performance to its formula. The performance of a component is calculated using a quality model S—see Section 3.2.1 and is defined as:

$$Performance = S(\bar{c}, \bar{o}) := P_{max} \times \tanh(\alpha \times \frac{\mathsf{replica}}{\mathsf{load}}) \tag{3.9}$$

The performance attribute is also stored as a component attribute—see the metamodel of mRUBiS for reference in Figure 2.8. Using a hyperbolic tangent function for the quality model S to map the $\bar{c}$ and $\bar{o}$ attributes to performance provides a saturating effect and, as a result, the performance values are bounded—see Equation 3.10 for definition of the $\tanh(x)$. Figure 3.7 shows plots of the $\tanh(x)$ with different saturation gradients. In saturating functions, the initial stage of growth is approximately exponential; as the

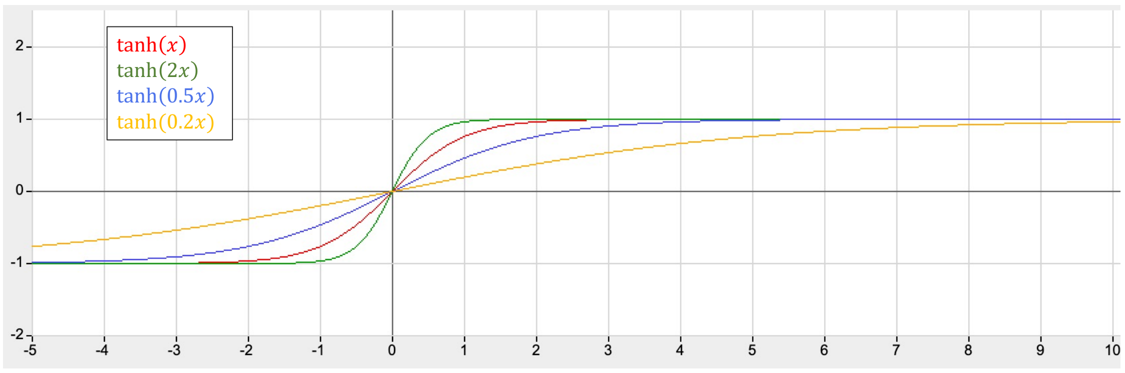


Figure 3.7: $\tanh(x)$ with different gradients.

saturation begins, the growth slows down and eventually stops at maturity. The sigmoid function [200] is another example for bounded functions with saturating phase.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.10}$$

As the mRUBiS metamodel in Figure 2.8 shows, each componentType has a P_{max} attribute that defines the maximum value to which the performance of the component saturates. A certain ratio of $\frac{\text{replica}}{\text{load}}$, captured by the satPoint attribute of componentTypes in the mRUBiS metamodel, is the saturating point of the hyperbolic tangent function in Equation 3.9 and defines the shape of the function. The satPoint is a configuration attribute of the system and may vary for each individual component type. Therefore, for each component type, the quality model S, defining the performance of the component, has a different shape. The variable α in Equation 3.9 defines the gradient of the hyperbolic tangent function for each componentType and is calculated as Equation 3.11.

$$\alpha = \frac{4}{\text{satPoint}} \tag{3.11}$$

In Equation 3.9, the hyperbolic tangent function only has positive input values, thus only the positive outputs of $\tanh(x)$ in Figure 3.7 are relevant for our context. For each component in mRUBiS, $\text{performance} = P_{max}$ when $\frac{\text{replica}}{\text{load}} = \text{satPoint}$. The satPoint and P_{max} are properties of component types and can be obtained from the system architectural model—see Figure 2.8.

Compared to the linear utility function, the saturating utility function in Table 3.2 extends the required context from the system architecture for calculating the utility. Figure 3.8 presents the context to calculate the saturating utility function for a match of pattern P_1^+. See the context for the liner variant in Figure 3.3.

Discontinuous $\mathcal{U}$**:** The discontinuous utility function extends the linear $\mathcal{U}$ in Table 3.2 in various ways; it introduces discontinuity to the function based on the component's providedInterface attribute. Discontinuity refers to the disconnected intervals in the utility function as a result of using the attribute β defined in Equation 3.12. The providedInterface of a component $\mathcal{C}$ in mRUBiS describes to which degree other components in the architecture depend on $\mathcal{C}$; components that are highly connected and provide inputs for multiple other components generally play a more central role in system functional-

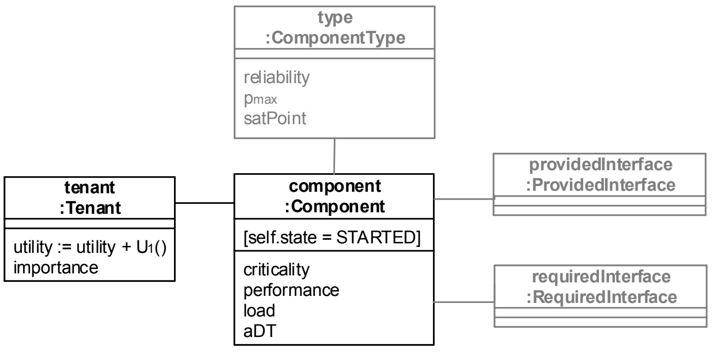


Figure 3.8: Positive pattern P_1^+ and relevant context (in gray) for saturating utility function calculation.

ity; thus components with higher number of providedInterface should contribute relatively larger values to the overall system utility compared to the components with fewer providedInterface. We capture this effect on the system utility by defining β in Equation 3.12 such that components with higher number of providedInterface obtain a larger weight in the utility function.

$$\beta = \begin{cases} 2 \times \mathsf{providedInterface}, & 2 \leqslant \mathsf{providedInterface} \\ 1, & \mathsf{providedInterface} < 2 \end{cases} \tag{3.12}$$

Moreover, the context required by the discontinuous utility function is also broadened compared to the linear variant. While the linear utility function is concerned with attributes of the Component and ComponentTypes, the discontinuous variant extends its computation context to also include the attributes of the Tenant via considering the importance of a tenant in the formula. In mRUBiS, different tenants have different importance based on their contract types; the importance of a Tenant represents the priority of the tenant for the business objectives of mRUBiS; tenants with higher importance contribute more to the overall system utility in terms of the business revenue. For instance, a *gold* costumer has a more expensive contract with mRUBiS and, consequently, benefits from certain priority maintenance and promotion services that *silver* and *bronze* costumers are excluded.

Finally, the discontinuous utility function includes a quality attribute *Average Deployment Time* (aDT) of a component, the attribute introduces the notion of cost to the utility function by adding the time variable. The aDT of a component is measured directly and stored as a component attribute in the architectural RTM—see the mRUBiS metamodel in Figure 2.8.

Combined U**:** For this variant, the previously introduced features, i.e., saturating and discontinuity effects, are aggregated in one function titled combined utility function in Table 3.2. Compared to the alternative variants, the combined function requires the largest context for utility calculation; the function contains the tenant-level quality attribute importance, the quality attribute performance, as well as the cost-related quality attribute aDT.

In this section, we presented four alternatives, belonging to different classes of mathematical complexity, for analytically constructed utility functions for mRUBiS. The utility functions defined in Table 3.2 are employed to compute the utility of a component in the mRUBiS architecture. As discussed in Section 3.1.1, the utility of a tenant in mRUBiS conforms to the additive utility formalism and is the sum of the utility of its constituting components. Similarly, the overall utility of the architecture is the sum of the utility of all the tenants that are hosted by the mRUBiS marketplace—see Equation 3.2.

3.3 LEARNING UTILITY-CHANGE PREDICTION MODELS

Non-linearities, complex dynamic architectures, and black-box models, are few examples of the sources for uncertainty and subjectivity that require specialized domain knowledge and render utility elicitation challenging [333, 383]. The absence of the domain knowledge required for constructing a representative utility function that captures system business objective is another reason that challenges the manual engineering of utility functions for software systems [51]. For complex, highly configurable systems or systems with a black-box model, it is often not trivial to obtain the relevant values for $\bar{c}$, $\bar{q}$, or $\bar{o}$ to analytically engineer a well-behaving $U(\bar{c}, \bar{q}, \bar{o})$.

In Section 3.2, we employed mathematical models to construct analytically derived models for $\hat{U}(\bar{c}, \bar{o})$ and showed how we calculate the utility-change function during $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$—see Equation 3.8. In this section, we are concerned with finding a proper approximation for the analytical utility-change function $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$. The approximation is captured as $\hat{U}^*_\Delta(\bar{c}, \bar{o}, a)$, henceforth, referred to as *predicted* utility.

A common practice in constructing the system utility is via hand-picking the features of the architecture [232] or reducing the number of the features by exploring the *inter-feature* relationships [129]. In this section, we side-step this problem by optimistically accepting the selection that is automatically made by certain types of machine learning algorithms, for instance, decision trees [55] and boosting methods [86, 153]. Additionally, similarly to Section 3.2.2, we explore the locality of the application context for action a; instead of $\hat{U}^*_\Delta(\bar{c}, \bar{o}, a)$, we employ a simpler function $\hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$. We consider only the sub-vectors for the architectural configuration $\tilde{\bar{c}}$ and observations $\tilde{\bar{o}}$, i.e., $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$ respectively, which are selected locally and relative to the context required by the adaptation action a.

In this section, we present a methodology, based on the supervised learning methods, to approximate the utility-change values during the adaptation process. The goal of the methodology is to find a proper approximation for the analytical utility-change function $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$ by means of a prediction model $\hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$. The outputs, i.e., utility-changes, correspond to the feedback signal for the learning techniques [361]. This is important because it makes the format of the collected data conducive to be used by a supervised machine learning method. In the following, we first briefly present the traditional steps of supervised learning followed by our proposed methodology that modifies and extends the conventional steps of the supervised learning process to train prediction models.

3.3.1 *Supervised Machine Learning*

In a supervised machine learning approach [101], a prediction model can be generally obtained by observing an output y and features $x_1, \dots, x_n \in X$ and assuming a functional relationship $y = f(x_1, \dots, x_n) + \epsilon$ where ϵ represents noise—see Section 2.2.1.2 for more details on supervised learning. The function f is then estimated using a *pre-*

diction model f^* such that $y^* = f^*(x_1, \dots, x_n)$ approximates the functional relationship with a *prediction error* [see 288] denoted as Equation 3.13[5].

$$|y + \epsilon - y^*| = |f(x_1, \dots, x_n) - y^*| = |f - f^*| \tag{3.13}$$

In cases where y is a numerical value, the learning problem is called *regression* [56]. Regression analysis in statistical modeling includes a set of statistical processes for estimating the relationships between a dependent variable, i.e., output y, and one or more independent variables, i.e., features $x_i \in X$. Solving a regression problem in general entails finding a conditional expectation or average value of y, because the probability of finding the exact real-valued number for y is zero [150]. Building a prediction model f^* based on the supervised learning technique involves iterations across a set of standard steps that we explain next:

1. Preparing includes three sub-steps; (1.1) generating sample data in the form of input-output (x_i, y_i) where each y_j is generated by an unknown function $y = f(x)$ for training and validation purposes (*data generation*). (1.2) choosing and setup of the machine learning algorithm and tuning the hyper parameters of the learning algorithm (*method choice and tuning*) [105, 372]. Method selection in general depends on the amount of the training data, the required accuracy of the output, the required speed or training time (that is inversely proportional to accuracy), and, finally, the linearity and the number of the features in the training data [435]. The hyperparameters of the learning algorithms are the high-level attributes that are set by the experts during the preparation step and before the model is assembled and trained; while many attributes can be learned from the training data, the learning algorithms cannot learn their own hyperparameters. The third step of the preparation is (1.3) selecting a subset of system features, $x_1, \dots, x_n$, to consider during training (*feature selection*) [2, 15, 196].
2. Training runs the machine learning algorithm of choice that builds the model f^*. This step, given a training set of N example input–output pairs $(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$, looks for a function f^* that approximates the true function f. Training f^* is optimized for a training error [276]. The training error here is obtained by testing the prediction model against data not seen during training[6]. Splitting the training data into two or more sets provides for training and then testing the model—during the training step—using a single source of data. This allows to detect if the model is *overfitting*, meaning that it performs well with the training data but poorly with the test data.

A common way of splitting the training data is using statistical model validation techniques such as K-fold cross-validation [see 5] where the data is split into K sets, allowing to train and test the data K times. During a K-fold cross-validation, the data is split into K equal parts (folds) where one fold is designated as the *hold-out fold*. The prediction model then is trained on the remaining $K - 1$ folds and validated on the hold-out fold. The process is repeated for K times where each time a different fold is chosen to be the hold-out fold; the average performance across the ten hold-out folds is the performance estimate, i.e., the *cross-validation score*.
3. Validating happens iteratively with steps 1 an 2 for each prediction model f^*. The goal is to *check prediction errors*. To measure the accuracy of a prediction model, we give it a validation set of samples. Validation mitigates over-fitting by using datasets that

5 The *real error* (without noise) would be $|y - y^*| = |f + \epsilon - f^*|$.
6 Note that the testing is an internal sub-step *within* the training step and is distinguished from the validation step that follows the training.

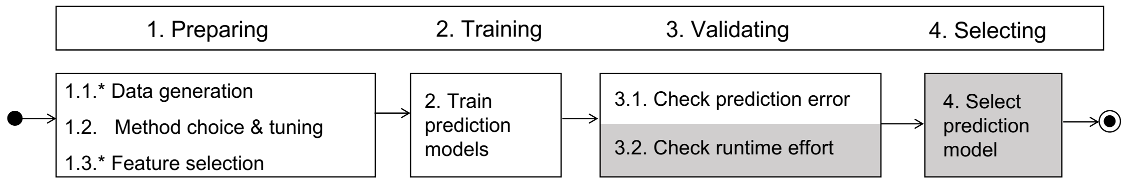

1. Preparing
2. Training
3. Validating
4. Selecting
1.1.* Data generation
1.2. Method choice & tuning
1.3.* Feature selection
2. Train prediction models
3.1. Check prediction error
3.2. Check runtime effort
4. Select prediction model

Figure 3.9: Steps of proposed methodology to train prediction models.

are distinct from training, e.g., 70% for training and 30% for validation. A prediction model is recognized to *generalize* well if it correctly predicts the value of y for novel examples. If the validation indicates that the prediction model is not yet appropriate, we can obtain more data, choose new methods/hyperparameters, or change the feature selection. At some point, we deem the prediction model acceptable regarding error and model size/complexity; since this stopping criterion is subjective, one usually needs to explore multiple prediction models to be confident in the outcome.

3.3.2 *Modifications to Standard Supervised Learning Steps*

We address the learning problem with a four-step methodology, presented in Figure 3.9, that extends (shown as gray) and modifies (marked with *) the standard three-step machine learning process in Section 3.3.1. We briefly discuss our proposed extensions and modifications to the standard machine learning process below:
1. Preparing We modify the *data generation* (1.1*) step to obtain unbiased and representative sampling. During this step, we have to mitigate specific realizations of the running system, e.g., the adaptable software, that can inject bias in the data. The *feature selection* (1.3*) step is challenging for large, dynamic software architectures, because the number of the configuration and the observation attributes can grow arbitrarily and cause state space explosion [290]. Since features can be composed with various arithmetic operators, analytically discovering the set of relevant features involves a search in the combinatorial space that can grow fast, even for small architectures. We benefit from the specific case of the architecture-based adaptation that allows for modifying this step based on the application context of the adaptation actions and reducing the search space for the relevant features.
2. Training No adjustments needed in our methodology with respect to the standard training phase of the supervised learning process.
3. Validating In addition to the typical *check prediction errors* (3.1) discussed in Section 3.3.1, we modify the validating step to *check runtime effort* (3.2*) of the prediction models. The reason is that, similarly to the prediction error of the model, the runtime effort of the prediction model also affects the performance of the system. Since we can use the data generated in step 1.1*, we can measure the runtime effort without having to execute the system for this propose.

4. Selecting We propose to train multiple prediction models, employing different learners in parallel, thus the extension includes adding a dedicated fourth step to the steps introduced in Section 3.3.1 for *selecting* the best fitting prediction model. In the case that we have learned several acceptable prediction models, we need to choose one to be deployed on the system; model selection, however, is known to be a difficult task [276]. Furthermore, while relying on the prediction error guarantees the best decision based on the predictions, in the context of using the (predicted) utility values to plan for adaptations, the concern is the ordering and selection of the adaptation actions with highest utility-change, rather than the accuracy of the prediction, i.e., the absolute value of the changes. Therefore, we have to identify a proper selection procedure that takes this into account.

Finally, applying the methodology determines one prediction model that approximates the optimal utility-change; the prediction model is then exported in a runtime executable format and replaces the analytic Utility Function in Figure 3.1 that may plug in to the system.

3.3.3 *Methodology*

The goal of the proposed methodology is to train a prediction model $\hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$ that properly approximates $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$. Consequently, $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$ is the output and $\bar{c}, \bar{o}, a$ are the features we observe for the functional relationship that has to be estimated. As we can measure (calculate) $\bar{q}, \bar{c}, \bar{o}$, and a by executing the system, we can also indirectly observe $\hat{U}_\Delta(\bar{c}, \bar{o}, a) := \hat{U}(\bar{c}', \bar{o}) - \hat{U}(\bar{c}, \bar{o})$ as defined in Equation 3.8. Next, we detail our methodology, shown in Figure 3.9, to train prediction models for utility-changes.

3.3.3.1 *Preparing: data generation*

In the conventional supervised learning process, data generation includes sampling the system execution in the form of input-output pairs [101]. However, in order to sample a self-adaptive system, we require the sampled data to be fully representative of the software configuration space together with the *adaptation space*. The adaptation space includes also mappings of the adaptation actions to system configurations. Consequently, in order to have a thorough coverage of the system configuration space, we suggest in our methodology to execute the system excluding realizations of the system that introduce bias to the sampled data.

During the data generation, we execute the adaptable system together with a randomly operating adaptation engine on top—see Figure 2.6 for a reference model of a self-adaptive system. This mitigates bias in the data that can be introduced by a specific realization of the adaptation engine. A random adaptation engine maps the configuration and observation attributes to the adaptation actions in a random manner. It provides a thorough coverage of the configuration-action combinations, i.e., adaptation space. Conversely, any other *rational* adaptation steering policy could introduce bias by leaving out certain combinations of configuration, observations, and adaptation actions.

3.3.3.2 *Preparing: method choice and tuning*

In this step of the methodology, we choose the learning algorithm and tune its hyperparameters accordingly to fit the problem at hand, i.e., learning the prediction model for utility-changes in large and dynamic architectures. As discussed in Section 3.3.1, learning a prediction model $\hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$ that approximates $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$ with numerical values is a regression problem. In line with best practices in machine learning [see 74], we suggest to use approximation methods based on decision trees [349].

A decision tree represents a functional relationship that takes as input a vector of attribute values X, runs a set of tests on the value of each input attribute $x_i \in X$, and reaches a *decision*, i.e., a single output value y. A decision tree partitions the space of input variables into homogenous rectangle areas by a tree-based rule system; each tree split corresponds to an If-Then rule over the input attributes.

Each internal node in the tree corresponds to a test, i.e., an If-Then rule, of the value of one x_i and the branches from the node are labeled with the result of the test, i.e., the possible values of the attribute. Each leaf node in the tree specifies a value to be returned by the function. The *decision tree learning* algorithm adopts a greedy divide-and-conquer strategy[7] where the *most discriminative attribute* is tested first. The most discriminative attribute is the one that makes the most difference to the classification of the provided sample set and consequently, obtaining the correct values is feasible with a small number of tests, meaning that all paths in the tree will be relatively short and the tree as a whole will be shallow. The greedy search used in decision tree learning is designed to approximately minimize the depth of the final tree [349, 361]. Decision trees are parameterized with the number of splits, or equivalently, the interaction depth.

Section 3.3.1 lists a set of criteria affecting the choice of the learning methods. Accordingly, the motivation to employ decision-tree learning algorithms in our methodology is twofold; (i) these methods automatically perform feature selection and (ii) are capable of handling non-linearity in data and modeling non-linear functions. Therefore, contrary to the linear regression methods, we do not have to hypothesize which features are used in the utility functions and how they are combined into an analytic formula.

Decision trees in general are known for having low bias and high variance.Variance is the amount that the estimate of the prediction model will change given different training data. Low variance suggests that changes to the training dataset lead to small changes to the estimate of the prediction models. Bias consists of high variability in the predictions across training sessions and might be a consequence of low signal-to-noise relationship or data that is not representative of all functional relationships that we need to learn. A low bias model incorporates fewer assumptions about the target function. Consequently, as a result of exhibiting low bias and high variance, decision trees pose a major challenge that is higher risks of overfitting compared to the alternative learning methods. Overfitting happens when the prediction model presents low error rate for the training data, but high error rate with data that the model has not seen, i.e., validation data. This is a consequence of overly complex models, e.g., deep decision trees [133, 220], which often results in factoring noise into the solution.

We mitigated the overfitting of the decision trees in two ways: (i) employing *ensemble machine learning methods* that is building a collection, or an ensemble of learners, as apposed to employing a single strong predictive model for data-driven modeling tasks and combining their predictions. The ensemble approach relies on combining a large number of relatively weak learners [8] to obtain a stronger ensemble prediction. The most prominent examples of such machine-learning ensemble techniques are random forests [55] and neural network ensembles [201], which have found many successful applications in different application domains [130, 291, 347, 382]. The ensemble models are a useful practical tool for different predictive tasks, as they can consistently provide higher accuracy results compared to the conventional, single, strong machine learning models. (ii) loss function optimization over hyperparameters [369]. Hyperpa-

7 The divide-and-conquer test recursively divides the problem into smaller sub-problems until they become simple enough to be solved directly—see [99].

8 A weak learner is a learning algorithm that always returns a prediction model with accuracy on the training set that is slightly better than random predictions [361].

rameter optimization is the process of tuning the high-level attributes of the learning algorithm (see Section 3.3.1), that finds a tuple of hyperparameters that yields an optimal prediction model minimizing a loss function on given independent data. The objective function takes as input a tuple of hyperparameters and returns the associated loss [202].

As discussed in Section 3.3.1, while the selection of the algorithms is primarily determined by the end use-case, there are a set of additional factors that steer the choice of the learning algorithms; the influencing factors include: algorithm-model complexity, performance, interpretability, computer resource requirements, and speed. To this end, we adopted three different learning methods to train multiple prediction models in parallel. The methods are extensively used by the practitioners in the machine learning community and fit the intended use case and include: *Random Forest* (RF) [55], *Gradient Boosting Models* (GBM) [153], and *Extreme Gradient Boosting Trees* (XGB) [86]. Next, we briefly introduce each method and our motivation for choosing them.

RF is a classification algorithm consisting of many tree predictors, i.e., decision trees, such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. It uses *bagging* and feature randomness when building each individual tree to try to create an uncorrelated forest of trees whose collective prediction is more accurate than that of any individual tree. The generalization error for forests converges to a limit as the number of trees in the forest becomes large—see Section 2.2.1.2.

Bagging [see 56] is the process of generating additional data for training from the original data set using random combinations with repetitions to produce multi-sets of the original data. It is an ensemble algorithm that fits multiple models on the generated multi-sets of the training dataset, then combines the predictions from all models. RF is an extension of bagging that also randomly selects subsets of features used in each data sample. Bagging results in decreasing the variance in the prediction. Decision trees in general are known for introducing high variance and since RF builds on decision trees, it mitigates the high variance issue via its bagging process. Variance is the amount that the estimate of the prediction model will change given different training data [240]. Low variance suggests that changes to the training dataset lead to small changes to the estimate of the prediction models. RF aim to reduce the complexity of prediction models that overfit the training data via bagging.

GBM has a learning procedure that consecutively fits new models to provide a more accurate estimate of the output. The method in general includes three main elements: a loss function to be optimized, a weak learner to make predictions, and an additive model to add weak learners to minimize the loss function. The loss functions captures the prediction error of the ensemble and can be any arbitrary error measure. As weak learners, GBM employs decision trees. More specifically, regression trees that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and correct the residuals in the predictions. This process in machine learning techniques is recognized as *boosting*. Trees are added one at a time, and existing trees in the model are not changed. The algorithm constructs the new learners to be maximally correlated with the negative gradient of the loss function [153, 235].

The boosting process in GBM, similarly to the bagging process in RF, is an ensemble method and builds several learners from one learner by generating several multi-sets of the original training dataset via random sampling. In contrast to bagging, boosting

sequentially determines weights for the sampled data, i.e., *weighted training set*. In a weighted training set, each data point has an associated weight $w_j \geqslant 0$; higher weights indicate higher importance of the sample during training and, as a result, the data point is more likely to get resampled and appear more often in several training sets. The motivation behind this is that instances, which are hard to predict correctly, will be focused on during learning, so that the model learns from past mistakes. Boosting reduces the risk of overfitting to the training data in prediction models by decreasing the variance of single estimates as it combine several estimates from different learners. Thus, the result is often a model with higher stability [234].
XGB is a specific implementation of the Gradient Boosting method which uses more accurate approximations to find the best tree model. It improves the baseline gradient boosting methods to be highly scalable, flexible and portable. XGB extends the GBM by using the second order derivative as an approximation. It computes the second-order gradients, i.e., second partial derivatives of the loss function, which provides more information about the direction of the gradients and how to minimize the loss function; regular GBM, however, uses the loss function of the base model, e.g., decision tree, as a proxy for minimizing the loss function associated with the collective. Moreover, XGB has advanced *regularization* which improves model generalization and results in scalable and fast training that can be parallelized across the clusters. Regularization is a technique to improve the generalization of a learned model by adding information in order to prevent overfitting [317].

3.3.3.3 *Preparing: feature selection*

Feature selection is the last activity in the preparing step of the methodology—see Figure 3.9 for an overview of the steps. The goal of feature selection in machine learning is to find the best set of features that support building prediction models of the studied phenomenon. The common challenge affecting the feature selection step arises in the context of large, complex systems with large state space in terms of dimension and/or number of instances [134], where conventional feature selection methods, e.g., search-based heuristics [see 87, 344] can be computationally prohibitive.

In this thesis, we are concerned with dynamic software architectures where the number of configuration and observation attributes can grow arbitrarily such that the combinations of $\bar{c}$ and $\bar{o}$ are not bounded. However, for the considered case, the learning problem is only to train a prediction model $\hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$ that properly approximates $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$ for a specific adaptation action a and its affected application context. In our methodology, by only considering relatively small architectural fragments, i.e., $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$, and the relevant application context of the adaptation action a, the learning problem can be limited to a smaller fixed number of features. Consequently, standard machine learning methods become applicable.

The sub-vectors $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$, as the selected features to learn $\hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$, are determined relative to the application context for action a—we will discuss this in more details in Section 4.2. Therefore, by systematically extending the context of an action a, such that more features become available, we can generate enough data to train a prediction model for large and unknown dynamic architectures. For example, consider an adaptation action that modifies a component state in mRUBiS; the application context of the action is the `state` attribute of the corresponding `Component` in the architecture; we can systematically navigate the architecture and extend the context to include more features, e.g., attributes of the Tenant containing the Component or of the ComponentType—see Figure 2.8 for a reference of mRUBiS metamodel.

In the proposed methodology, we only consider how to extend the offered set of features with additional candidates; the reason is that we will keep only the prediction models that themselves select a suitable subset of the offered features—see Section 3.3.3.2; we do this by relying on the training step of the methodology to reduce the features to the most significant ones.

Note that a constructed analytic function $\hat{U}_{\Delta}(\bar{c}, \bar{o}, a)$ will usually only consider a small fragment of the architectural configuration $\bar{c}$ and observations $\bar{o}$ denoted by $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$, which contain only a few dimensions of the original vectors—see Section 3.2.2. Therefore, we assume that a local, hence small, fraction of the configuration and observation attributes correspond to a feature space that is sufficient to learn the utility-changes. Moreover, we aim to predict only the impact of each change, i.e., adaptation action application for a given context, rather than learning the complete model S, which would not be feasible. Model S predicts the quality attribute values $\bar{q}$ for a planned configuration $\bar{c}$ and observations $\bar{o}$—see Equation 3.5. However, this would require a thorough coverage of all combinations of $\bar{c}$ and $\bar{o}$ and can lead to combinatorial explosion regarding the size of the architecture, combinations of $\bar{c}$ and $\bar{o}$ attributes, and their domain sizes. Consequently, standard machine learning approaches that require a fixed number of features are not directly applicable due to this effect.

3.3.3.4 *Training*

As discussed earlier, this step of the methodology conforms to the conventional model training process in supervised learning—see Section 3.3.1, thus no novel contributions are introduced. Figure 3.10 shows a flowchart of the activities and data objects during the Training step of the methodology. Following the *method choice and tuning*, this step initially is concerned with finding the most proper split of the sampled data generated during the *data generation* step. The generated data is split to distinct *training set* and validating set, e.g., 70% training and 30% validating. Next, the training set is again split using statistical K-fold cross-validation technique–see Section 3.3.1. The training data, i.e., in the form of input-output pairs, is then fed to the employed model training methods. The training step is a repeatable process that maximizes utilization of the available training data. Following the K-fold cross-validation for splitting the data, we loop through K iterations of train and test.

At the end of the K^{th} iteration, the average model performance, i.e., the prediction error of the K trained models during the K tests, is evaluated. In cases where the trained model exhibits low cross-validation score, the training is re-initiated after adjustments of the hyperparameters of the employed learning algorithm. During the training, we optimized for model accuracy via minimizing the prediction error. We use *Root Mean Square Error* (RMSE) as the performance metric of the prediction models that is the square root of the mean of the squared differences between actual outcomes and predictions—see Equation 3.14. The stopping criteria for the hyperparameter tuning during training may be defined based on the number of iterations, upon obtaining certain model performance, or when the prediction performance saturates to a stable value. Once the stopping threshold is met, among different prediction models, we chose the one with the smallest RMSE, i.e., the highest performing model. Next, we switch to the model validation step based on the unseen validating data set.

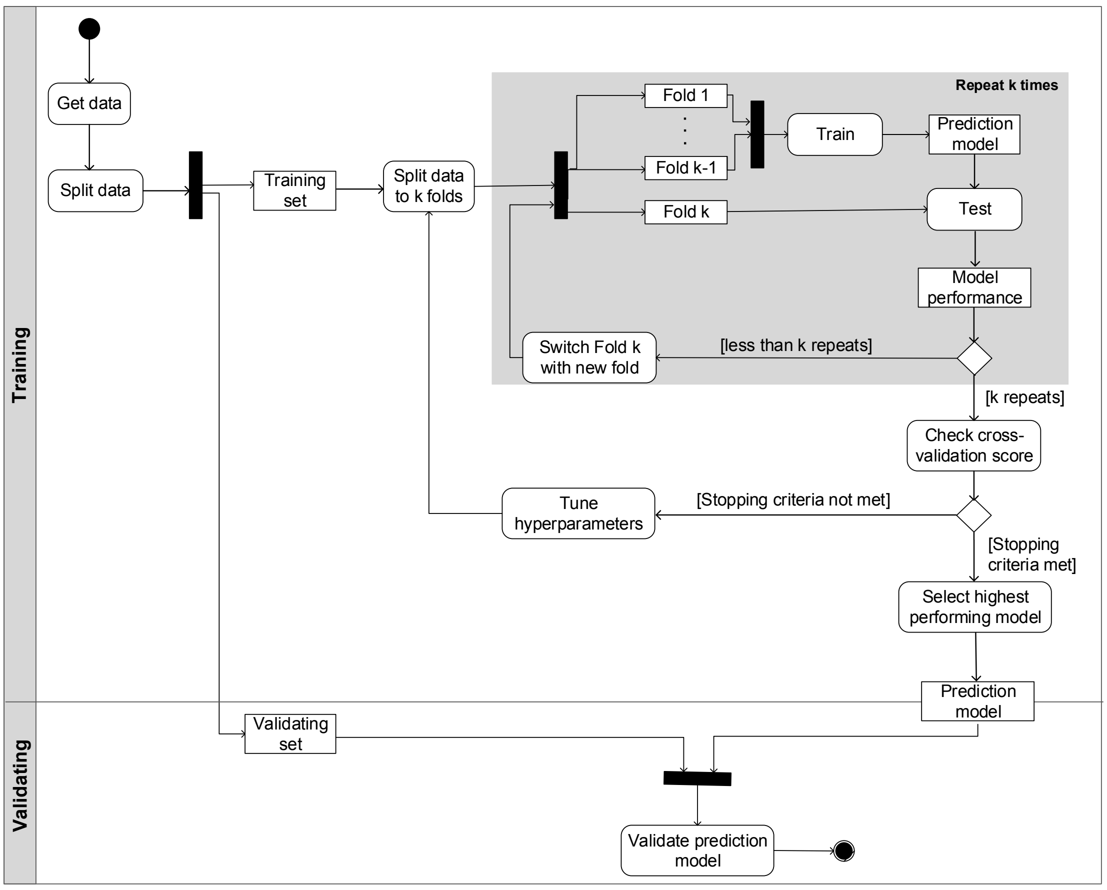


Figure 3.10: Overview of model training.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{i=N}(x_i - \hat{x}_i)^2}{N}} \tag{3.14}$$

3.3.3.5 *Validating: check prediction error*

During the validating step, we evaluate the trained prediction models across the validating data set. For this purpose, we utilized the *Mean Absolute Deviation Percent* (MADP) metric which gives normalized values between zero and 100% and is computed according to Equation 3.15. Different data splits between the training and validating sets may yield different MADP values.

$$\text{MADP} = 100 \times |\frac{\text{Actual value} - \text{Predicted value}}{\text{Actual value}}| \tag{3.15}$$

In the context of learning prediction models for utility-changes, the error between the actual and predicted utility-change values, defined as Equation 3.16, is normalized to serve as MADP for the prediction models.

$$e^*_\Delta(\bar{c}, \bar{o}, a) := |\hat{U}_\Delta(\bar{c}, \bar{o}, a) - \hat{U}^*_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, a)| \tag{3.16}$$

Note that *Actual value* in Equation 3.15 and $\hat{U}_{\Delta}(\bar{c}, \bar{o}, a)$ in Equation 3.16 represent the *ground truth* for the target variable and are obtained from the output values in the validating data set that comes in the form of input-output.

For the employed set of learning methods, we opt for minimizing MADP by allowing the decision trees to select all the available input features. This is accomplished by tuning various hyperparameters of the employed methods such as number of the leaf nodes and depth of the trees.

3.3.3.6 *Validating: check runtime effort*

In addition to the standard machine learning validation step, i.e., checking prediction errors, in the proposed methodology we suggest an additional step, i.e., *check runtime effort* in Figure 3.9, to validate the K prediction models $\hat{u}^*_{\Delta,1}(\tilde{\bar{c}}, \tilde{\bar{o}}, a), \dots, \hat{u}^*_{\Delta,k}(\tilde{\bar{c}}, \tilde{\bar{o}}, a)$. Runtime effort is the amount of time required to load and execute the prediction models on the running system until the prediction is made available by the models. However, we can evaluate the runtime effort of the models by running them on the already available execution traces of the system that is the data generated for learning.

At the end of this step, based on the observed runtime effort and the characteristics of the system, a prediction model might be discarded or further tuned to prevent performance reduction at runtime. The validation step is concluded by choosing best performing prediction models regarding the model accuracy, i.e., prediction errors and their runtime effort.

3.3.3.7 *Selecting*

Selecting models based on the prediction error does not guarantee that employing them on a running system yields the highest reward, i.e., accumulated utility over time. In other words, minimizing the prediction error does not necessarily maximize the system performance. The reason is twofold; the prediction error values are averages across predictions, hence, the error is not necessarily uniform across all the architecture elements. The second reason is that in the context of self-adaptive software system, system performance depends on how the adaptation process is carried out.

The decision-making process during adaptation follows a three step process: (1) for a current system state, represented by $(\bar{c}, \bar{o})$, estimate the utility-change $\hat{U}^*_{\Delta}(\bar{c}, \bar{o}, a)$ during $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$. (2) select an adaptation action a based on the estimates, and in case a sequence of adaptation actions are selected for execution, (3) rank the actions regarding their priority for execution. We discuss the decision-making process, and more specifically, steps (2) and (3) in Chapter 4. In this section, the proposed methodology is concerned with step (1), i.e., predicting the utility-changes. However, it is natural that the predictions further influence step (2) and (3) of the process. Thus, in the following, we discuss the fourth step of our methodology, i.e., *selecting* the best-fitting prediction models that, in addition to the prediction error and runtime effort, considers model performance in finding the *optimum* ranking of the adaptation actions during step (3). At this part of the text, let us assume that the optimum ranking list of adaptation actions for sequential execution is a list of actions that are ranked in a decreasing order of the estimated utility-change. Therefore, actions resulting in higher utility-change in their corresponding adaptation $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$ are prioritized. We revisit and provide more context for this assumption later in Chapter 4.

For each sorted list of adaptation actions, the adaptations with higher estimates of utility-change should be on the top of the list. Nonetheless, even small prediction errors can make the order of the actions diverge from the optimum. Our solution is to directly compare the decision lists produced by the prediction models (predicted lists) to optimal lists. The predicted lists can be obtained by executing the prediction models on the real system. For each predicted list, it is also possible to *observe* the real utility-change, i.e., the ground truth, via measurable quality attributes. Adaptation actions are then manually sorted in a descending manner regarding the utility-changes. This generates an optimal list for each predicted list which can now be compared. The lists are then compared regarding differences, i.e., mismatches. For that, we extend a set of standard similarity metrics to cover three families of mismatches: the number of items with different ranking positions (counting), the magnitude of the ranking differences (distance), and the location of these differences (position), which we explain next.

Count-based metric. The Jaccard similarity index [229], with a range from 0 to 1, is a measure of similarity for the two sets of data. It compares members for two sets to identify their shared as well as the distinct members and is defined as the intersection of two sets divided by the union of the two—see Equation 3.17. In our methodology, we adopted the Jaccard coefficient to count the number of mismatches, i.e., items with different positions in the ranking, between the predicted list and the optimal list. We are concerned with comparing *lists* which are *ordered* sets. Moreover, the two lists are only different with respect to their rankings. This means that the predicted list is fully contained in the optimal list. Therefore, in calculating the Jaccard distance metric for two lists, the intersection of the lists is the number of the items in similar positions while the union is the length of the lists.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.17}$$

Distance-based metric. Counting the mismatches between two sets might hide the fact that mismatches can be distinct regarding the distance between the actual position in the ranking and the predicted one. Distances can be seen as the magnitude of the error for each mismatch. Items with larger mismatch distances then have a larger negative impact on a distance-based similarity index. This is important when certain list items should present smaller mismatch distances than others. In the context of this thesis, where lists include a sequence of adaptation actions, certain actions that are more critical to system functionality should be prioritized over others, thus it is desirable to minimize their mismatch distance in the predicted list; a distance-based similarity metric allows for prioritizing these items in the list.

Kendall-tau, also written as Kendall τ [250], is a non-parametric correlation metric that computes distances among the mismatches. Kendall-tau is a measure of rank correlation indicating the similarity of the orderings and is calculated according to Equation 3.18. The variable N is the length of the lists, c is the number of *concordant pairs* and d is the number of *discordant pairs*. For lists A and B, let $A[i]$ and $B[i]$ represent the items in position i of the lists; any pair $(A[i], B[i])$ and $(A[j], B[j])$ where $i < j$, are concordant if the sort order of $(A[i], A[j])$ and $(B[i], B[j])$ agrees; that is, if either both $A[i] > A[j]$

and $B[i] > B[j]$ holds or both $A[i] < A[j]$ and $B[i] < B[j]$; otherwise, they are considered discordant [95].

$$\tau = \frac{c - d}{\binom{N}{2}} \tag{3.18}$$

The numerator $c - d$ in Equation 3.18 is calculated according to ALGORITHM 3.1.

ALGORITHM 3.1 : $c - d$

```
numer ← 0
for i = 2 to N do
    for j=1 to i-1 do
        numer := numer + sign(A[i] − A[j]) × sign(B[i] − B[j])
    end
end
return numer
```

Position-based metric. Mismatches can also be distinct regarding the position that they happened in the ranking. Take for instance an item that is ranked at second position in a list, but should have been ranked first. Now take the case of an item ranked 46th, but should have been ranked 45th. Note that in both cases, there are two mismatches, hence, same counting and the mismatch distance is the same, i.e., exactly one. However, the mismatch at the top of the list might be more relevant. The premise of position-based metrics is that highly relevant items appearing lower in a list should be penalized. Among several metrics from the information retrieval field [82], we employ the *Discounted Cumulative Gain* (DCG) metric [233].

DCG uses a *graded relevance* scale of ordered items, e.g., order of the results in a web search. The graded relevance scale measures the usefulness or gain of an item based on its position in the list. The gain is accumulated from the top of the list to the bottom, with the gain of each item discounted at lower ranks. DCG penalizes wrong positioning of items that should be in higher ranked positions via logarithmically reducing the graded relevance value proportional to the position of the item. Equation 3.19 characterizes the DCG metric for an item in position p of a list where rel_i is the graded relevance of the item at position i that is defined by the user or a domain expert.

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} \tag{3.19}$$

In our methodology, we extend the metric to consider the position of the mismatch and to apply the discount factor only to the item that caused a mismatch. Moreover, we obtain rel_i values for items in the predicted list from the ground truth, i.e., the optimal list. The rel_i for an item in position i of the predicted list is its position in the optimal list.

Aggregation of metrics. Each of the three similarity metrics introduced above might reflect an architectural concern, which a designer of an adaptive system would need to combine in different ways. At this step of our methodology, we enable this by aggregating all three metrics into one single metric, i.e., *Similarity Aggregation Metric* (SAM). SAM, defined in Equation 3.20, is the weighted average of the three metrics. It is normalized between 0 and 1 and allows for determining distinct levels of importance per metric and assigns the weight w_i to the value accordingly. In order to compute SAM, the Jaccard, Kendall-τ, and DCG metrics are measured for the predictions made by each prediction model mod. Finally, the output of this step of the methodology is the model yielding the highest similarity of the predictions to the ground truth, indicated by the SAM metric.

$$\mathsf{SAM}(mod) = \frac{w_1 \times Jaccard(mod) + w_2 \times Kendall(mod) + w_3 \times DCG(mod)}{w_1 + w_2 + w_3} \tag{3.20}$$

3.4 SUMMERY

The objective of this thesis is to address the quality and cost trade-off during software self-adaptation. As a first step towards achieving this goal, in this chapter, we discussed our approach to define utility functions for dynamic software architectures that enable incremental evaluation of the architecture. We introduced the notion of pattern-based utility and described how utility values are assigned to fragments of the software architecture. The pattern-based definition of utility supports dynamic architectures that may change in size or complexity at runtime, for the context of the pattern-matching can also be dynamically extended. Assuming a restricted scope of change in the architecture caused by each action execution, we proposed incremental utility calculation, i.e., via using the locality information. Instead of evaluating the utility of the complete architecture, the changes of the utility, caused by the action, are incrementality computed and maintained. This way, we avoid the costly process of observing and evaluating the complete sate space and only track the changes of the architecture and of the utility. We referred to this concept as the utility of the configuration-changes or simply, utility-change. We proposed to consider utility-change calculation (prediction) rather than evaluating the utility of the complete architecture, thus supporting incremental utility calculation (prediction) for large and dynamic architectures.

In the context of dynamic architectures, we showed (i) how to analytically construct utility functions from domain knowledge, and (ii) how to train prediction models for utility-changes from observations. We demonstrated the analytical utility construction based on business objectives for mRUBiS application example as the process requires system-specific goal models. Further more, this chapter addressed the problem of initially unknown runtime knowledge of user and system preferences by proposing a methodology that modifies and extends the conventional supervised learning process to train prediction models for utility-changes—see $\mathcal{R}6$ in Table 1.1. This chapter provides the *Utility Function* building block of our generic solution for architecture-based self-adaptation in this thesis—see Figure 1.2 for an overview. In the following chapters, we discuss how the adaptation engine leverages the utility values provided by the *Utility Function* introduced in this chapter.

4 VENUS: UTILITY-DRIVEN RULE-BASED SCHEME FOR ARCHITECTURE-BASED SELF-ADAPTATION

This chapter presents Venus, a utility-driVen rule-based scheme for architecture-based self-adaptation. Being integrated in a MAPE-K feedback loop, Venus is realized in an external approach for self-adaptation where the adaptation engine dynamically observes and adjusts the adaptable software. The scheme relies on existence of a *Utility Function* and implements the *Analyze* and *Plan* activities of the adaptation engine—see Figure 4.1 for an overview. We showed in Chapter 3 how to construct the *Utility Function* building block; in this chapter, we discuss how Venus utilizes the Utility Function during the Analyze and Plan.

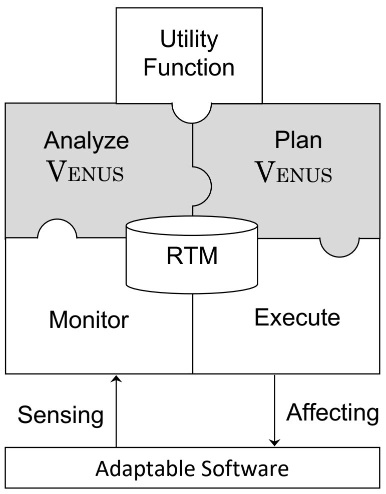


Figure 4.1: Chapter overview: analyze and plan with Venus in adaptation engine.

MDE principles, constituting one of the major underpinnings of this thesis' solution to software self-adaptation, support creation and runtime evolution of causally connected RTMs. Building on the MDE principles, Venus uses models of software architecture at runtime for maintaining the self-adaptive software. The fundamental requirement of Venus is that the adaptation engine uses RTMs of software architecture as introduced in Section 2.4.

While the state-of-the-art mainly subscribe to one of the main formalisms for the adaptation policies, i.e., ECA rules and optimization-based solutions (goal-based or utility-based), Venus tailors both classes, at design time, to assemble a compromise that collectively exhibits the benefits of two ends of the spectrum, while, inevitably, inheriting certain limitations of its constituent elements. Venus achieves this combination via a set of novel steps; leveraging graphs and graph-based formalism, Venus captures the architectural RTMs as graphs. Consequently, the runtime issues requiring adaptation are realized as *graph patterns* in the architectural RTMs. Venus employs ECA rules to capture the adaptation decisions. In addition, the scheme uses utility functions, introduced in Chapter 3, to capture system objectives and steer the adaptation towards maximizing the utility.

Similarly to the adaptation issues, Venus defines the adaptation rules as well as the system utility in the space of graph patterns; this supports mapping between the utility values and the adaptation rule applications. This way, Venus embodies ECA rule-like constructs and relies on utility theory to choose an adaptation. Moreover, defining adaptation rules and utility functions based on the patterns in system architecture supports adaptation of software systems with dynamic architectures that may change in size and complexity. The idea of graph pattern-based characterization of the adaptation issues, adaptation rules, and utility values is the underlying principle of Venus.

In this chapter, we first present the graph-based realization of the adaptation concepts applicable to Venus in Section 4.1. Next, following the discussion of pattern-based utility introduced in Chapter 3, we discuss the linking of utility values to adaptation rule applications in Section 4.2. Then, we present in Section 4.3 an overview of our solution for incremental execution of the adaptation engine whereby we discuss the integration of Venus in a feedback loop. Finally, we provide an assessment of the computational effort in Venus in Section 4.4 that is followed by a set of assumptions for applicability of Venus and validity of the claims.

4.1 GRAPH-BASED REALIZATION OF ADAPTATION CONCEPTS

Venus is realized in an external approach for self-adaptation (see Section 2.3.2) and is integrated in a MAPE-K feedback loop where the adaptation engine dynamically observes and adjusts the adaptable software—see the reference model for the feedback loop in Figure 2.6. The adaptation engine maintains a causally connected architectural RTM, introduced in Section 2.4, as part of its knowledge to represent the architecture of the adaptable software. Thus, adaptations of the software are achieved via adding, removing, and reconfiguring components as well as connectors—see [302]. The RTM enables capturing structural and behavioral dynamics of the causally connected system [47, 302]. Moreover, software architecture can be characterized as a graph of components and connectors [282, 322]. Thus, Venus captures the required modeling artifacts for architectural RTMs by considering a graph-based realization of the model that enables the use of graphs-based formalism and graph transformations [115, 415]—see Section 2.6.1.

The architectural RTM represents the system architecture as a graph of interacting components [322]. Nodes in the graph, i.e., components, represent the principal computational elements and data stores of the system; arcs, termed connectors, represent the pathways of interaction between the components. Although there are different views of architecture [96], in this thesis, we are primarily interested in the component-connector view as it characterizes the abstract state and behavior of the system at runtime to enable reasoning about problems and courses of adaptation—see example of an architectural RTM for mRUBiS in Figure 2.9.

By realizing the architectural RTM as a graph G and building on MDE principals, Venus enables realization of model transformation via the established graph operations, i.e., graph transformation [27, 186, 377] that is suitable for structural changes of models. Thus, Venus uses attributed graphs and graph transformation to capture the architectural RTM of the adaptable software and the structural changes of the models

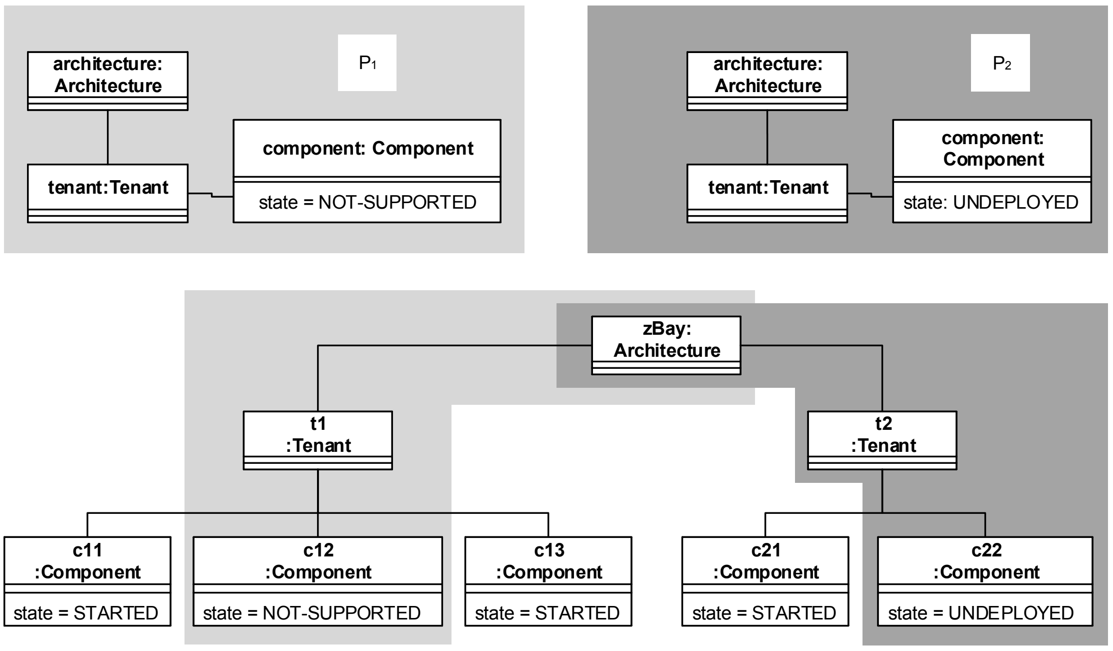


Figure 4.2: Examples of adaptation issues (top) and excerpt of mRUBiS architectural RTM with matches for issues (bottom).

respectively [186]—see Section 2.6.1. Particularly, we used *Story Diagrams* (SDs)/*Story Patterns* (SPs) [141] as the modeling language to realize the graph (model) transformation rules in Venus. They allow to specify patterns at the same level of abstraction as the RTM[1]. SDs are similar to ECA rules except that SDs are a visual language and based on graph transformations—see Appendix A for an introduction to SD formalism. Technically, we employ an SD interpreter, i.e., an execution engine for graph transformations; we will discuss this in Section 7.1 together with the implementation details of Venus.

Venus primarily employs a rule-based scheme that defines failures or performance issues that can be directly identified and localized as faults or bottlenecks in the architectural RTM. Moreover, it defines adaptation strategies that can address these issues. To specify a model query over the architectural RTM we use a pattern P. A pattern P of a set of patterns $\mathcal{P}$ represents a structural fragment of the architecture G. In Venus, we express the adaptation issues, henceforth, issues, in the architecture as model (graph) patterns such that concrete issues relate to occurrences of these patterns in G. Since the architecture is represented by the RTM, we also use G to refer to the model. An occurrence of a pattern P in G corresponds to a match m of P in G and is represented as $G \models_m P$—see Section 2.6.1. Figure 4.2 shows examples of patterns P_1 and P_2 capturing two adaptation issues in mRUBiS. P_1 indicates a runtime component failure where the state of a component in the architecture is set to NOT-SUPPORTED and P_2 represents the issue where a component is in UNDEPLOYED state. In the bottom, two matches, m_1 and m_2, in an excerpt of the architectural RTM of mRUBiS are marked. The RTM conforms to the metamodel of mRUBiS in Figure 2.8.

1 We used the *Story-driven Modeling* (SDM) tool as the model execution engine that comprise an editor, interpreter, and debugger for SDs. Besides SDs, other graph transformation languages have been proposed for expressing adaptation rules [390, 404, 424].

VENUS enables structural adaptation of the software—see Section 2.3.2—where software components, connectors, and their attributes define the points of variability in the configuration space for the adaptable software [35]. The configuration space of the adaptable software is defined by a vector of configuration settings $\bar{c}$ and thus the system is adapted via changing the current configuration $\bar{c}$ to the new configuration $\bar{c}'$. A pair of the configuration attributes and the corresponding observations $\bar{o}$, i.e., $(\bar{c}, \bar{o})$, constitute the state of the adaptable software system at each point in time—see Section 2.3.2. The architectural adaptation of the software system changes the current state of the system, captured by $(\bar{c}, \bar{o})$, to a new state $(\bar{c}', \bar{o})$ by execution of an *adaptation action* $a \in A$ with A the set of available actions to adapt the system. We show this as $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$.

VENUS captures the adaptation actions as ECA rules—see Section 2.3.2. Adaptation rules are fine grained units of change and collectively constitute the repertoire of adaptation actions A. As introduced earlier in Section 2.3.2, ECA rules are guarded with conditions that directly map specific event combinations to actions, i.e., modifications of the adaptable software. In our context, adaptation issues create change events. Conforming to the ECA-based adaptation policies, once a change event occurs, if any of the adaptation rule conditions are satisfied, the rules become applicable as an adaptation action to adapt the system accordingly and resolve the adaptation issue(s). VENUS is particularly tailored to capture self-healing and self-optimization of large and dynamic architectures; this entails automatic resolution of the adaption issues, e.g., runtime failures, by *general* repair and optimization actions that perform architectural adaptation and reconfiguration.

An adaptation rule r in rule set $\mathfrak{R}$ uses the patterns $P_i \in \mathcal{P}$ for already identified matches in $M_i(G)$, with $M_i(G) = \{m | G \models_m P_i\}$ as the set of matches for P_i in G, to localize an issue and to change the model, *in-place*, thus resolving the issue. Remember from Section 2.4 that in-place model transformation is a rule-based modification of a source model resulting in a target model. In this process, both models are typed over the same metamodel [45]; the automated matching of a pattern and execution of the subsequent adaptation rule constitute the self-adaptation of the software.

Additionally, the adaptation rule r is characterized by LHS and RHS patterns which define the pre-condition and post-condition of an application of $r = (\mathrm{LHS}, \mathrm{RHS})$, respectively. An application of the rule r is required if the condition, described by the LHS pattern, is satisfied, i.e., for *each* match of the LHS in the model. For $r = (\mathrm{LHS}, \mathrm{RHS})$, if a match m for LHS in the model G_i exists, then, applying the rule results in a modified model G_j by $G_i \rightarrow_{\mathrm{LHS},m} G_j$. For simplicity, we show this as $G_i \rightarrow_{r,m} G_j$. The RHS of the transformation rule determines how to modify G_i to obtain G_j. Intuitively, the execution of r searches for LHS in G_i and transforms it according to the RHS pattern—see Section 2.6.1.

VENUS operates on a graph-based realization of the architectural RTM where potential adaptation issues are captured as undesired patterns in the architecture G. Consequently, by defining ECA rules with LHS defined as the patterns that capture the issues, application of the rules entails querying the architectural RTM to identify the adaptation issues. The adaptation rule uses the LHS patterns and their identified matches to localize the adaptation issue. Next, the RHS of the transformation rule determines how to modify the model enacting the action part of the adaptation rule and thus adapts the system through causal connection to resolve the detected issue. VENUS enables analysis of the RTM via model (graph) queries and realizes the adaptation rules with in-place model transformations.

4.2 LINKING ADAPTATION RULES TO UTILITY

VENUS relies on utility theory to evaluate large, dynamic software architectures and their modification. The utility function $U(G)$ assigns a real scalar to the system architecture G indicating the desirability of G according to system objectives—see Section 3.1. The need for evaluating dynamic architectures is motivated by architectural self-adaptation. Before each adaptation, the adaptation engine has to identify a desirable target configuration and select the adaptation actions that move the system in that direction. In this section, we describe how VENUS enables this via linking utility values to adaptation rule applications.

VENUS is integrated in a MAPE-K feedback loop, thus it involves the conventional MAPE activities, i.e., Monitor, Analyze, Plan, and Execute—see Figure 2.6. The *monitoring* phase observes the current system configuration and updates the architectural model accordingly. During *analysis* and *planning*, the adaptation engine is concerned with two decisions: (I) the target configuration of the system after the adaptation; (ii) identifying the applicable rules and their matches that move the system towards the target configuration. This is illustrated in Figure 4.3 where a target configuration G_j is reachable through three alternative paths from the source configuration G_i. These two decisions are inspired by the idea of MPC that first defines a target and then predicts the optimal path to reach the target [374]—see Section 2.1.2. VENUS primarily is concerned with identifying and addressing runtime failures and performance issues. Thus, selecting a configuration where these issues are resolved is equivalent to defining the target configuration; during the adaptation, selecting the *best* sequence of the adaptation rules and their matches that resolve all issues is equivalent to building the path towards the target configuration. Paths that achieve, at an earlier point in time, larger increase of the utility are preferred. Finally, the last step of the feedback loop *executes* these rules for their matches on the running system.

VENUS captures adaptation issues as negative architectural utility patterns. Hence, occurrences of adaptation issues, i.e., finding matches for their corresponding patterns in the architecture G, negatively affects the overall utility of the architecture G, i.e., $U(G)$, relative to their corresponding utility sub-functions—see Section 3.1.2.

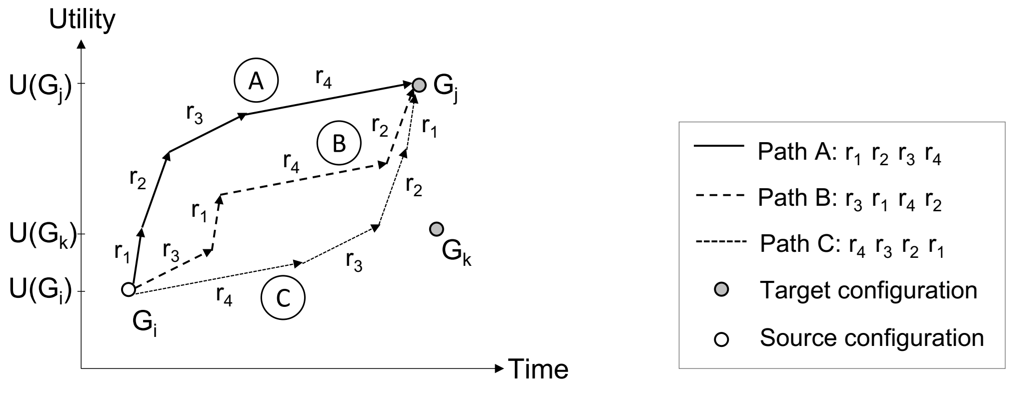


Figure 4.3: Target configuration reachable through different paths.

In the rule-based adaptation scheme, adaptation issues trigger the adaptation rules by matching their pre-condition pattern. In Venus, the application of an adaptation rule is concerned with resolving exactly *one* match for the LHS. Therefore, execution of the rule removes the match for the issue, i.e., the negative architectural utility pattern, and thus eliminates the negative impact of the issue on $U(G)$. For the proposed rule-based adaptation scheme we require that: (i) rules identify and resolve occurrences of negative patterns in the architecture; (ii), a rule application should not affect existing positive patterns, rather enable new occurrences of positive patterns by resolving occurrences of the negative patterns. The individual context of each occurrence of a pattern could cause variations in the final expected utility after the adaptation, thus plays an essential role in steering the decision-making mechanism in Venus.

Starting from a current configuration G_i that is represented by a vector of configurations and observations as $(\bar{c}, \bar{o})$, applying the rule r with a match m in G_i adapts the system to a target configuration G_j represented by $(\bar{c}', \bar{o})$. The adaptation is denoted as $(\bar{c}, \bar{o}) \rightarrow_{r,m} (\bar{c}', \bar{o})$. The graph equivalent of this notation is defined as $G_i \rightarrow_{r,m} G_j$. For the general class of self-adaptation solutions, we *compute* the utility-change $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$ during $(\bar{c}, \bar{o}) \rightarrow_a (\bar{c}', \bar{o})$ in Equation 3.8 or approximate it via the *predicted* utility-change $\hat{U}^*_\Delta(\bar{c}, \bar{o}, a)$—see Section 3.3. Venus realizes the adaptation action a via the adaptation rule r and the match m. In the context of rule-based adaptation, for an adaptation rule r and a match m in G_i with G_j as the next configuration, we define the utility-change as follows: (for simplicity, in this chapter, we use the notation U instead of $\hat{U}$—see Equation 3.5).

$$\begin{aligned} U_\Delta(\bar{c}, \bar{o}, r, m) &= U(\bar{c}', \bar{o}) - U(\bar{c}, \bar{o}) \\ &= U(G_j) - U(G_i) \end{aligned} \tag{4.1}$$

The utility-change calculation for a rule application and its corresponding match in Equation 4.1 does not require the full architectural configuration $\bar{c}$ and observation $\bar{o}$, but only sub-vectors of them, i.e., $\tilde{\bar{c}}$, and $\tilde{\bar{o}}$ respectively—see Section 3.2.2. The sub-vectors are determined relative to the match m and the selection could be specific to each rule r. In order to obtain utility-change values in Equation 4.1, we compute $U_\Delta(\tilde{\bar{c}}, \tilde{\bar{o}}, r, m)$. In Chapter 3, we discussed that sub-vectors $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$ are determined relative to the application context for the selected adaptation action. However, we argued that this assumption is only valid if the scope of the change caused by execution of a is local, that is, it does not include the whole architecture—see Section 3.2.2 and Section 3.3. In the context of rule-based adaptation, the scope of a change, caused by a rule execution, is limited to its match m. Thus, in Venus, we determine $\tilde{\bar{c}}$ and $\tilde{\bar{o}}$ relative to the match m for rule r.

For a target configuration G_j, it must hold that its utility $U(G_j)$ is higher or equal to any utility $U(G_k)$ of all possible next and intermediate configurations G_k that are the outcomes of resolving the issues in the current configuration G_i—see Figure 4.3. For the class of self-adaptation properties that concern this thesis, i.e., self-healing and limited scope self-optimization, the target configuration G_j is always reachable unless there are resource limitations.

Venus avoids enumerating the complete search space, i.e., all the reachable configurations from G_i, by computing the impact of each possible rule application for a match on the related utility sub-function, thus on the overall utility—see Equation 4.1.

After identifying the target G_j, a set of adaptation rules with their matches has to be selected to reach G_j—see Figure 4.3. A sequence of rule applications changes the configuration G_i towards G_j, which we denote as $G_i \rightarrow_{r_{i'},m_{i'}} G_{i'} \rightarrow \dots \rightarrow_{r_j,m_j} G_j$. Based on the impact of each rule application on the utility, VENUS determine the path. To resolve a single issue, alternative rules are applicable and an estimation of their impacts on the utility allows to select a conflict-free subset of them. In VENUS, we assume that the impacts of the adaptation rules on the utility are independent of each other, thus for all such sets, we can compute the utility impacts, i.e., utility-changes, regardless of the order in which the rules are executed—we mark this as a required assumption for applicability of VENUS later in this chapter.

VENUS seeks optimality with respect to the system objective satisfaction that is captured by $U(G)$ via pursuing a greedy algorithm before each adaptation. A greedy algorithm always makes the choice that produces the largest immediate gain while maintaining feasibility. There are two key ingredients in proving the correctness of a greedy algorithm [see 121]: first, the *greedy-choice property* that indicates a globally optimal solution can be arrived at by making a locally optimal, i.e., greedy choice, and second is the *optimal sub-structure property*; a problem has an optimal sub-structure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems [438]. Thus, the optimality claim in VENUS is restricted to the class of self-adaptation problems that satisfy the *greedy choice property* where a global optimal solution can be reached by choosing the optimal choice at each step [see 6]. In the light of this, VENUS guarantees the following achievements for each adaptation:
(i) executing the selected rules eventually leads to the target G_j *with optimal utility* $U(G_j)$.
(ii) executing the selected rules in the right order results in the highest achievable reward, i.e., accumulated utility over time. The area under each path curve in Figure 4.3 represents the reward.

VENUS fulfills (i) by pursuing a greedy choice when there are two or more alternative rules to resolve the same issue. The scheme selects the rule with the highest (expected) impact on the utility. As discussed above, provided by the greedy choice property of the targeted problems, VENUS arrives at a globally optimal solution by making locally optimal choices. To achieve (ii), the choice that produces the largest immediate gain is prioritized, thus VENUS executes the selected rules in a decreasing order of their impact on the utility. To maximize the reward, VENUS offsets the designated utility increase with the estimated cost of executing each rule. The collective fulfillment of (i) and (ii) suggests that VENUS is optimal regarding both the final utility $U(G_j)$ and the achieved reward. Reward optimality in the context of (i) is achieved by selecting the best rule in terms of utility impact for each issue. In the context of (ii), the scheme is optimal since it prioritizes those rules from the selected set that have larger impact on utility. In both cases, if multiple rules have the same impact on utility, cheaper rules, i.e., rule with lower cost, are prioritized as cost, besides utility, also affects the reward.

Employing utility functions to evaluate dynamic architectures at runtime, as defined in Chapter 3, enables optimization-based approaches for self-adaptation. These approaches search the configuration space and compute the utility for each possible next configuration; however, such a solution for making adaptation decisions does not scale if the utility is computed for each configuration completely anew. Large configuration spaces impose additional challenge for optimization-based approaches. In contrast, VENUS determines at runtime the impact of each *possible rule application* on the utility. This way, the search space for next configuration is limited to those reachable from

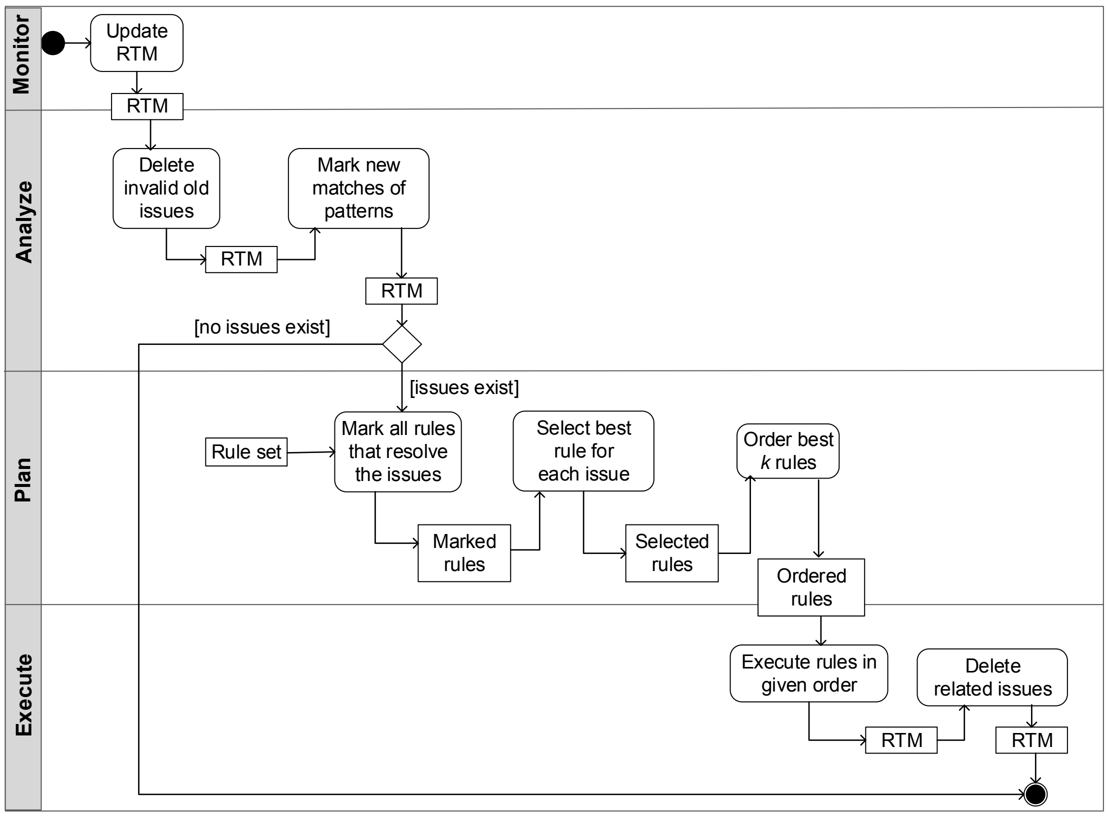


Figure 4.4: Activities of MAPE-K feedback loop realizing Venus.

the current configuration and via the specified rule execution. Based on these impacts, Venus selects the *optimal* adaptation rule to address each issue and identifies the *optimal* sequence of the rule executions required to maximize the reward.

4.3 REALIZING VENUS IN A FEEDBACK LOOP

Venus is realized in a MAPE-K feedback loop that is amenable to incremental execution. In this section, we detail our approach for incremental execution of an adaptation engine where the Analyze and Plan activities are implemented by Venus. We discuss Venus and its features following the MAPE-K blueprint using the running mRUBiS example—see Section 2.5—to discuss all aspects of Venus.

4.3.1 *Feedback Loop*

Figure 4.4 visualizes the steps of Venus, integrated in the MAPE-K feedback loop, in the form of an activity diagram. The MAPE-K blueprint for feedback loops has been discussed in Section 2.3.2 and it considers a sequence of Monitor, Analyze, Plan, and Execute activities that all share and jointly operate on some Knowledge.

As discussed earlier, Venus implements the Analyze an Plan activities and is integrated in a feedback loop where the Monitor and Execute activities also support incrementality, therefore, collectively construct a complete adaptation loop. The adaptation engine uses an architectural RTM of the adaptable software to monitor the system and reason about the required changes. To handle the individual adaptation activities and the RTM within a feedback loop as well as the interplay between them, the adaptation loop adopts the principles of MDE—see Section 2.4. Thus, the MAPE activities are considered as model operations that operate on an RTM. The interplay is reflected by the control flow among the operations and the model usage defining how operations use the RTMs.

The MAPE activities, including VENUS, leverage event-based processing of the changes, thus enable incremental execution of the adaptation loop. The execution of the loop is triggered by every event that notifies about the changes of the adaptable software. The feedback loop is not reentrant, thus all events that occur *during* a feedback loop execution are queued to be processed by the next loop. When the current feedback loop run is finished, the next run is triggered if there is at least one queued event. The loop execution continues until it processes all of the currently queued events. Thus, VENUS does not required any static, time-based frequency to be determined manually for executing the feedback loop.

All of the four MAPE activities operate on the architectural RTM. The activities in the diagram in Figure 4.4, indicated by the rounded rectangles, work on the data objects that are represented by rectangles. Figure 4.4 shows one RTM that captures the knowledge part of MAPE-K. The usage of data objects like RTM as inputs or outputs of an activity is defined by arrows pointing to the activity or to the data object, respectively.

Throughout multiple executions of the adaptation loop, the Monitor activity updates the RTM in each run, therefore, the RTM is not created from scratch in each run of the feedback loop, rather is preserved. Temporary updates of the models are allowed via model *annotations*, e.g., to exchange information among activities, such that they can be destroyed by activities after they have fulfilled their purpose. For instance, the Monitor operation adds the symptoms to the RTM to provide hints for the Analyze operation that could destroy the annotations after having used these hints. Likewise, the Plan and Execute operations could destroy the markers created for the applicable and selected rules when having finished the planning respectively the execution.

During monitoring, the model is updated to reflect the changes of the system. The analysis deletes the *old* issues from the model. Old issues are matches of the negative patterns that have been identified by a previous run of the loop that are not valid anymore. In addition, the new issues are detected and annotated in the RTM during analysis. The subsequent planning considers all possible adaptation rules in the rule set that can address the existing issues. For each applicable rule, i.e., marked rule in Figure 4.4, the impact on the utility and cost of execution are calculated. For each issue, the best rule regarding the impact and cost is selected. The selected rules over all issues are sorted according to their utility impact and cost. In the execution phase, the ordered list of rules is executed on the model and thus on the running system followed by removing the corresponding issues from the RTM. The incremental monitoring derives from work of Vogel et al. [417, 418], while the incremental analysis, planning, and execution mechanisms are new to this thesis. In the following, we provide a detailed description of the MAPE-K feedback loop activities realizing VENUS.

As mentioned earlier, modifications to the RTM in VENUS are done via SDs as the modeling language that realize the graph (model) transformation rules. An SD structures an initial, a final, and one or more action node(s) in a control flow that characterizes an activity. Action nodes are realized by Story Patterns (SPs) or Call Actions. An SP describes and executes a graph transformation while a Call Action invokes either another

SD or code. SDs are similar to ECA rules and capture the LHS and RHS of a graph transformation rule. They allow to specify patterns at the same level of abstraction as the RTM. The Analyze, Plan, and Execute activities of the MAPE-K loop in Venus are realized via SD-based rules and are referred to as analysis, planning, and execution rules respectively—see Appendix A for more details on the SD formalism.

4.3.2 *Monitor*

During monitoring, owing to the causal connection between the adaptable software and the RTM, enabled by MDE techniques, change events emitted by the system are reflected in the RTM. Employing MDE principles allow the adaptation engine to maintain the causal connection, thus incrementally updating the architectural RTM based on changes of the running system. The causal connection serves as a layer that links the model space to the runtime space [see 310], i.e., an infrastructure that translates a change to the system into a change in the RTM and vice-versa—see Section 2.4. The Monitor activity observes the adaptable software through the sensors provided by the adaptation engine, aggregates, and filters the gathered data to determine the symptoms that need to be handed to the Analyze activity. According to the observations, the Monitor activity updates the RTM of the adaptable software and creates symptoms that should be analyzed.

We implement the incremental monitoring proposed by Vogel et al. [417, 418] where the system and the RTMs are causally connected and changes in the system are incrementally reflected in the RTM during monitoring via TGG rules. TGG rules, defined at the level of metamodels, specify by means of model transformation rules how the architectural RTM, reflecting the adaptable software, is synchronized with the software system. Thus, monitoring the system keeps the architectural RTM up-to-date.

In mRUBiS, the monitoring operation is realized with code that invokes the causal connection to incrementally update the architectural RTM based on changes of the running mRUBiS. For instance, changes of the lifecycle state of a component, e.g., indicating a stopped, crashed, or removed state, are observed during monitoring and the RTM of mRUBiS is updated to reflect those changes and current state of mRUBiS accordingly–see the mRUBiS metamodel in Figure 2.8.

4.3.3 *Analyze*

During the Analyze activity, the updated RTM is analyzed for any symptoms that might indicate a need for adaptation. Analyze is a model-driven activity as models are used to specify and execute the analysis; on the other hand, it is event-/state-based as the analysis is driven by change events resulting from updates of the architectural RTM, i.e., of the state, during monitoring. The change events contain locality information as they point to the parts of the RTM which have changed during monitoring. Thus, the analysis can be performed directly on the affected parts of the RTM without having to search and check the whole model, which supports incremental processing of the RTM.

In this phase, the updated RTM is analyzed to check whether known matches of negative patterns, i.e., old issues, are still valid; otherwise, the annotations representing these issues are removed from the RTM—see the activity diagram in Figure 4.4. Next, the updated RTM is analyzed to detect new matches of negative patterns, i.e., new

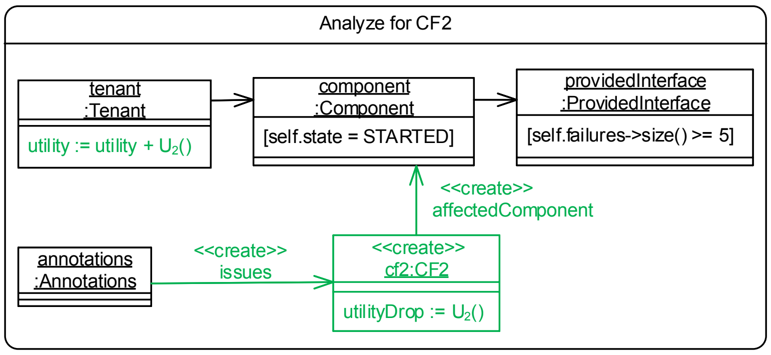


Figure 4.5: Annotating an occurrence (match) of a negative pattern in RTM.

issues, that enrich the known set of matches. This analysis is driven by events notifying about model updates.

VENUS computes and adds new matches of newly observed negative patterns while preserving the set of old matches that are still valid. The invalid matches are simply removed. Detection of new matches entails drops in the utility caused by new occurrences of negative patterns—see Section 3.1.2. The output of the analysis is the updated set of matches for negative patterns, marked in the RTM, along with their impacts on the system utility. The updated RTM is then handed over to the Plan activity.

As a first step in computing the impact of the newly detected matches on the system utility, VENUS computes the utility incrementally, rather than for each configuration anew. Given a former RTM G, i.e., before occurrence of new matches, and an updated version G', the set of new matches for the pattern P_i—see Section 3.1—is $M_i^{new} = M_i(G') \backslash M_i(G)$. Similarly, $M_i^{del} = M_i(G) \setminus M_i(G')$ contains the matches for the pattern P_i in G that are no longer valid in G'. The patterns represent adaptation issues, thus are negative architectural utility patterns $\mathcal{P}^- = \{P^-_{k+1}, \dots, P^-_n\}$. We define the utility sub-function U_j for a match m of a negative pattern P^-_j in architecture G, that is, $U_j(G, m)$, as a negative real value—see Section 3.1.2. We can therefore calculate the corresponding change of the overall utility $U(G') - U(G)$ via the utility-change function $U_\Delta(G', G)$ in Equation 4.2. Besides computing the change of the utility, VENUS keeps track of the newly identified matches for the negative patterns by marking them in the RTM, i.e., adding annotations to the model.

$$U_\Delta(G', G) = -\sum_{i=1}^{n} \sum_{m \in M_i^{del}} U_i(G, m) + \sum_{i=1}^{n} \sum_{m \in M_i^{new}} U_i(G', m) \qquad (4.2)$$

The Analyze activity consumes the change events for the RTM; based on these events, it runs the relevant analysis by invoking the SD interpreter to execute the corresponding SDs. In the context of mRUBiS, the Analyze activity checks the RTM for occurrences of negative patterns and adds Annotations to the RTM that mark the issues pointing to the relevant affectedComponent. In this thesis, we use the self-healing and self-optimization properties of mRUBiS to discuss VENUS—see Section 2.5. We consider the following issues: crashes (CF1) and removals (CF3) of components, occurrences of Failures in terms of exceptions (CF2), connector crashes (CF4), and sub-optimal service performance (PI1)—see Figure 2.8 for metamodel of mRUBiS. For instance, if exceptions have occurred when using a providedInterface of a component, the model is updated by representing these exceptions as failures attached to the providedInterface in the RTM. Such an update causes a specific change event that indicates a potential occurrence of a CF2 issue. Figure 4.5 shows an analysis rule in mRUBiS realized by an SP that detects the negative pattern P_2^-—introduced in Figure 3.4. The green parts are modifications to the RTM elements during analysis. The occurrence of the negative pattern results in a drop in the tenant utility equal to $|U_2|$. This rule creates the CF2 annotation (in green) in the RTM with the computed utilityDrop that points to the affected component; we omit the details to avoid multiple annotations for the same issue.

4.3.4 *Plan*

If the Analyze activity does not discover any issues in the RTM, Venus skips the Plan and Execute activities of the MAPE-K loop and terminates immediately after the RTM is analyzed; otherwise, Venus performs the Plan activity followed by the Execute—see Figure 4.4. Based on the annotations indicating new or remaining matches for negative patterns, i.e., issues in the RTM, Venus incrementally proceeds during the planning by following three steps: (1) compute the set of all possible adaptation rule applications (matches) that resolve each issue, (2) select the best rule application for each issue based on the impact on the utility and cost, and (3) order the best rule applications across all issues to maximize the reward. In the following, we elaborate on the three steps of the Plan activity in Venus:

4.3.4.1 *Compute all possible adaptation rule matches*

In self-healing and self-optimization, applying an adaptation rule always leads to an improved utility as it resolves an issue that has previously caused a drop in the utility. For this case, we will show that adaptation rules have to be linked to negative patterns, for their removals improves the utility. Moreover, knowing the matches for these patterns allows Venus to incrementally compute all relevant adaptation rule matches, i.e., all the rules that are applicable to resolve the issues. In the context of Venus, an adaptation scheme with pattern-based utility functions, all the patterns that need to be matched and resolved are negative patterns, i.e., upon occurrence, cause a drop in utility, and the following requirements must hold:

Req 4.1 *if there are no matches of negative patterns, there is no need for adaptation and no improvement of the utility is possible.*

Req 4.2 *any improvement of the utility must necessarily resolve the identified matches of negative patterns, otherwise no improvement is possible.*

Thus, we can safely assume that: (A1) for any adaptation rule $r_j = (LHS, RHS)$ in the set of all adaptation rules $\mathfrak{R}$, where LHS of the rule is captured by pattern P_j, a negative pattern P_i^- exists such that any match m_j for P_j (m_j makes r_j applicable) includes a match m_i for P_i^- (m_i denotes an occurrence of the negative pattern P_i^-). Thus, any adaptation rule must be linked to a negative pattern such that the rule can only be applied if there is an occurrence of the negative pattern. Otherwise, the rule could be applied even though there is no occurrence of a negative pattern, and no utility improvement can be achieved, which contradicts ReQ4.1. It can be the case that the

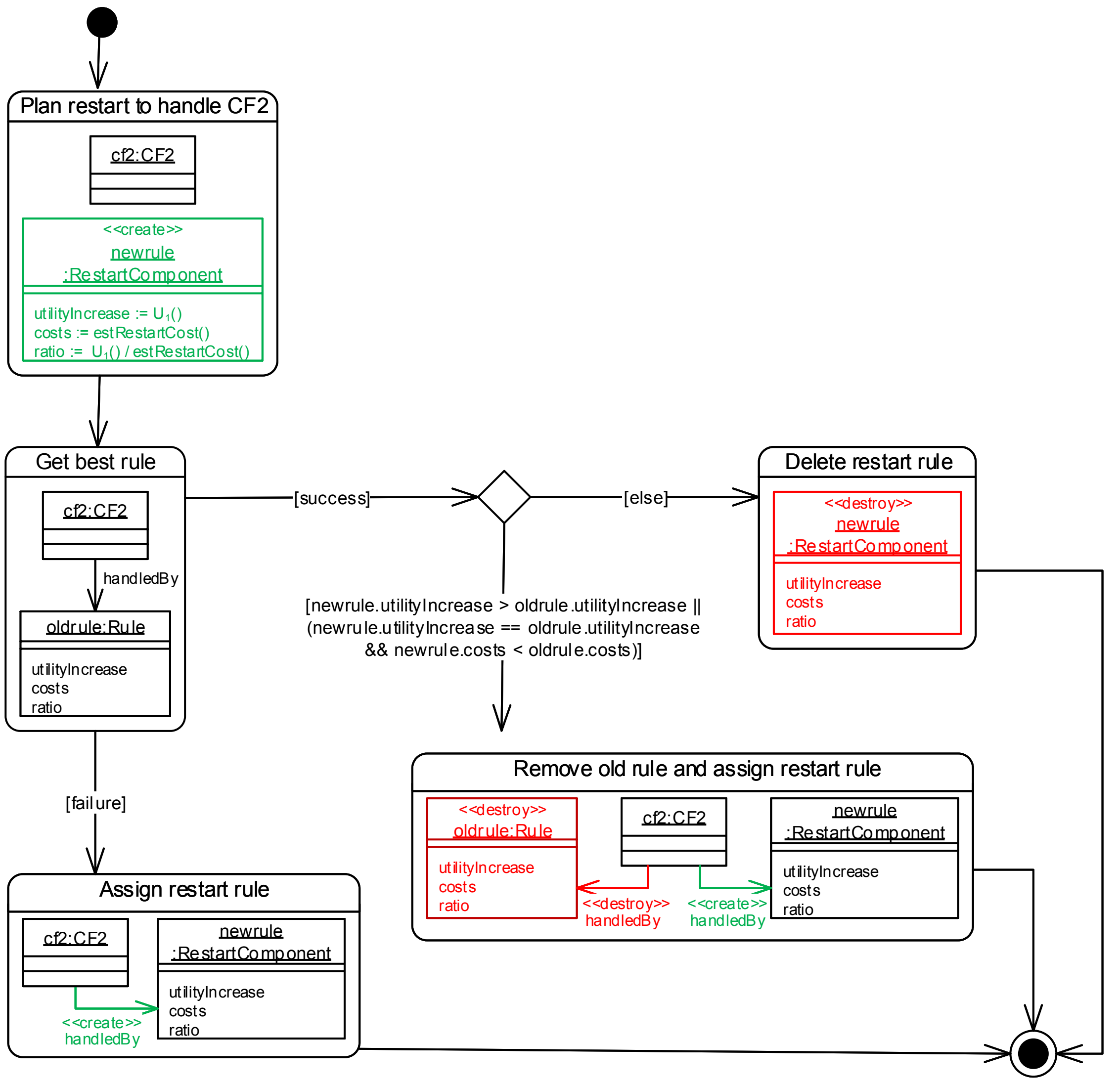


Figure 4.6: SD of a planning rule.

LHS of r_j has a larger context and is, thus more restricted than the negative pattern P_i^-. However, both patterns are exactly the same in the presented examples in this chapter.

Furthermore, we can plausibly assume that: (A2) for rule $r_j = (LHS, RHS)$ in $\mathfrak{R}$, where LHS is captured by P_j, and any match m_j for P_j with the included match m_i for the related negative pattern P_i^-, applying r_j for m_j will make the match m_i invalid. This means that executing an applicable rule resolves the related occurrence of the negative pattern by resolving the issue. Otherwise, execution of r_j does not resolve the identified occurrence of the negative pattern P_i^-, therefore, does not lead to the improvement of the utility as expected by ReQ4.2. VENUS only considers the case where each rule covers exactly *one* negative pattern. Based on the assumptions (A1) and (A2), during planning, VENUS incrementally computes all the matches for rule applications provided with the set of new matches M_i^{new} for the related negative pattern P_i^-.

Following the MAPE-K feedback loop to conduct adaptations in VENUS, the Plan activity decides which adaptation rules among all possible ones should be applied; next, the Execute activity actually applies the chosen rules to prescribe an adaptation of the RTM that is subsequently, owing to the causal connection between the RTM and the running system, propagated to the system—see Section 2.4.

In the context of applying Venus to mRUBiS, the Plan activity selects the adaptation rules to be executed by modifying the RTM with Rule annotations that will handle the identified Issues—see the metamodel of mRUBiS in Figure 2.8. These rules are finally enacted by the Execute activity. As adaptation rules, mRUBiS support restarting, redeploying, and replacing components, adding and removing service replica, as well as recreating connectors—see Section 2.5. To realize the planning rules, similar to analysis rules presented in Section 4.4, we use SDs. Using SDs, we can structure SPs (nodes of the rule) in a control flow that complies with *Unified Modeling Language* (UML) activity diagrams—see [141].

Figure 4.6 presents a planning rule in mRUBiS for an issue of type CF2. The first node of the planning rule matches the CF2 annotation, created by the analysis phase, and creates the new adaptation rule of type RestartComponent (shown in green) as a Rule annotation to resolve the CF2 instance. Next, Venus decides for the adaptation rules to be executed by selecting the best among all applicable rules for each issue.

4.3.4.2 *Select best adaptation rule match for each negative pattern match*

Each issue in the form of a negative pattern match may be resolved by alternative, i.e., more than one, adaptation rules. Thus, for each issue, the planning must select one rule for execution. To determine the *best* among all applicable adaptation rules for each issue, Venus computes the impact on the utility as well as the cost of each possible rule execution. Formally, for a single rule $r_j = (LHS, RHS)$ where LHS is captured by pattern P_j and P_j extends the negative pattern P_i^-, the assumption (A2) indicates that each time r_j is applied to m_j with m_j a match for P_j, then the match m_i for a negative pattern P_i^- is removed. We further assume that: (A3) execution of r_j does not result in any new match or removed matches besides m_i for any negative pattern. Then, we conclude for any architectural RTM G and the resulting configuration G' from applying rule r_j to G for match m_j ($G \rightarrow_{r_j,m_j} G'$) that:

$$U_\Delta^{r_j}(G, m_j) = U_\Delta(G', G) = U_i(G, m_i) \tag{4.3}$$

$U_\Delta^{r_j}(G, m_j)$ in Equation 4.3 is obtained according to Equation 4.1 and can be computed either based on analytically defined utility functions (Section 3.2), or based on learned utility-change prediction models (Section 3.3).

If the assumptions (A1) to (A3) hold, Venus *locally* computes the impact of the adaptation rules on the utility. We further assume that (A4) rule r_j does not affect any utility sub-function for any match m_k of another negative pattern P_k^-. Consequently, applying r_j for a match m_j does not affect the impact on the utility of any other rule r_k for match m_k. Thus, if (A1) to (A4) hold, Venus can *independently* and *locally* compute the utility impact of each rule application.

However, there can be cases where the RHS of a rule r_j with $G \rightarrow_{r_j,m_j} G'$, results in new matches for one or more *positive* patterns. In such cases, the impact on the utility, caused by the corresponding positive utility sub-function of the matches, is added to $U_\Delta^{r_j}(G, m_j)$ in Equation 4.3. For this reason, we assume that: (A5) all the potential positive patterns are completely within the scope of the application condition and side effect of r_j and do not match only partially. Otherwise, matches for the positive patterns cannot be enabled by applying r_j. Thus, the impact can be considered in $U_{\Delta}^{r_j}(G, m_j)$ because the formula in Equation 4.3 for the corresponding increase of the utility can be determined off-line, i.e., at design time. When repairing the local authentication component of mRUBiS via replacing it with a third-party service, e.g., google authentication service, each available service offers different reliability, thus results in a different increase of the utility accordingly—see the definition of the linear utility for mRUBiS in Table 3.2. If the selected service offers a higher reliability than the replaced service, not only the repair restores the dropped utility to its former value, i.e., before the local authentication component fails, but also further increases the utility due to its higher reliability that is captured as a positive pattern in the RTM. This is an example of a case where removing a match for a negative pattern results in a match for a positive pattern.

In the process of selecting the best adaptation rule, besides the estimated impact on utility, the cost of rule applications are also considered. VENUS considers the cost of adaptation rules in terms of their estimated execution time by means of a cost function $Cost^{r_j}(G, m_j)$ for each application of a rule r_j. The estimated execution time may depend on the match m_j with its context in G. Hence, for each issue, the planning rules in VENUS determine the expected utilityIncrease $U_{\Delta}^{r_j}(G, m_j)$ and costs $Cost^{r_j}(G, m_j)$ of executing each applicable adaptation rule on the running system before selecting the best among all applicable adaptation rules.

Following the analysis phase where matches of the negative patterns are marked as annotations of the RTM, e.g., annotation for CF2 issues in Figure 4.5, during the planning phase, all the planning rules in $\mathfrak{R}$ are checked if they match any of the annotations in the RTM. Rules that match are considered to be selected for planning. In the planning rule example presented in Figure 4.6, the first SP of the rule matches the CF2 annotation and creates the Rule annotation of type RestartComponent. It further determines the utilityIncrease and costs of restarting the component—see attributes of newrule:RestartComponent. For this purpose, the corresponding utility sub-function U_1—as discussed in Section 3.1—is used to calculate the expected utilityIncrease and the impact of the rule execution on the utility. The repair rule restores the dropped utility, therefore, the utilityIncrease value assigned to newrule:RestartComponent is equal to the utilityDrop caused by CF2 as shown in Figure 4.5, i.e., $U_2 = |U_1|$. This utility sub-function takes the context of the match and thus runtime information into account, e.g., the reliability and criticality of the affected component—see definition for linear U in Table 3.2.

Cost functions such as estRestartCost() in Figure 4.6 for each adaptation rule type estimate the costs of executing the rule, e.g., restarting a component in the system. In mRUBiS, the cost estimation for each rule type is static, context-independent, and based on past measurements of the time that is needed to execute the corresponding change to a running system. However, VENUS does not require the cost values to be defined statically; the cost functions can be more elaborate taking the context of the match into account, e.g., it is more costly to replace a component with a higher connectivity, i.e., a larger number of associated connectors.

Based on the utilityIncrease and costs of all applicable adaptation rules for an issue, VENUS proceeds with selecting the best rule. The planning checks for each applicable adaptation rule, e.g., newrule in Figure 4.6, whether it results in a higher utilityIncrease than the rule that is so far determined as the best rule within the current run of the feedback loop—see oldrule:Rule in Figure 4.6. In the case that both rules yield equal utilityIncrease values, VENUS checks if the new rule has lower costs. Therefore, VENUS prioritizes the utilityIncrease over costs. If the old rule, i.e., the rule that has been determined as the best rule so far, does not exist, the planning selects the new rule to handle the CF2 issue–see SP Assign restart rule. Otherwise, if the new rule is better than the old rule in terms of utilityIncrease and costs, the old rule is deleted and the new rule is selected—see SP Remove old rule and assign restart rule in Figure 4.6. For the case where the old rule is better than the new rule, the planning proceeds with the old rule and deletes the annotation for the new rule—see SP Delete restart rule. This way, the planning rules select for each issue the best adaptation rule for execution. The rule is then associated to the issue by the handles/handledBy association in the RTM—see the metamodel of mRUBiS in Figure 2.8 defining the relationship between Rule and Issue.

4.3.4.3 *Order all selected adaptation rule matches*

The final planning step in Venus determines the sequential order to resolve the issues during a MAPE-K execution if multiple issues are to be resolved in one cycle—see Figure 4.4. Let us assume that during one cycle of the feedback loop, maximum of k issues are resolved, thus k adaptation rules may be executed, e.g., due to time or resource constraints. Therefore, the k best adaptation rules, selected during the previous planning step, are sorted in descending order regarding their impact on the overall utility divided by the costs. This metric combines the benefits and costs of the adaptation rules and is reflected by the ratio attribute of a Rule, see Figure 2.8. Venus calculates the ratio value via the planning rules—see Figure 4.6. Applying the adaptation rules in this order, as maintained by the association Annotations.bestRules in the RTM (see Figure 2.8), guarantees that the maximal utility is re-established as fast as possible in the execution phase and that the loss of reward is minimized. As depicted in Figure 4.3, prioritizing the adaptation rules with a higher impact on utility, i.e., rules with larger slopes, maximizes the area under the curve. A curve is a path to the target configuration which is recognized as utility over time or reward. During planning, Venus maximizes this area via pursuing the policy of increasing the utility as fast as possible resulting in an optimal adaptation sequence in terms of system utility and reward. As depicted in Figure 4.3, path A comprises of a rule sequence sorted in descending utility values and, compared to path B and C that employ different ordering or rules, yields the largest area under the curve, hence the largest reward.

4.3.5 *Execute*

Provided with the sorted list of adaptation rules by the planning phase, this phase executes the rules accordingly in a sequential manner—see Figure 4.4. Thus, each issue is handled by the most appropriate rule. The rules are executed in the order such that those yielding the best trade-off, i.e., ratio of utilityIncrease and costs, are prioritized. Similarly to the Monitor activity, this phase follows an incremental scheme in executing the adaptation rules on the RTM and propagating the corresponding changes through the causal connection to the running system.

The synchronization of the adaptable software and its RTM is the causal connection problem—see Section 2.4, i.e., propagating changes from the software to the model by the Monitor activity and vice versa by the Execute activity. This problem can be generically solved because in this thesis we are concerned with generic changes at the architecture level, e.g., creating and deleting components and changing their interconnections and parameters [see 273, 302]. Thus, changes of the software that are monitored or adap-

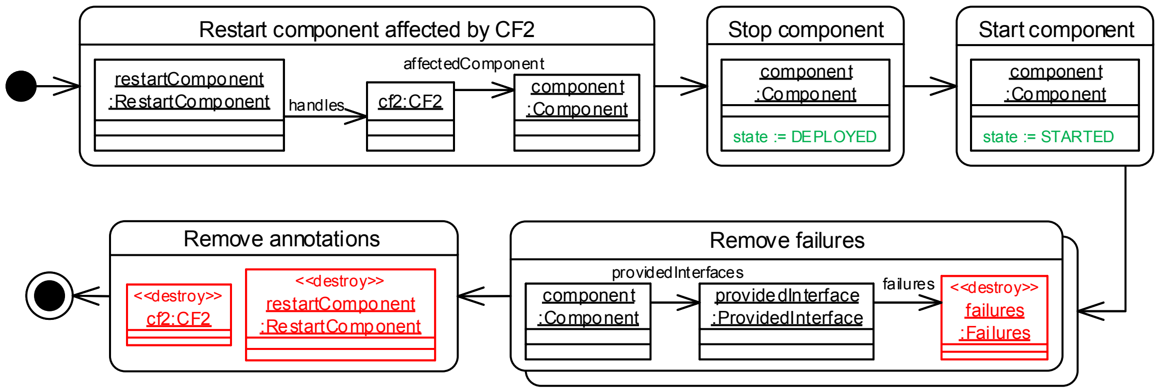


Figure 4.7: Rule for executing a component restart.

tations that are planned can be generically described as such architectural changes and used by the causal connection regardless of the addressed self-adaptation capability, e.g., self-healing or self-optimization.

Figure 4.7 illustrates an execution rule to restart a component in mRUBiS for addressing the CF2 issue. Based on the analysis and planning phases, see Figure 4.5 and Figure 4.6 respectively, an adaptation rule, in this case restartComponent, has been selected to handle CF2 affecting the specific component—see the first node in Figure 4.7; this component is then restarted by setting its lifecycle state to DEPLOYED, i.e., stopped, and then to STARTED—see the second and third nodes. Next, the RTM is cleaned by removing, i.e., destroying, the observed exceptions (failures) and the annotations for the executed rule (restartComponent) and the handled issue (CF2).

4.4 ASSESSMENT OF VENUS

In this section, we analyze and discuss the computational effort as well as the optimality of VENUS in terms of utility and reward. First, we present the generic algorithms for the Analyze and Plan phases of VENUS and show that their computation is done incrementally and that resulting adaptation leads to optimal reward. Next, we discuss the limitations imposed by the assumptions for VENUS.

4.4.1 *Detailed Algorithms for Analyze and Plan*

The ANALYZE and PLAN functions presented in ALGORITHM 4.1 and 4.2 respectively, constitute the Analyze and Plan activities of the MAPE-K loop in VENUS. Both algorithms use a global data structure, i.e., an RTM, defined by a corresponding metamodel—as an example, see excerpt of mRUBiS metamodel in Figure 4.8 showing annotations. The singleton object $annotations$ of type Annotations in Figure 4.8 captures the current matches for the issues with the $issues$ association of type Issue. Annotations also captures the matches for the best k rules to resolve the issues with its bestRules association of type Rule. The matches for the issues in the RTM and their assigned rules are accessed by $annotations.getAllIssues()$ and $annotations.getAllBestRules()$ in ALGORITHM 4.1, respectively. Additionally, the association handledBy between the matches for the issues and the matches for the rules maintains the best rule match for each issue.

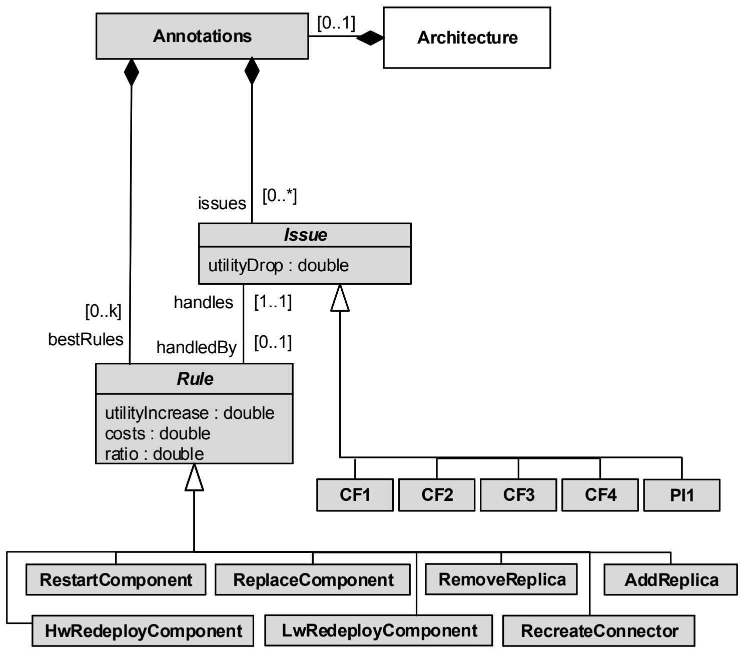


Figure 4.8: Excerpt of mRUBiS metamodel showing annotations.

For an Issue, we assume that a test, namely check(), exists that checks in constant time whether the match of the issue is still valid. To maintain the matches of the issues in a global list without double entries, three constant time operations are considered: annotations.existsIssue(), annotations.addIssue(), and annotations.deleteIssue(). The operations check for existence of, add, and delete issues respectively. The constant time operations are possible when using indexed data structures with unique matches as indices[2]. Each issue is always linked to one unique component (see Figure 2.8) to realize the annotations.existsIssue() test. The test checks whether the same issue already exists for the same component.

Similarly, Analyze employs constant time operations annotations.addBestRule() and annotations.resetBestRule() to maintain and clear the list of matches for the best k rules with respect to their *ratio* attributes—see Section 4.3.4. k is chosen large enough to cover as many rule applications as possible to fit into the time window of a single MAPE-K run. The dedicated time window of a MAPE-K run is a design decision. A shorter window allows more frequent planning. As a result, rules achieving a high impact on the utility are executed earlier and not blocked by rules that achieve lower increases but are scheduled for execution in the running MAPE-K cycle. This is beneficial if the planning is time-efficient. For a selected time window, k can be determined using estimates of the planning time of the scheme and rule execution time. Thus, k can vary for different time windows accordingly. Since at most k adaptation rules are applied in a single MAPE-K run, k is a constant upper bound on the rule elements that have to be stored and ordered in the RTM. A run of the feedback loop is finished after the k^{th}

2 An indexed database, e.g., a hash table, allows for constant-time retrieval and examination of data items based on the value of the item's index. However, the constant time complexity, i.e., $O(1)$, of the operation in a hash table is pre-supposed on the condition that the hash function doesn't include any colliding indices [406]; thus the performance of the hash table is directly proportional to the chosen hash function ability to disperse the indices. An indexed database with unique indices supports $O(1)$ performance [see 159].

ALGORITHM 4.1 : ANALYZE(C)

```
1  forall issue ∈ annotations.getAllIssues() do        // for all old issues in C
2      if ! issue.check() then                          // delete old issue if no longer valid
3          annotations.deleteIssue(issue) ;             // delete issue from global list
4      end
5  end
6  forall c ∈ C do                                      // iterate over modified or created elements
7      forall $P_i \in \mathcal{P}$ containing a node that may be matched to c do
8          forall $m : G \models_m P_i \wedge m$ contains c do     // find all new issues for $P_i$
9              if !annotations.existsIssue($P_i$, m) then         // issue for $P_i$, m exists?
10                 annotations.addIssues(new Issue($P_i$, m)) ;   // add new issue
11             end
12         end
13     end
14 end
```

adaptation rule has been executed. The remaining issues that have not been addressed in this run will be handled in the subsequent run of the feedback loop.

ANALYZE in ALGORITHM 4.1 requires as input the set C of elements in the RTM that have been updated (modified) by the Monitor activity—see Figure 4.4. Next, ANALYZE (i) removes the old issue in annotations.getAllIssues() that are no longer valid (line 1-5); (ii) iterates over all changes $c \in C$ (line 6-14) for all patterns in $\mathcal{P}$ that are relevant for c (line 7-13) and for all possible new issues whose matches contain c (line 8-12) to check whether these issues are new (line 9), and if yes, the issues are added to the global list (line 10). Thus, after executing ANALYZE, the set of issues stored in annotations.issues is up-to-date.

PLAN in Algorithm 4.2 describes the steps of the Plan activity in VENUS. The algorithm initiates with clearing the global list of the best k rule matches (line 1). Next, it checks for all the current issues in annotations.getAllssues() (line 2-14), for all the rules that may resolve the match of the issue (line 3-11), and for all the matches of these rules that extend the match of the issue (line 4-10), whether the new rule match is better than the current best rule match captured by issue.getHandledBy(). The new rule is considered better if it causes higher increase of the utility, or if the increase is the same, is has lower cost. If such a match is detected, the current best rule match is replaced by the new rule match. Thus, for each issue, the best rule match is determined and added to the global list of k best rules (line 13).

4.4.2 *Computational Effort of Analyze and Plan in* VENUS

During the ANALYZE and PLAN in ALGORITHM 4.1 and 4.2, detecting matches for the patterns that represent the issues and rules in VENUS does not require a global search, rather a local search that starts from a change, a marking, or an existing match. We can safely assume that the patterns have a constant upper bound concerning their size as their sizes is bounded by the size of the architecture G. Further more, it is assumed that: (A6) the association links which have to be traversed by a local search for matches have a small constant upper bounds. As a result, finding a single match for an issue or a

ALGORITHM 4.2 : Plan()

```
1  annotations.resetBestRules() ;        // erase all entries from list of best rules
2  forall issue ∈ annotations.getAllIssues() do
3      forall $r_i \in \mathfrak{R}$ that can resolve issue do
4          forall $m : G \models_m r_i$ with $m \supseteq$ issue.match() do
5              rule := new Rule(m) ;                              // create new rule
6              oldrule := issue.getHandeledBy() ;           // get best rule until now
7              if oldrule = NULL || rule.utilityIncrease > oldrule.utilityIncrease ||
                (rule.utilityIncrease == oldrule.utilityIncrease &&
                rule.utilityIncrease/rule.cost > oldrule.utilityIncrease/oldrule.cost) then
8                  issue.setHandeledBy(rule) ;        // rule is better than old one
9              end
10         end
11     end
12     rule := issue.getHandeledBy() ;                         // get overall best rule
13     annotations.addBestRules(rule, rule.utilityIncrease/rule.cost) ; // keep k best
         rules for all issues according to ratio
14 end
```

rule requires only a constant computational effort, i.e., $O(1)$. Based on the assumption (A6), that typically holds, we next show that the algorithms for the Analyze and Plan only require incremental computational efforts in $O(\Delta + \Delta')$ where Δ is the number of changes in the RTM and Δ' is the number of the unprocessed issues.

In Analyze (ALGORITHM 4.1), the first loop (line 1-5) requires $O(\Delta')$ steps to check whether the Δ' many old, unprocessed issues in $annotations.getAllIssues()$ are still valid. Conducting such a single check requires constant time. The second loop (line 6-14) iterates over all the changes $c \in C$ with $|C| = \Delta$ and thus results in $O(\Delta)$ iterations. The inner loop (line 7-13) considers all the patterns in $\mathcal{P}$ that potentially match c; the loop has a constant number of iterations because of the small number of patterns, a constant subset of which can actually be matched to c. The other inner loop of the algorithm (line 8-12) considers all the matches for the patterns that actually include c and is bounded by the number of the changes $|C|$. Therefore, it requires $O(\Delta)$ iterations. Thus, Analyze requires an incremental computational effort in $O(\Delta + \Delta')$ to process the existing issues and create the list of current issues to be handed over to Plan.

Plan in ALGORITHM 4.2 as a first step erases the list $annotations.bestRules()$ in constant time. The list contains k best rule matches from the last cycle (line 1). Next, maximum number of $\Delta + \Delta'$ issues are handled in the outer loop (line 2 - 14) in $O(\Delta + \Delta')$; for each iteration processing an issue, constant many rules that may resolve the issue at hand are considered (line 3-11). In the worst case, the number of these rules is equal to $|\mathfrak{R}|$, i.e., the number of the adaptation rules in the finite set of all adaptation rules $\mathfrak{R}$. Thus, the number of the considered rules is limited by this constant upper bound and can be checked in $O(1)$. As a result, constant number of rule matches—each considered rule either matches or not—that extend the match of the issue are considered in the loop from line 4 to 10. While iterating through all the applicable rules for an issue, bounded by $|\mathfrak{R}|$, one *best* rule regarding the utility increase and cost is selected. Therefore, only constant number of times the best rule match for an issue is updated and stored in $issue.getHandledBy()$ in line 8. Finally, in line 12, the *best* rule match among the alternatives to resolve each match of an issue, stored in issue.getHandledBy(), is selected and added to the annotations (line 13). As discussed earlier, VENUS restricts the number of issues that can be resolved within a feedback loop run to k—see Section 4.4.1—thus constant number of best rule matches, maximum of k, are kept by annotations.addBestRule() in line 13, yielding a constant computational effort for this step of the algorithm. Consequently, PLAN requires an incremental computational effort in $O(\Delta + \Delta')$ and the resulting list of the best rule matches contains k elements, i.e., $O(1)$.

The Monitor and Execute activities of VENUS are implemented in an event-based and incremental manner, conforming to the scheme presented by Vogel et al. in [415, 417, 418] for monitoring—see Section 4.3. In this section, we showed that the Analyze and Plan phases of VENUS require incremental computational efforts. Overall, we can conclude that VENUS, realized in a MAPE-K feedabck loop, can operate in a highly efficient, incremental manner as it requires $O(\Delta + \Delta')$ steps for Analyze and Plan with Δ the number of the changes of the RTM and Δ' the number of the unprocessed issues. The Analyze and Plan activities are considered the bottleneck operations of the MAPE-K loop in terms if performance.

4.4.3 *Optimality of a Single MAPE-K Run with* VENUS

In this section, given the assumptions made through out the chapter and an appropriate selection of k, we discuss why VENUS guarantees an optimal adaptation behavior concerning utility and reward in a single MAPE-K run—see Section 4.4.1 for a discussion on appropriate selection of k.

Executing all k selected matches for the best rules in annotations.bestRules() guarantees a maximal increase of the utility because it removes all matches of the negative patterns—see assumption (A2)—and does not affect any other matches for any other negative patterns—see assumptions (A3) and (A4). Thus, the final utility after executing all the selected rules is maximal. This effect on the overall utility remains in the system as long as the system is operating. Any other selection of rules that would lead to a lower utility results in a lower reward, i.e., accumulated utility over time, even though yielding lower costs. Thus, pursuing a faster adaptation scheme at the cost of lower utility, i.e., requiring shorter execution time, results in a lower final utility and does not pay off as the the system continues operating at a lower level of utility (compared to the possible maximal utility). Such a system, however, will be equally affected by any future issues, that is, will lose equal amount of utility, compared to an identical system pursuing an optimal adaptation scheme that operates at a higher utility level.

Furthermore, the ordering of the k adaptation rules in annotations.bestRules in VENUS ensures, for the considered time window when the rules are executed, that the resulting reward is maximal. The reason is that any reordering of two rule matches with different ratios results in prioritizing rules with smaller gradients, i.e., smaller area under the curve in Figure 4.3, which indicates lower reward. Similarly, more complex reordering can be achieved by iteratively exchanging two rule matches, that however, would eventually also lead to a lower reward. Thus, the correctly ordered sequence of rules, i.e., in descending order with respect to their impact on utility, results in the maximal reward.

In cases where k is larger or equal to the number of identified issues before each MAPE-K run and all of these issues can be resolved within the time window of a single MAPE-K run, executing the k rules in the given order results in the maximal reward. In this case, all the identified issues can be resolved during one run of the feedback loop. If k is smaller than the number of the detected issues, the resulting reward of executing the chosen sequence of the adaptation rule matches is still optimal given the rationale for defining k—see Section 4.4.1. This would avoid blocking the execution of rules achieving a high impact on the utility by rules that achieve lower increases but are already scheduled to be executed during the current MAPE-K run.

As discussed earlier in Section 4.2, VENUS seeks optimality with respect to system utility and reward via pursuing a greedy algorithm before each adaptation. A greedy algorithm always makes the choice that produces the largest immediate gain. Consequently, the optimality claim in VENUS is restricted to the class of self-adaptation problems that satisfy the *greedy choice property* [see 121] that indicates a globally optimal solution can be arrived at by making a locally optimal, i.e., greedy, choice [6]. Such problems exhibit the *optimal sub-structure property*; a problem has an optimal sub-structure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems—see [438].

4.4.4 *Discussion of Assumptions*

In the following, we review the assumptions we made through out this chapter, summarized in Table 4.1, and discuss their justifications as well as the consequences on the validity and applicability of VENUS if the assumptions do not hold.

A violation of assumption (A1) indicates that the adaptation rules can be applied even if there is no match for a negative pattern. This can be generally ruled out for self-healing systems where only repair rules that resolve occurrences of negative patterns are relevant. However, it might be an issue for self-optimizing systems if adaptation rules are to be continuously applied to improve the utility while a notion of negative patterns might not exist. VENUS demonstrates both self-healing and self-optimization properties; however, in the context of self-optimization, we are only concerned with performance degradation issues which VENUS captures via negative architectural utility patterns. Thus, similar to self-healing, the self-optimizing property of VENUS is realized via negative patterns, hence, assumption (A1) renders valid.

Assumption (A2) states that any adaptation rule, if applied, is effective and therefore resolves the corresponding issue. A violation of (A2) implies that adaptation rules are not always effective, that is, applying a rule does not always resolve the issue. In the context of VENUS, we assume a deterministic behavior for the adaptation rules that rules out violations of (A2); however, there might be cases in which rules might not succeed in resolving the issues. We will therefore investigate such cases, i.e., probabilistic adaptation rules, by considering a likelihood for the success of each rule application during evaluation of VENUS in Section 7.5.

Assumption (A3) can be discussed in two parts; adaptation rules that cause new issues are not reasonable, thus we can safely accept assumption (A3a). We suggest designing the rules in such a way that they immediately resolve all the additional issues they might cause. In VENUS, if a rule accidentally causes a new match of a negative

Table 4.1: List of assumptions for applicability of VENUS.

(A1)	For any in-place model transformation rule $r_j = (\mathrm{LHS}, \mathrm{RHS})$ in the adaptation rule set $\mathfrak{R}$, where LHS of the rule is captured by pattern P_j, a negative pattern P_i^- exists such that any match m_j for P_j includes a match m_i for P_i^-.
(A2)	For any rule $r_j = (\mathrm{LHS}, \mathrm{RHS})$ in the rule set $\mathfrak{R}$, where LHS is captured by P_j, and any match m_j for P_j with the included match m_i for the related negative pattern P_i^-, applying r_j for m_j will make the match m_i for P_i^- invalid.
(A3)	Applying r_j does not result in any new match or removed matches besides m_i for any negative pattern.
(A3a)	Applying r_j does not result in any new match for any negative pattern.
(A3b)	Applying r_j does not result in any removed matches besides m_j for any negative pattern.
(A4)	Applying r_j does not affect any utility sub-function for any match m_k for another negative pattern P_k^-, then applying a rule r_j for a match m_j does not affect the impact on the utility for any other rule r_k and match m_k.
(A5)	All the potential positive patterns are completely within the scope of the application condition and side effect of r_j and do not match only partially.
(A6)	The links for associations that have to be traversed for local search of matches have always small constant upper bounds.

pattern, which will trigger another rule (violation of (A3a)), the new match will be detected and resolved in the next feedback loop run.

A violation of assumption (A3b) results in a case where applying a rule impacts the applicability of other rules. An example of such a violation is applying a rule that replaces a faulty component, which makes the repair rule of the related faulty connectors inapplicable, for instance, because the new version of the component needs different types of connectors and thus a different rule to reestablish the connectors. In this case, the issue of the faulty component overlaps with the issue of the faulty connectors; the issue of the faulty connectors will not be resolved in the current but in the subsequent feedback loop run if it can be matched by a negative pattern. However, we suggest designing the rules in a way that avoids such unwanted dependencies between them, one way is to define the scopes of the rules such that each issue type is completely treated by one rule and the scopes do not overlap. This requires that the scopes of different types of issues do not overlap, otherwise, overlapping issue types should be combined to one type. For instance, such a design results in defining the scope for the rule replacing a faulty component to also cover the relevant faulty connector(s); meanwhile, the scope for the connector repair rule would be restricted to faulty connectors that are not associated to faulty components.

Assumption (A4) excludes cases where executing a rule affects the impact of other rule executions on the utility. A violation of (A4) implies dependencies between the rules similar to a violation of (A3). Again, we suggest designing the rules in a way that avoids such unwanted dependencies. If the adaptation rules influence each other regarding the impact on the utility (violation of (A4)), Venus would not necessarily find the optimal rule for each issue and consequently, the optimal ordering of all rules, since it does not take such dependencies into account. However, all issues would still be resolved although not necessarily with the *best* rules.

Assumption (A5) indicates that executing a repair rule achieves the intended improvement of the utility. If it does not hold, then the utility function or the context to calculate the impact of the rules on the utility are not appropriate. This requires a more expressive utility function or a larger context—see Table 3.2 for examples of utility functions with different context and expressiveness. While it can be challenging to define an appropriate utility function [see 175], establishing a larger context is in principle always possible by splitting such rules into multiple, more specialized rules that have a larger context to achieve the overlap with the positive patterns.

Assumption (A6) comprises the constant upper bound of the size of the patterns and the local search for matches of patterns/rules. In the context of Venus, this is justified based on the simple nature of the patterns that we encountered and given that the size of the patterns is limited by the size of the system architecture. Nevertheless, if the assumption does not hold, also other schemes such as pure rule-based solutions might also yield high execution costs.

To summarize, the assumptions listed in Table 4.1 are usually justified for rule-based self-healing approaches and, for the limited scope of rule-based self-optimizing systems, because designing rules that are not triggered by any issue or that do not resolve any issue is not justified (see (A1) and (A2)); rules that cause new issues are a design bug and could thus be excluded (see (A3a)); rules that affect other issues or other rules are not useful and should thus be avoided (see (A3b) and (A4)); rules that do not completely cover the positive patterns should be replaced by alternatives that offer a full coverage for the resulting positives patterns (see (A5)); and finally, rules and patterns that do not allow a local search affect the incrementality of the approach, are not usual, and can be avoided (see (A6)).

4.5 SUMMARY

Having defined utility functions for dynamic software architectures in Chapter 3, as a second step towards addressing the quality-cost trade-off in software self-adaptation, we presented in this chapter an incremental solution for self-adaptation via introducing Venus. The scheme tailors the ECA rule- and utility-based formalisms to assemble, at design-time, a combination solution to engineering self-adaptation of large, dynamic architectures. We integrated Venus in a MAPE-K feedback loop that leverages event-based processing of the change events as well as the incremental utility-change calculation to support incremental adaptation.

We position Venus as a reactive solution to self-adaptation that targets a class of software self-adaptation problems that are usually identified by symptoms such as self-healing systems. Enabled by model-driven principles, we described our approach for implementing the Analyze and Plan activities in Venus; we discussed how Venus leverages the outcome of the *Utility Function* module during Analyze and Plan—see Figure 4.1 for an overview—and elaborated the mapping of the utility values to the adaption rule applications to carry out a rule-based, utility-driven adaptation. We discussed the execution complexity and optimality of VENUS in terms of utility and reward of the adaptation; we argued that for a class of self-adaption problems with the greedy choice property, VENUS can provide optimal solutions in terms of utility and reward, while owing to its incremental execution, demonstrates properties of its ECA rule-based constituent in terms of efficient runtime processing of the changes and adaptation scalability. Finally, we provided a list of assumptions that are required for applicability of VENUS and validity of the claims and discussed why they are justified.

5 HYPEZON: HYBRID SELF-ADAPTATION WITH RECEDING HORIZON CONTROL

When the complexity of the adaptation space exceeds the ability of the individual adaptation solutions, to solely, fulfill both the quality and cost requirements of the adaptation, the quality-cost trade-off can be addressed via customized solutions that combine multiple individual policies to construct a compromise adaptation mechanism that can cope with the growing complexity of the software and its input space. In Chapter 4, we introduced Venus, a solution for architecture-based self-adaptation that combines the rule-based and utility-based policy formalisms in its design. Developing customized solutions like Venus, while potentially beneficial at runtime, imposes a design-time development effort that might be unproportionate to the complexity of the problem. In short, such approaches might provide over-engineered solutions to a problem that a simpler process, in terms of development-effort, is adequate. We propose to address this problem via a systematic *coordination* of multiple off-the-shelf policies. The proposed solution supports cost-effective achievement of system quality objectives while, compared to solutions such as Venus, reduces the development cost because it precludes the effortful process of developing new algorithms/heuristics from scratch.

In this chapter, as a complementary solution to Venus , we propose HypeZon, a coordinating HYbrid planneEr for self-adaptive software employing receding horiZon—see Section 2.3.3. HypeZon implements the Plan activity of our solution for incremental architecture-based self-adaptation—see Figure 5.1 for a schematic overview. We showed in Chapter 3 how to construct the *Utility Function* building block; in Chapter 4, we discussed design details of an adaptation engine, amenable to incremental execution, where the Analyze and Plan activities were implemented by Venus. HypeZon builds on the Monitor, Analyze, and Execute activities of the incremental adaptation engine, presented Section 4.3, and offers an alternative solution for the Plan activity. The scheme relies on coordination of multiple, existing policies and leverages control-theoretic principals to systematically construct a generic hybrid planning mechanism. The scheme, at runtime, coordinates multiple off-the-shelf adaptation policies that individually cannot fulfill both the quality and cost requirements of the adaptation for a potential range of the system input space. HypeZon is generic as it is agnostic to the inner workings of the individual adaptation policies and hols a black-box view.

In Figure 5.2, we repeat the notional representation of the adaptation space with growing complexity from Figure 1.1. Adaptable software with dynamic architectures are prone to evolution during system execution, thus adaptation complexity, e.g., size of the system architecture or number of the adaptation issues affecting the system, is subject to change. The change might instantly render a well-performing solution insufficient—see X_2 in Figure 5.2 where a slight change in the complexity of the adaptation space disqualifies the, until then, best-performing optimization-based solution in comparison to the alternative rule-based solution. The parts of the x-axis before X_1 represent adaptation complexities where an individual solution, i.e., the optimization-based solution, per-

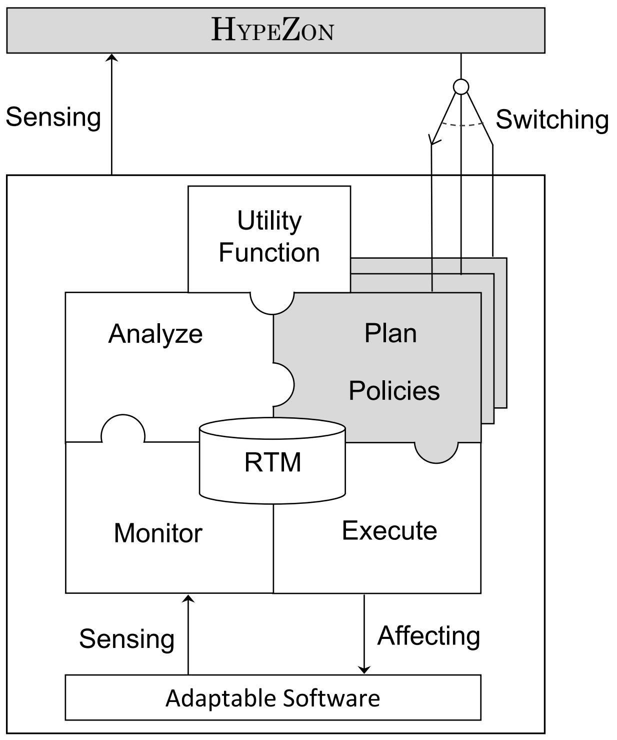


Figure 5.1: Chapter overview: plan with HypeZon in adaptation engine.

forms desirably (obtains Q*). X_1 marks the point in the adaptation space where the individual solutions cannot desirably satisfy the adaptation objectives (below Q*). X_2 represents a point where the complexity of the adaptation space hinders the optimization-based solution to outperform the rule-based alternative. A hybrid, coordination-based solution composes the two policies, i.e., rule-based and optimization-based, at runtime, to balance the quality-cost trade-off—see the switch at X_2. However, the hybrid solution is bounded to the performance margins of its constituents. A customized, combination-based solution, e.g., Venus, may combine multiple policy formalisms in its design to construct a new customized policy that leverages the strength of its constituents, thus outperforms them in terms of cost minimization and quality maximization.

Capturing the complexity of the multi-dimensional adaptation space as points across the x-axis is a an oversimplification of the phenomenon. Note that Figure 5.2 is used only for conceptual illustration purposes and does not correspond to any empirical

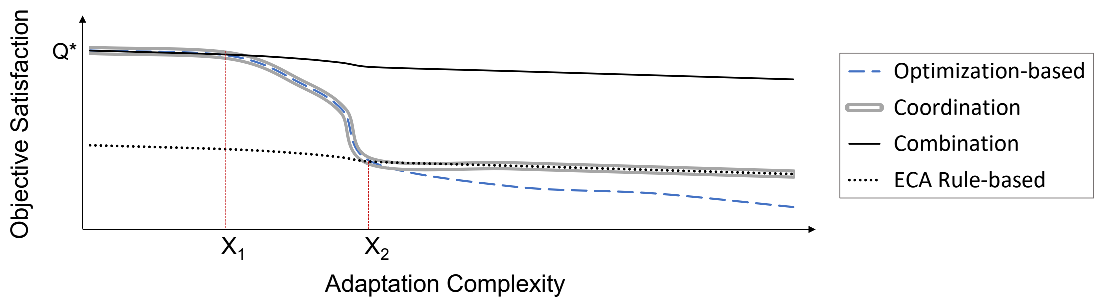


Figure 5.2: A notional representation of a space for self-adaptation solutions in domains with growing complexity (Q* denotes optimal objective satisfaction).

measurements. We replace this chart by an alternative[1] that is based on quantitative experiments in Chapter 9 where we report on a series of experiments during which we maintain all the dimensions of the adaptation space constant and emulate the increasing complexity of the adaptation by step-wise increasing the number of the adaptation issues. This chapter introduces HypeZon that implements the coordination-based solution in Figure 5.2.

Conforming to the well-established practice of explicitly capturing the control loops in the architecture of the system [see 208, 367], HypeZon adopts the hierarchical arrangement of the control loops from adaptive control in its design—see Section 2.1.1. The scheme is concerned with the Plan activity of the MAPE-K loop as a controller that, inspired by the MPC, leverages *receding horizon* to utilize runtime information and adjusts its control parameters at runtime—see Section 2.1.2. The control design and the architecture of HypeZon build on control theory; however, compared to the adaptive control that restricts the scope of the controllers to calculating set-points and prescribing required changes in the input parameters, HypeZon extends the involvement scope of the higher-level control loop in the lower-level loop.

HypeZon enables the self-adaptive software to observe its own behavior jointly with the one of the feedback control loop in terms of objective satisfaction, reason about changing the trade-offs during its lifetime, and explore compositions of multiple off-the-shelf adaptation policies, e.g., rule-based, goal-based, or utility-based policies, at runtime. To engineer HypeZon for self-adaptive software, we propose two alternative designs that conform to meta-self-aware architectures: *external* and *internal*. The designs build on the framework for realizing meta-self-awareness in the architecture from earlier work by Giese et al. [187].

The main focus in providing hybrid solutions for self-adaptive software so far has been set on developing individual adaptation policies such that, in coordination together, they cover a large spectrum of the solution space for self-adaptive software—see [4, 28, 225, 299, 398, 399]. In the context of HypeZon, we focus on the coordination mechanism, i.e., the realization of the decision-making mechanism within the hybrid planner.

In Section 5.1 we present a motivating example for application of the hybrid planning with mRUBiS. Section 5.2 presents computational aspects of HypeZon and the architectural design of HypeZon is presented in Section 5.3. Finally, in Section 5.4, we summarize the chapter and discuss how HypeZon fulfills the requirements for architecture-based self-adaptation.

5.1 MOTIVATING EXAMPLE

This section presents an exemplary scenario for mRUBiS. As discussed in Section 2.5, mRUBiS is a web-based client-server system conforming to a component-based architecture style and serves as an online marketplace that hosts arbitrary number of shops. Each shop, i.e., tenant, includes 18 components each providing a different type of service within a shop. Clients send bid and buy requests to the shops of interest, the requests are further transferred to the relevant services withing the shop, e.g., query management service or authentication service.

1 see Figure 9.4.

Two major types of adaptation issues affect mRUBiS: Critical Failures (CF) that affect the availability of services and Performance Issues (PI) that affect performance of the services as a result of changes in component's load. The system's workload depends on the request arrival rate, which is uncertain as it depends on the external demand. mRUBiS needs to optimize profit, i.e., maximizing revenue and minimizing operating costs, via adaptation. As a market place, the business objective of mRUBiS is to provide for *high availability* and *low response time*—see goal model of mRUBiS in Figure 3.6. To maximize the revenue, it is desirable to (i) maintain the components available by prompt resolution of CFs and (ii) maintain the request response times below an acceptable threshold, e.g., based on the SLA. Typically, an increase in the request arrival rate leads to increased load on the services in the shops and causing the performance to drop. In such situations, the system can add more replicas to the heavily loaded services within a shop, using adaptation rule addReplica<type>, to handle the increased workload, but also increasing the operating cost. To reduce costs, the system has an adaptation rule, i.e., removeReplica<type>, to remove under-utilized service replicas when the main service is experiencing relatively moderate load. Thus, during traffic surges, addReplica<type> adds more replicas to services of the shop to cope with the increasing demand and removeReplica<type> reduces the operation cost of the shops by removing underutilized replicas. Moreover, the adaptation options applicable to resolve the CFs impose different execution cost to the system, e.g., it is more expensive to replace a crashed component than to restart it—see Section 2.5.

There is a penalty for violations of system goals, i.e., if the shops in mRUBiS do not have request response times below the promised threshold or their services are not continuously available. Therefore, in case of a high response time, the system needs to react quickly by adding replicas. Service crashes should also be handled as promptly as possible. However, once response time is within the acceptable margins, the system should execute self-optimizing adaptations to bring down the operating cost, execute more robust, i.e., possibly time consuming, repair of the malfunctioning components to maximize the long-term reward.

mRUBiS aims to maximize its revenue via increasing service performance and keeping the response time below the threshold. Moreover, efforts towards minimizing its operation cost include reducing the number of the active replicas and considering low-cost repair actions for the CFs. These objectives are collectively captured in a multidimensional utility function and a cost function—see Section 3.2. For this example, we define the utility of each component in mRUBiS, in current state s, as $U_c(s) = reliability \times criticality \times connectivity \times Performance$, i.e., the Saturating function in Table 3.2. The performance of a service in mRUBiS is defined according to Equation 3.9. The overall utility of mRUBiS is defined based on the utility of the tenants, i.e., shops, as Equation 3.2. The utility of a tenant, in current state s, is then calculated as Equation 5.1. The goal of mRUBiS is to maximize the ratio of the utility to cost—see Section 4.3.4.

$$U(tenant) = \sum_{i=1}^{18} U_c(s) \tag{5.1}$$

The combination of the multidimensional utility function and cost function capture conflicting requirements such as lowering response time, i.e., increasing performance, increasing revenue, and reducing operating cost. The combination is concerned with both the quality and the timeliness of the adaptations. The utility function captures quality since it has various adaptation goals as its constituents. Timeliness is captured since there is a penalty, i.e., utility drop, for response time above the threshold that cause a performance drop in the utility formula. In such a situation the system needs to react quickly to mitigate the negative impacts on the utility.

The trade-off between the adaptation quality, e.g., robust repair of crashed components or adding replicas to the server pool, and minimizing the operation cost, e.g., lightweight, fast repair of crashed components, needs to be addressed during execution of mRUBiS. As briefly discussed in Chapter 1, reactive ECA rule-based approaches for self-adaptation tend to deliver prompt solutions for adaptation but they often only provide sub-optimal solutions in terms of system utility as they are bound to design-time assumptions. For instance, a reactive solution such as a rule-based approach might quickly provide a plan to a response time constraint violation, and thus improve the utility in the short-term; however, the plan is likely to be sub-optimal due to uncertainty in the request arrival rate, which is difficult to predict at design time, i.e., when formulating the rule. Conversely, optimization-based mechanisms for self-adaptation preclude design-time assignments of actions to conditions (events), instead, they often perform an exhaustive search in the possible adaptation space. These solutions solve a cost-intensive optimization problem before each adaptation which may render attaining optimal adaptation plans time-intensive [361] which would be an issue in mRUBiS, particularly for situations such as a response time constraint violation.

For systems such as mRUBiS that interact with an input space with volatile characteristics, using an adaptation mechanism with a single planner, i.e., either a deterministic, reactive policy or an optimization-based, proactive policy, fails to meet the contradicting quality and cost (time) objectives. In this context, a hybrid adaptation solution composes multiple off-the-shelf solutions at runtime, e.g., employing a reactive ECA rule-based solution to provide timely adaptation plans while an optimization-based solution searches the solution space for high-quality adaptation plans. For instance, a hybrid solution for adaptation of mRUBiS can be instantiated using a rule-based and an optimization-based approach; the rule-based approach can provide a timely response in time-sensitive situations, e.g., response time above the threshold or surges of component crashes, whereas the optimization-based may be employed to provide higher-quality plans, thus balance the quality and cost(timeliness) of the adaptation.

5.2 HYBRID PLANNING FOR SELF-ADAPTATION WITH HYPEZON

In this section, we present the computational aspects of HypeZon. The scheme complies with the definition of coordinated hybrid planner provided by Trollmann et al. in [405] and introduced in Section 2.3.3. HypeZon is embedded in the incremental adaptation engine introduced in Chapter 4. The scheme builds on the Monitor, Analyze, and Execute activities of the adaptation engine presented in Section 4.3 and offers an alternative solution for the Plan activity. Therefore, incremental calculation of the utility-changes as the impact of the adaptation rule applications are available to HypeZon—see utility-change calculation in Venus as Equation 4.1.

5.2.1 *Hybrid Planning: Preliminary Definitions*

A hybrid solution for planning adaptation in general is concerned with two aspects of the available policies: quality and runtime cost, e.g., timeliness. A policy is generally defined as a set of control decisions that map states to actions [361]. An adaptation policy π represents an encapsulation of the system's adaptive behavior governing the choice of adaptation actions when applicable. For each state s, $\pi(s)$ indicates the adaptation action a to be executed, i.e., $\pi(s) = a$. We use the term policy to refer to the mechanism that governs the decision-making process in the plan activity and is used to express and operationalize the high-level objectives of the system [214]—see Section 2.3.2.

The set of adaptation issues that affect the self-adaptive software in a time-correlated manner, i.e., they are detected and considered by one adaptation loop, constitute a *planning problem*. For a planning problem, a *plan* is an *ordered* list of adaptation actions such that each action resolves at least one of the adaptation issues in the planning problem. The actions are ordered based on the corresponding adaptation issues, e.g., more critical issues are prioritized.

The *look-ahead horizon* $\mathcal{L}$ is the steps into the future, in terms of system changes, that are considered during planning. A *planning horizon* Φ is a prefix of the look-ahead horizon that is planned for—see Figure 5.3. A planning horizon of size $|\Phi| \leqslant |\mathcal{L}|$ with $|\mathcal{L}|$ the size of the look-ahead horizon, only plans for $|\Phi|$ out of $|\mathcal{L}|$ adaptation issues in the look-ahead horizon. An infinite planning horizon allows for considering the entire look-ahead horizon for planning.

Let $\pi_i(\Phi)$ be the plan generated by policy π_i for a planning problem captured in Φ. For a given planning problem in Φ, the quality of π_i is quantified as the utility of the plan by π_i, that is, $U(\pi_i(\Phi))$. The *utility of plan*, $U(\pi(\Phi))$, is the expected *improvement* of the overall system utility upon execution of the adaptation actions that constitute the plan. $U(s)$ represents the utility of the state s, i.e., a scalar value as a quality metric that identifies the degree to which system goals and requirements are satisfied in state s—see Section 2.2. When the utility $U(\pi(\Phi))$ cannot be accurately measured, an estimations of the value, $\hat{U}(\pi(\Phi))$, is considered. The accumulated utility over time is the reward. The cost of π_i for providing a plan to address the issues in Φ is depicted as $C(\pi_i(\Phi))$. Given a planning problem, the policy cost may be captured as the time required by the policy to provide a plan. Therefore, $C(\pi_i(\Phi))$ can be provided by the developers for different $|\Phi|$, e.g., as a lookup table. When the exact value for the cost $C(\pi_i(\Phi))$ cannot be accurately measured, an estimations of the cost, i.e., $\hat{C}(\pi_i(\Phi))$, is considered. Estimations of the policy cost can be obtained via theoretical modeling such as employing worst-case time models [327] or based on empirical profiling [389].

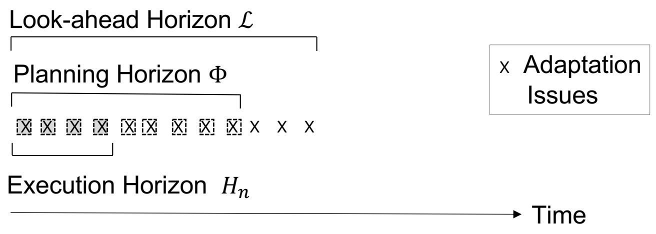


Figure 5.3: Look-ahead, planning, and execution horizon in HYPEZON.

5.2.2 *Receding Horizon in* HypeZon

HypeZon addresses the planning problem for software self-adaptation by coordinating multiple off-the-shelf adaptation policies, at runtime, and selecting based on the cost (time) and quality objectives. The scheme implements the Plan activity of a MAPE-K loop that is realized by the incremental adaptation engine discussed in Section 4.3. During an adaptation cycle, once the Analyze activity has detected the need to plan an adaptation, i.e., there exists a planning problem captured by Φ, HypeZon takes the runtime conditions into account and decides which of the available adaptation policies best suits the said conditions while balancing the quality and cost trade-off.

Inspired by MPC, HypeZon implements the Plan activity using a receding horizon. In Section 2.1.2, we introduced details of a model predictive controller—see Figure 2.3 for a block diagram. At each sampling interval $\mathcal{I}$, HypeZon makes new measurements, and based on the operation conditions, decides if a *policy switch* is required. As a result, the running adaptation policy that steers the decision-making during the Plan activity is replaced by a better fitting alternative. A sampling interval is the frequency of data collection.

The process of resolving a set of adaptation issues in the planning horizon via selecting a set of adaptation actions that maximize an objective function in HypeZon is formulated similar to MPC; in a model predictive controller, the control calculations are based on current measurements as well as the predictions of the future values of the system outputs; the objective of the control calculation is to determine a sequence of M control actions so that the predicted output moves towards the set-points in an optimal manner. Control calculations determine a sequence of M control actions, i.e., *control horizon*, such that the *predicted output* moves towards the set-points over a finite *prediction horizon* P—see Section 2.1.2. HypeZon implements a hybrid planner whereby the look-ahead and planning horizons of the planner provide similar functionality as the prediction and control horizons in a typical MPC controller. Given a planning problem, the objective of HypeZon during planning is to select a sequence of adaptation rules that upon execution, adapt the system towards the desired configuration, while obtaining highest possible reward, i.e., utility over time. HypeZon borrows the notion of MPC receding horizon of size *one* and extends it to an *execution horizon* H_n with adjustable size n. Figure 5.3 shows an example of a look-ahead horizon, planning horizon, and execution horizon in HypeZon. As explained in Section 2.1.2, in MPC, the control horizon is a list of *all* the actions that are planned during one interval; this is captured as planning horizon Φ in HypeZon.

As stated by Pandey et al. [328], it is difficult to verify the compatibility between the plans of different policies, how to choose the planning horizon, and when to stop using one plan and switch to another policy for planning. In the following, we describe how HypeZon addresses these challenges. The scheme uses runtime information such as the planning cost (time) of the adaptation policies, system load, number and type of the adaptation issues, and the cost of switching between the policies to decide on the size of the planning and execution horizons.

Employing planning and execution horizons with adjustable size provides for runtime flexibility. Before choosing a policy, HypeZon adjusts the size of the planning horizon Φ with respect to the policy cost estimations $\hat{C}(\pi(\Phi))$. For example, if large number of components in mRUBiS are affected by critical failures (CF) and, at the same time, certain tenants are experiencing performance issues causing latency in request response times, HypeZon may reduce the size of the planning horizon Φ to first consider the more critical issues, i.e., component crashes such that the corresponding services become available as soon as possible. This way, HypeZon reduces the expected planning overhead by reducing the size of Φ, thus the critical issues are resolved relatively faster.

An execution horizon H_n only considers the first n adaptation actions in the planning horizon for execution in the current adaptation cycle. After executing the n adaptation actions, HypeZon stops the execution of the plan and the remaining unresolved issues are considered together with the newly observed issues in a subsequent adaptation cycle as a new planning problem. Employing execution horizon of small size in HypeZon results in utilizing the most recent adaptation issues immediately. In contrast, large execution horizons ignore the recent observations until all the actions in the planning horizon are executed—see Figure 5.3. Small sizes for H_n demand more frequent planning. Moreover, the execution of the actions that are in the planning horizon and not in the execution horizon is postponed to the subsequent adaptation cycle(s). In cases where the planning in a policy is time-intensive, frequent planning might affect the adaptation time negatively. When HypeZon switches the adaptation policy, the employed policy plans for the remaining adaptation issues whose corresponding adaptation actions in the planning horizon were not included in the execution horizon. Moreover, the employed policy also considers the newly detected adaptation issues. Consequently, after each policy switch, the planning problem is considered anew during the control calculation in HypeZon. This way, HypeZon guarantees that after a policy switch, the active policy calculates the plan according to the most recently observed conditions while taking into account the already existing issues.

In order to guarantee the compatibility between a plan and the planning problem, HypeZon only executes one policy at a time and avoids concurrent executions of multiple policies. As a result of the sequential execution of the policies, the provided plans are always compatible with the planning problem because the planning problem that is assigned to a policy remains unchanged during the planning time. This way, once the plan is ready, HypeZon does not check if the plan is still applicable to the current planning problem. This feature in HypeZon avoids the runtime overhead that is caused by compatibility analysis between the planners; however, concurrent executions of policies may reduce the time that the hybrid planner has to wait until a plan is ready.

5.3 HYPEZON: A CASE FOR META-SELF-AWARENESS

The black-box-oriented scheme of the control theory extends towards adaptive control, where the controller may change its own control regime [280]. This requires the controllers to have adjustable parameters—see Section 2.1.1. Moreover, an adjustment mechanism needs to be in place to oversee the parameter tuning via layered arrangements of control loops, where the lower-level entities are controlled by the immediate higher-level loop [113].

In the realm of self-adaptive systems, adaptive control is perceived as reasoning about the adaptation logic [332]. The reasoning requires observing the behavior of the control loop in terms of effectiveness and performance, realizing the need for change, and prescribing the necessary decisions to steer the controller towards the desired behavior. *Meta-self-awareness* captures the requirements for equipping self-aware systems with advanced self-reflective properties [271]. As a result, systems with meta-awareness proper-

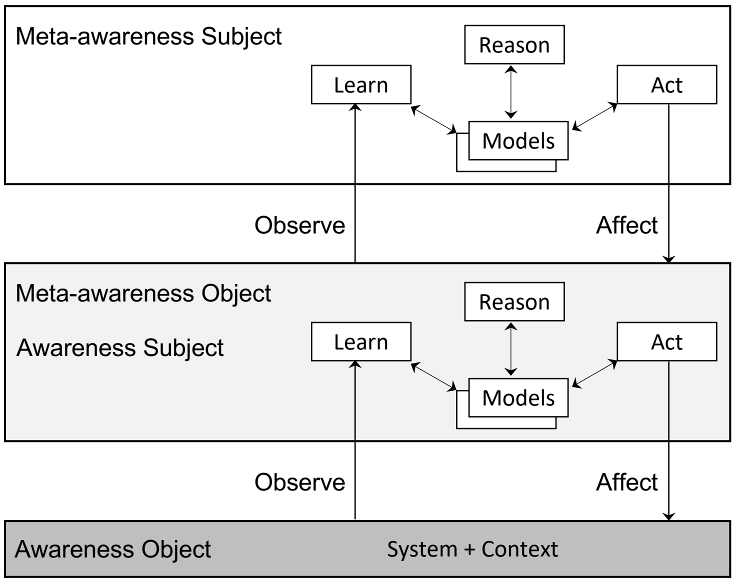


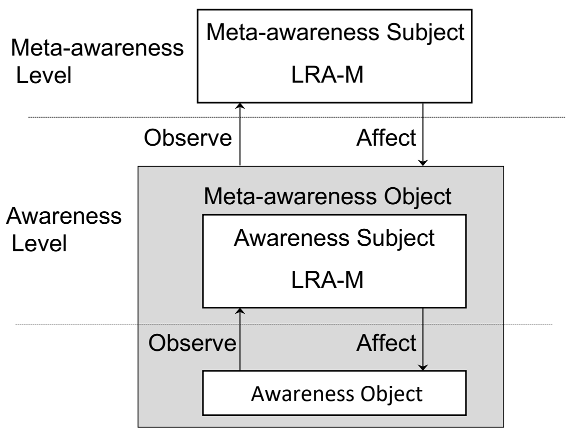
Figure 5.4: Awareness levels and scopes.

ties can reason about changing trade-offs during their lifetime [284]. The control design and architecture of a meta-self-aware system builds on Control Theory as a prominent base, however, it extends the involvement *scope* of the higher-level control loops in the lower-level entities.

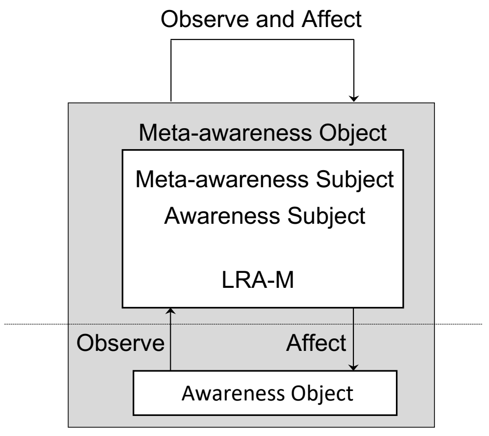
A hybrid planner with adjustable control parameters requires a decision-making mechanism that utilizes system-level as well as feedback control-level information at runtime. In this section, we argue that equipping a self-adaptive software with hybrid planning should be realized as a meta-self-awareness property. To this end, after presenting brief background on self-awareness and meta-self-awareness in Section 5.3.1, building on previous study by Giese et al. [187] towards making meta-self-awareness visible in the architecture, first we propose two designs to engineer a self-adaptive software with meta-self-awareness properties in Section 5.3.2. In Section 5.3.3, we show how HypeZon can be realized by the two designs and present detailed algorithms to instantiate HypeZon.

5.3.1 *Self-awareness and Meta-self-awareness*

A self-aware system is identified by two main characteristics; first, the ability to *learn* models capturing knowledge of the system, its context, and its goals on an ongoing basis. Second, *reasoning* about using the models for analysis and planning concerns. Computational self-awareness is achieved via a *Model-based Learning, Reasoning, and Acting* (LRA-M) loop. There can be multiple variations to *Acting* e.g., explaining, reporting, suggesting, and adapting. Self-adaptation, realized via a MAPE-K feedback loop, is one of the advanced characteristic of self-aware systems where the *scope* of *Acting* is set to *Adapting* [284]. In this chapter, we target self-aware systems with adaptation capabilities that conform to an LRA-M architecture—see lower-level loop in Figure 5.4. The *learning* process uses system *observations* to learn *models* that are then used for *reasoning* and *adapting*. As a result, a system realizing the LRA-M loop becomes aware of itself and its context. The *object* of the awareness is the entity being reasoned upon, i.e., the



(a) External design.



(b) Internal design.

Figure 5.5: External and Internal designs for meta-self-awareness.

system and its relevant context in Figure 5.4. The *subject* of the awareness is the entity performing the reasoning.

A meta-self-aware system can obtain knowledge of its own awareness and how it is exercised. A higher-level self-aware entity such as an LRA-M control loop—see higher-level loop in Figure 5.4—*reflects* on the benefits and costs of maintaining increased awareness as well as the capacities for it. Meta-self-awareness is concerned with two classes of objects; The elements of the lower-level awareness loop, e.g., a learning process or an adaptation logic, and the output of these elements, i.e., the models or specifications produced by the object being reasoned upon [284]. In order to explicitly capture the meta-self-awareness properties in the architectural design, meta-self-awareness can be realized either as a built-in capability or as an external meta-awareness layer [187]—similar to the internal and external approaches for engineering systems with self-adaptation properties [92].

5.3.2 *Meta-self-aware Designs to Realize Hybrid Planning*

In the proposed designs[2], the self-awareness capabilities are realized via the LRA-M loop introduced in Section 5.3.1. The model in the LRA-M loop overlaps with the notion of an RTM—see Section 2.4—insofar it is evolving at runtime to reflect the current state of the system and its context via being causally connected [47]. Moreover, the model may include knowledge about system's goals and requirements.

5.3.2.1 *External design*

In order to explicitly separate the awareness and the meta-awareness levels, as depicted in Figure 5.5a, two hierarchically arranged LRA-M loops are employed. The lower-level loop, henceforth the *awareness level*, serves as the awareness subject for the underlying system and its context, i.e., the awareness object—see Figure 5.4. The LRA-M loop at the

2 Giese et al. have previously introduced different architectural concepts for modeling meta-self-aware computing systems in [187] where they also discuss variations of internal and external concepts for realizing meta-self-awareness. In this section, we build on their study and propose more concrete designs in the context of self-adaptive systems and hybrid planning.

higher level, henceforth *meta-awareness level*, observes the awareness level in combination with the system and context, i.e., meta-awareness-object, learns models that capture knowledge about the meta-awareness object, reasons about the meta-awareness object and, if required, adapts it accordingly. The *scope* of *Affect* in the meta-awareness level includes changes to the awareness level and to the underlying system and its context.

The External design supports the explicit separation of concerns, i.e., awareness and meta-awareness, at the architecture level and allows for reusability, easier maintenance, and independent evolution of each level. As a result, separate and independent mechanisms for observing, learning, and reasoning logic may be employed by each level. Realizing meta-awareness at an external level provides for a global view on the meta-awareness object. This allows for observing phenomena with global scope that are not observable at the awareness level. The External design operates the meta-self-awareness subject on a different timescale than the one of the object, thus the lower-level loop is executed more frequently. The meta-awareness loop however, is executed less frequently since it is inherently concerned with relatively more sparse phenomena to react upon [112].

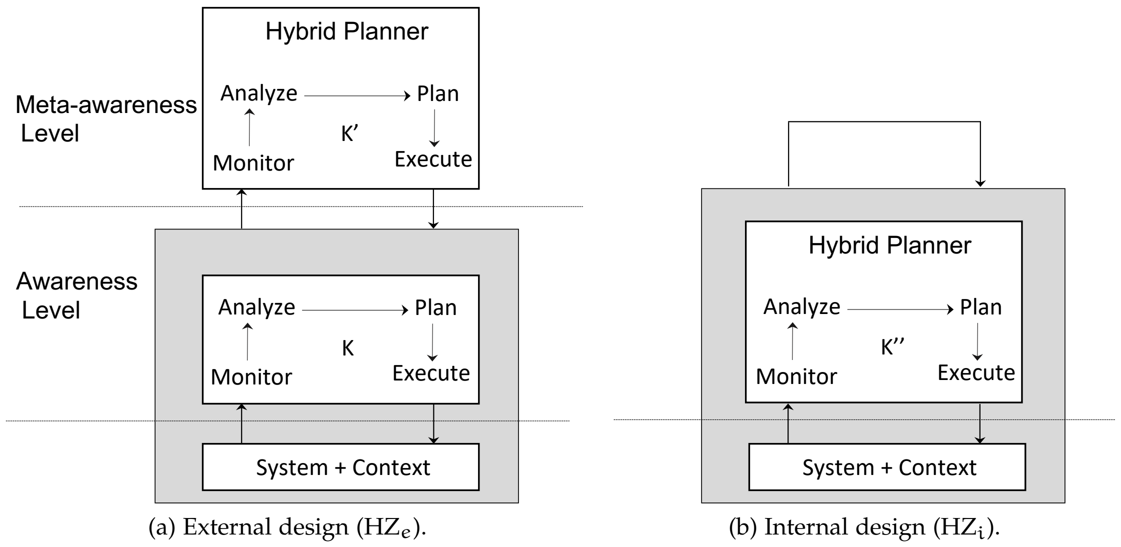
5.3.2.2 *Internal design*

An internal realization of the meta-self-awareness properties is possible via employing an awareness *self-loop* as depicted in Figure 5.5b. In this design, the LRA-M loop *observes* and *affects* itself– known as self-loop [187]. The subject and object of meta-awareness are not architecturally separated, consequently, one element, i.e., the LRA-M loop, performs both the *reasoning* and *being reasoned about* parts of the meta-awareness. This is similar to the *internal approach* for self-adaptation where there is not an explicit MAPE-K loop and one element performs both the adapting and adapted part of the self-adaptation [367]—see Figure 2.5. In the Internal design, the awareness level is also aware of itself. The employed LRA-M loop observes the meta-awareness object—while being part of it—learns the required models from the observations, reasons about the meta-awareness object and adapts it accordingly. The *scope* of the *Affect* for the meta-awareness subject includes changes to itself, the underlying system, and its context. The internal design implements the awareness and the meta-awareness properties in an intertwined manner and keeps the subject of the meta-awareness close to the object. In this design, the meta-awareness self-loop operates at the same timescale as the awareness loop.

5.3.3 HypeZon *as Meta-awareness Subject*

HypeZon is realized as a meta-awareness subject via both External and Internal designs presented in Figure 5.5. The variants are called HZ_e and HZ_i respectively. Figure 5.6 shows how the suggested designs for meta-self-awareness realize hybrid planning at the architecture level. As discussed in Section 5.3.1, self-adaptation, often carried out via a MAPE-K feedback loop, is one of the advanced characteristic of the self-aware systems where the scope of *acting* in the LRA-M loop is set to *adapting*—see Figure 5.4. Both HZ_e and HZ_i variants realize the self-awareness capabilities via a MAPE-K loop.

In the external design, i.e., HZ_e, the loop at the meta-awareness level implements the hybrid planner. The hybrid planner observes the awareness level in combination with the system and context, reasons about them, and plans accordingly. In the context of hybrid planning with HypeZon, the adaptations of the awareness level include



(a) External design (HZ_e). (b) Internal design (HZ_i).

Figure 5.6: Meta-self-aware External and Internal designs realizing hybrid planning.

control parameter tuning and *policy switch*—see Section 5.2. HZ_e operates the hybrid planner in coarser time intervals compared to the lower-level MAPE-K loop. Consequently, compared to HZ_i, the control loop containing the hybrid planner in HZ_e has larger sampling intervals ($\mathcal{I}$)—see Section 5.2. While the sampling interval in control theory refers to the frequency of data collection, in the context of HypeZon, it captures the execution intervals of the hybrid planner insofar the hybrid planner is executed at each sampling interval, hence, it makes new measurements and decides accordingly. The measurements are the accumulated observations during the sampling interval. The lower-level loop in HZ_e is executed more frequently to guarantee timely adaptation concerning the part of the system under its direct control. The meta-awareness loop however, operates at a relatively larger timescale. This is both due to the sparsity of the relevant concerns (compared to adaptation issues that are more frequent than other concerns) and its more coarse grained execution intervals by design.

HZ_i, as depicted in Figure 5.6b, implements the hybrid planner at the awareness level. Therefore, at each cycle of the MAPE-K loop, the hybrid planner is executed as well. The knowledge base of the MAPE loop in this design, K'', contains different information compared to K, K', and, $K + K'$. The reason is that HZ_i implements the adaptation loop and the hybrid planner in an intertwined manner and keeps the subject of the meta-awareness, i.e., the hybrid planner, close to the object. In this design, the meta-awareness self-loop, hence the hybrid planner, operates at the same frequency as the awareness loop.

Figure 5.7 is a high-level flowchart showing the steps of hybrid planning in HypeZon. For a planning problem captured in Φ, in both HypeZon variants, the decisions for tuning the control parameters and the policy switch is made based on the average values over system past executions during the sampling interval $\mathcal{I}$. The observations are shown as *Monitor and Analyze output*. Compared to HZ_i, HZ_e collects accumulated observations of system executions that can be used to estimate the operation conditions during the next interval. HZ_i however, is executed more frequently, that is, at each adaptation cycle, and makes control and switch decisions based on the observations over a relatively shorter period. In the proposed designs, the focus is on the *functional* aspects

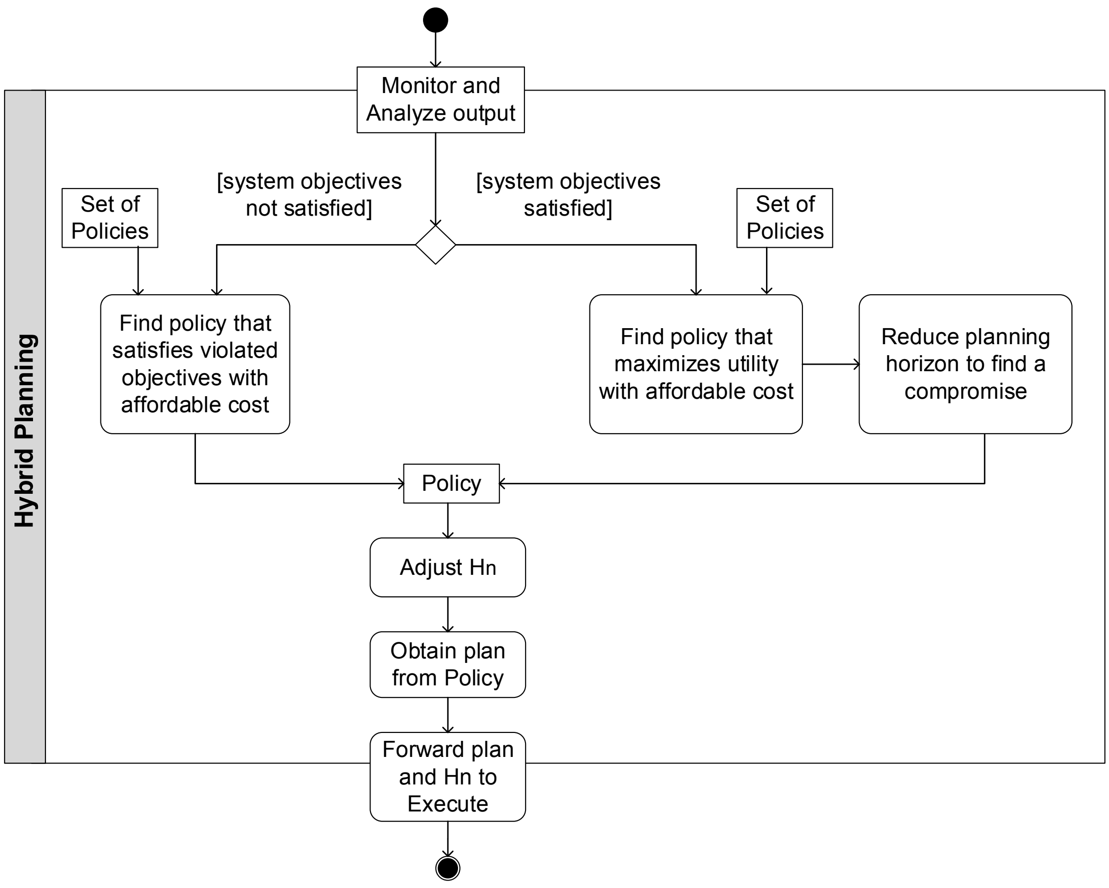

Hybrid Planning
Monitor and Analyze output
[system objectives not satisfied]
[system objectives satisfied]
Set of Policies
Set of Policies
Find policy that satisfies violated objectives with affordable cost
Find policy that maximizes utility with affordable cost
Reduce planning horizon to find a compromise
Policy
Adjust Hn
Obtain plan from Policy
Forward plan and Hn to Execute

Figure 5.7: Generic description of hybrid planning in HypeZon.

enabled by each design rather than the *architectural* aspects. However, the architectural decision of separating the awareness and the meta-awareness loops in HZ_e and combining them in HZ_i steers the decisions that are relevant to the functional aspect, i.e., the sampling (execution) intervals of HypeZon in each design.

Provided the observations in Figure 5.7, HypeZon checks the indicator(s) for system objectives; if the objectives are satisfied, the hybrid planner searches the *Set of Policies* for a policy that maximizes system utility with the hard constraint regarding the budget. The cost of the policy, i.e., $\hat{C}(\pi(\Phi))$, should be within the permitted budget. HypeZon may reduce the size of the planning horizon Φ, thus reducing the cost of planning $\hat{C}(\pi(\Phi))$ to find a compromise policy that maximizes $\hat{U}(\pi(\Phi))$. If one or more system objectives are not satisfied, HypeZon searches for a policy that can address the violated objective with affordable cost. Once the *Policy* is selected, HypeZon tunes the size of the execution horizon H_n according to the expected conditions, e.g., estimated system load and policy cost. Finally, HypeZon obtains the plan from the selected policy and, together with H_n, forwards them for execution. As discussed earlier, HypeZon relies on our incremental Analyze scheme pretested in Chapter 4, therefore, the issues in Φ that constitute the planning problem are in the form of an ordered list with respect to the importance of the issues. Consequently, H_n is also an ordered list. Therefore, any adjustments to the size of the planning and execution horizons removes the elements from the end of the lists, i.e., the least important issues.

ALGORITHM 5.1 : HZ_e / HZ_i

```
1  Require: $\hat{load}$, $\Pi$, $RT_{\mathcal{I}}$, $\hat{RT}$
2  $\Phi \leftarrow \mathcal{L}$          // Planning horizon gets all issues in look-ahead horizon
3  $\pi^* \leftarrow null$
4  if $RT_{\mathcal{I}}$ optimal then                                  // Objective satisfied
5      $\pi^* \leftarrow \pi_i \in \Pi : i = \underset{i}{\operatorname{argmax}} \hat{U}(\pi_i(\Phi))$        // Policy to maximize utility
6      if $(c_{curr,*} + \hat{C}(\pi^*(\Phi)) + \hat{RT})$ high then            // If cost not affordable
7          $(\pi^*, \Phi, H_n) \leftarrow$ FindPolicy( $\hat{RT}$,$\Phi$,$\hat{load}$) // Function call to find policy
8      else
9          $H_n \leftarrow \infty$                      // Cost affordable; execute full plan
10     end
11 else
                                         // $RT_{\mathcal{I}}$ is high; objective not satisfied
12     $\pi^* \leftarrow \pi_j \in \Pi : j = \underset{j}{\operatorname{argmin}} \hat{C}(\pi_j(\Phi))$            // Policy to minimize cost
13     $H_n \leftarrow$ Adjust($\Phi$,$\pi^*$, $\hat{load}$)              // Function call to adjust $H_n$
14 end
15 List of Actions $\leftarrow \pi^*(\Phi)$
16 Return List of Actions , $H_n$
```

HypeZon is concerned with (i) *control parameter tuning*, i.e., tuning the size of the planning and execution horizons, Φ and H_n respectively, and (ii) *policy switch*. ALGORITHM 5.1 shows an instantiation of the flowchart in Figure 5.7. The algorithm is a high-level description of HypeZon that is shared by both HZ_e and HZ_i. The only difference is the execution interval $\mathcal{I}$ of the hybrid planner in the two variants. Instantiating the generic steps of hybrid planning with HypeZon that are described in Figure 5.7 requires making certain assumptions and specifications; consequently, the steps can only be implemented as heuristics in the algorithms that are approximated instantiations. Therefore, the three algorithms in this section are heuristics that approximate the generic HypeZon scheme presented in Figure 5.7.

Assumption. Analogously to the considerations for Venus, we assume that the adaptable software and its operational context do not react to the adaptations in an unexpected way—see assumption (A3) in Chapter 4. Therefore, execution of an adaptation action results in the expected changes of the system and of the system utility.

Required inputs. ALGORITHM 5.1 requires estimation of system load or the number of the adaptation issues $\hat{load}$, set of available adaptation polices Π, the average response time of the system during the sampling interval $\mathcal{I}$, i.e., $RT_{\mathcal{I}}$, and estimation of response time for the current adaptation cycle, i.e., $\hat{RT}$. $\hat{RT}$ is approximated using linear regression based on $RT_{\mathcal{I}}$ and expected system load, i.e., $\hat{load}$.

For the planning problem captured in Φ, Π also includes estimations of quality and cost for the policies, i.e., $\hat{U}(\pi(\Phi))$ and $\hat{C}(\pi(\Phi))$ respectively. The planning horizon Φ is initially set to fully include the look-ahead horizon $\mathcal{L}$, i.e., all the the existing adaptation issues. $\mathcal{L}$ is the steps into the future, in terms of system changes, that are considered during planning. Φ is a prefix of the look-ahead horizon that is considered for planning—see Figure 5.3.

Policy switch thresholds. ALGORITHM 5.1 uses average response time $RT_{\mathcal{I}}$ as an indicator of the system objective satisfaction—see the decision node in Figure 5.7. However, any other system objective metrics with relevant thresholds can be considered. Later in the thesis, in the context of one of the application examples for evaluating HypeZon, instead of response time RT, we use performance ($\mathcal{P}$) of the services as indicators for system objective satisfaction. This means, ALGORITHM 5.1 replaces $RT_{\mathcal{I}}$ with $\mathcal{P}_{\mathcal{I}}$ that is the average performance of the system during the sampling interval $\mathcal{I}$. We discuss this in more details in Chapter 9. Based on the requirements for system business objectives, HypeZon partitions the range for objective satisfaction to *optimal* and *high* zones; *optimal* indicates that $RT_{\mathcal{I}}$ is within the acceptable range while *high* indicates that $RT_{\mathcal{I}}$ is higher than the permitted upper bound—see line 4 and 11 in ALGORITHM 5.1. The two zones constitute the full range for the response time values. HypeZon supports modifications of policy switch thresholds at runtime to reflect the changing goals and requirements.

Switching between the policies is charged with a cost proportionate to the runtime overhead that is caused by the switch insofar it requires deploying specific settings for the new policy, e.g., initializing a constraint solver or loading prediction models. The switch from policy π_i to π_j is charged with a cost c_{ij} that is subtracted from the system utility. HypeZon reasons about the trade-off between the cost and benefit of the switch at runtime—line 6 in ALGORITHM 5.1. $c_{curr,*}$ is the cost of switching from the current policy to π^*. When system business objectives are satisfied, i.e., $RT_{\mathcal{I}}$ is *optimal* (line 4), HypeZon selects the policy yielding the highest estimated utility with affordable cost—see Figure 5.7. HypeZon searches for a policy with the highest $\hat{U}(\pi(\Phi))$ (line 5) where the policy cost and the switch cost are acceptable. To do this, line 6 checks if the the sum of the policy switch cost ($c_{curr,*}$), $\hat{C}(\pi^*(\Phi))$, and the estimated response time $\hat{RT}$ is *high*.

For a planning problem in Φ, if ALGORITHM 5.1 does not find in its first attempt a match for policy π^* in Π that maximizes $\hat{U}(\pi^*(\Phi))$ with an affordable cost $\hat{C}((\pi^*(\Phi))$ (line 6), in line 7, HypeZon calls function FindPolicy to reduce the size of the planning problem until a quality-cost compromise is reached. If $RT_{\mathcal{I}}$ is *high*, indicating that the system objectives are not satisfied, HypeZon searches for a policy that reduces the response time as fast as possible—see Figure 5.7. To this end, HypeZon selects a policy with minimum planning cost (time) and adjusts its execution horizon accordingly—see line 11-14. Note that ALGORITHM 5.1 demonstrates only one of the several possible instantiations of the generic HypeZon scheme presented in Figure 5.7; the step in line 12 is defined according to the considered objective satisfaction indicator in ALGORITHM 5.1, i.e., the response time. Considering different objectives than response time requires different actions at this step of the algorithm to restore the violated objectives.

The FindPolicy function is defined in ALGORITHM 5.2; in addition to the policy information, estimated load, and estimated response time, the algorithm also requires an input parameter k. k is provided by the developer and defines how many iterations HypeZon is allowed to reduce the size of the planning horizon Φ to find a policy with an affordable cost that maximizes $\hat{U}(\pi(\Phi))$—see Figure 5.7 for an overview of the steps. FindPolicy is called by ALGORITHM 5.1 because for the given Φ, the main method could not find a policy that satisfies the cost and quality constraints. Therefore, FindPolicy reduces the size of the planning problem by 10% in line 6 of ALGORITHM 5.2[3];

3 The discounting factor for step-wise reduction of Φ is subject to change for different application domains or runtime conditions.

as a result, $\hat{C}(\pi^*(\Phi))$ is reduced accordingly. In order to save planning effort, before searching for a policy that maximizes the $\hat{U}(\pi(\Phi))$ in line 9, FindPolicy first calculates the size of the execution horizon H_n in line 7. H_n includes the issues that will be addressed in the current cycle—see Figure 5.3. In line 8, Φ is adjusted to only include the issues that remain in the execution horizon, thus planning for the rest of the issues is avoided; consequently, the planning cost $\hat{C}(\pi^*(\Phi))$ is reduced. As discussed earlier, both Φ and H_n are ordered lists and adjustments to their sizes entails removing the elements from the end of the lists.

ALGORITHM 5.2 : FindPolicy($\hat{RT}$, Φ,$\hat{load}$)

```
1  Require: $k$, $\Pi$, $\hat{RT}$,$\Phi$,$\hat{load}$
2  $\pi^* \leftarrow null$
3  $iteration \leftarrow 0$
4  while $iteration < k$ and $\pi^* = null$
5   do                                          // Check if planning horizon can be reduced
6      $|\Phi| \leftarrow 0.9 \times |\Phi|$                          // Reduce size of planning horizon
7      $H_n \leftarrow$ Adjust($\Phi, \pi^*, \hat{load}$)            // Function call to adjust $H_n$
8      $\Phi \leftarrow H_n$  // Planning Horizon is reduced to size of execution horizon
9      $\pi^* \leftarrow \pi_i \in \Pi : i = \underset{i}{\operatorname{argmax}}\, \hat{U}(\pi_i(\Phi))$   // Policy to maximize $\hat{U}$ for new $\Phi$
10     if ($c_{curr,*} + \hat{C}(\pi^*(\Phi)) + \hat{RT}$) high then      // If cost not affordable
11         $\pi^* \leftarrow null$                                 // No policy found
12     end
13     $iteration{+}{+}$
14 end
15 if $\pi^* == null$ then                                     // No policy found
16     $\Phi \leftarrow \mathcal{L}$      // Planning horizon gets all issues in look-ahead horizon
17     $\pi^* \leftarrow \pi_j \in \Pi : j = \underset{j}{\operatorname{argmin}}\, \hat{C}(\pi_j(\Phi))$        // Policy to minimize cost
18     $H_n \leftarrow$ Adjust($\Phi, \pi^*, \hat{load}$)            // Function call to adjust $H_n$
19 end
20 Return $\pi^*, \Phi, H_n$
```

For a policy π^*, the cost of the policy switch $c_{curr,*}$ and the estimated response time $\hat{RT}$ in line 10 are constant, therefore, to find a policy, only $\hat{C}(\pi^*(\Phi))$ can be reduced. FindPolicy reduces $\hat{C}(\pi^*(\Phi))$ via reducing $|\Phi|$, until a match that satisfies the policy switch condition is obtained or the algorithm reaches the end of the permitted iterations, i.e., k times (line 4-13). Failing to find a policy that balances the utility-cost trade-off (line 15), FindPolicy restores the original planning horizon Φ (line 16) and selects the policy with the minimum planning cost (line 17). The algorithm then tunes the execution horizon H_n accordingly (line 18). Finally, ALGORITHM 5.2 returns π^*, Φ, and H_n to the main method in ALGORITHM 5.1.

Both ALGORITHM 5.1 and ALGORITHM 5.2 call the function Adjust to tune the size of the execution horizon H_n based on the selected policy and the runtime conditions. This function is defined in ALGORITHM 5.3. Tuning the execution horizon at runtime requires information about the adaptation logic in the policies and the characteristics of the MAPE-K loop at the awareness level, e.g., operation time—see Figure 5.6. HypeZon captures this as the policy cost. Estimations of $\hat{C}(\pi(\Phi))$ are provided as inputs for Hype-

ZON. In addition to $\hat{C}(\pi^*(\Phi))$ in ALGORITHM 5.3, ADJUST requires information about the average system load or estimated number of adaptation issues to tune the size of the execution horizon. HYPEZON captures this as $\hat{load}$. Estimations for $\hat{load}$ are based on the average observations made over the sampling interval $\mathcal{I}$, i.e., since the most recent execution of the hybrid planner. The upper bound values for $\hat{load}$ and $\hat{C}(\pi^*(\Phi))$ are set by the developers and define the acceptable thresholds for each attribute. The thresholds are shown as the range *high*—see line 3 and 4. The *high* range threshold for the cost $\hat{C}(\pi^*(\Phi))$ in ALGORITHM 5.3 is defined according to the available budget.

ALGORITHM 5.3 : ADJUST($\Phi, \pi^*, \hat{load}$)

```
1  Require: $\Phi, \pi^*, \hat{load}$
2  $H_n \leftarrow \Phi$     // Execution horizon gets all the issues in planning horizon
3  if $\hat{load}$ high then                          // Estimated load/ issue # is high
4      if $\hat{C}(\pi^*(\Phi))$ high then             // Estimated policy cost is high
5          $|H_n| \leftarrow 0.9 \times |H_n|$                    // Mildly reduce $H_n$
6      else
                                                      // Estimated policy cost is low
7          $|H_n| \leftarrow 0.5 \times |H_n|$                  // Severely reduce $H_n$
8      end
9  end
10 Return $H_n$
```

ADJUST in ALGORITHM 5.3 initially assigns all the issues in Φ to be included in H_n (line 2). When the expected system load is not *high*, independent of the planning cost of π^*, the algorithm executes all the actions in the planning horizon. High system load or larger amount of anticipated adaptation issues, e.g., a failure burst, demand adjustments of the execution horizon according to the planning cost of the corresponding policy.

When the system is experiencing high load, it is important to make sure that the most recent critical issues are not ignored because of a lengthy execution of the current adaptation cycle. Therefore, when $\hat{load}$ is *high*, ALGORITHM 5.3 reduces the size of H_n (lines 3-9) to only execute the prioritized actions in the current adaptation cycle. The cycle is aborted afterwards to consider the most recent issues in the planning. A plan to resolve the issues in Φ is an ordered list of actions where actions that address the most critical issues are prioritized—see Section 5.2. However, partial execution of a plan is only beneficial if the planning cost of π^* is relatively low and the expected number of issues ($\hat{load}$) is high.

When both $\hat{C}(\pi^*(\Phi))$ and $\hat{load}$ are high, ALGORITHM 5.3 mildly reduces H_n, e.g., by 10% (line 5), as otherwise, critical issues might be ignored for a long time if H_n is too large, i.e., $|H_n| = |\Phi|$. Another risk is that reducing H_n to a very small size forces π^* to re-plan frequently. As a result, the high policy cost adds more overhead to an already over-loaded system. Conversely, low policy cost allows for more extreme reduction of the execution horizon (line 7). Smaller H_n entails more frequent planning by π^* but leads to shorter waiting time for the most recent adaptation issues in the queue—as otherwise they are ignored until the subsequent adaptation cycle and after the full plan is executed by the current cycle. For a π^* with small $\hat{C}(\pi^*(\Phi))$, if the hybrid planner

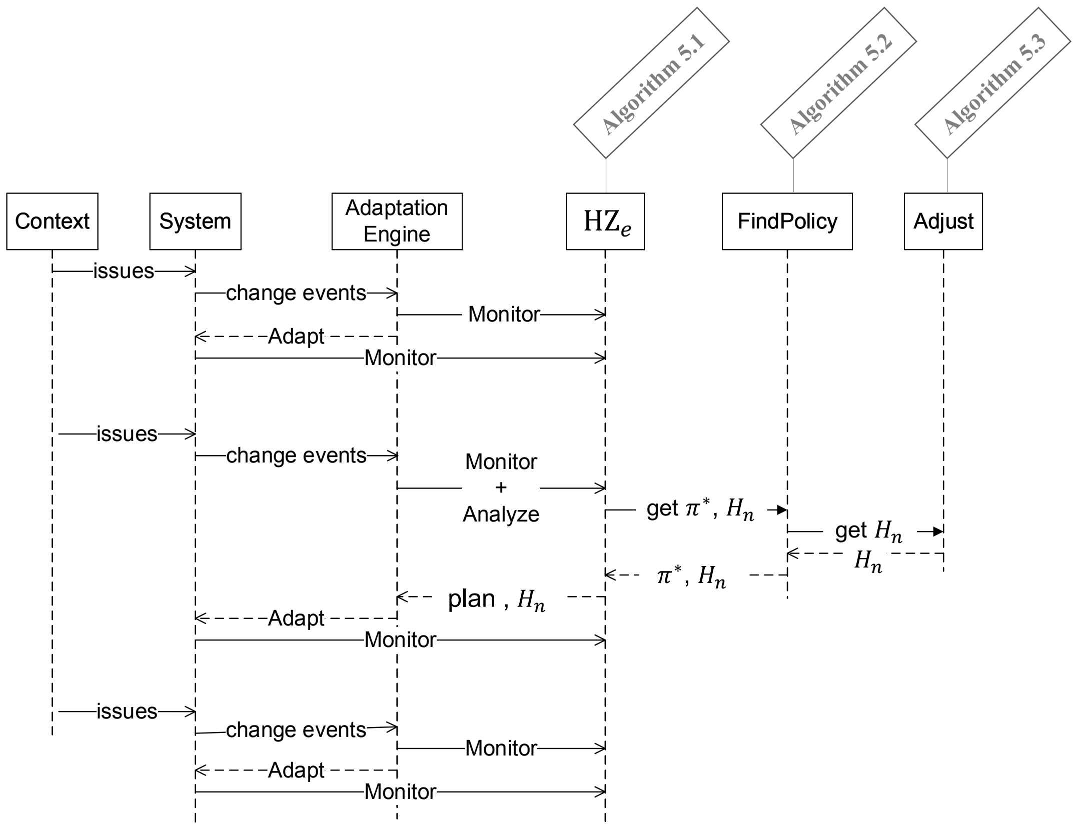


Figure 5.8: Sequence diagram for HZ$_e$ execution with $\mathcal{I} = 2$.

expects high system load or large number of adaptation issues, H_n is reduced more severely, e.g., by 50%. Finally, the algorithm returns H_n to the calling methods.

In mRUBiS, consider a case where the hybrid planner is dealing with a list of issues indicating service crashes and under-utilized service replicas. Addressing a service crash is more critical for the availability of the system, thus should be prioritized over optimizing the utilization of service replicas—see Section 5.1. Thus it is beneficial to address the service crashes as soon as they occur. Reducing the size of the execution horizon results in shorter execution times. Consequently, this allows for making new observations more frequently, thus considering most recent issues sooner as otherwise, would be ignored until the full plan is executed.

Figure 5.8 shows a sequence diagram of HZ$_e$ execution with $\mathcal{I} = 2$. The diagram includes three adaptation cycles; each occurrence of the adaptation issues emits change events to the adaptation engine. The adaptation engine that implements a MAPE-K loop is executed to address the issues via adapting the system. The execution interval $\mathcal{I}$ for the hybrid planner is set to two, therefore, HZ$_e$ is only executed in every other adaptation cycle, i.e., the second cycle in Figure 5.8. While the meta-awareness loop in HZ$_e$ monitors the adaptation engine in combination with the system before and after *each* adaptation, the hybrid planner, implemented via ALGORITHM 5.1–5.3, is only executed for every other adaptation since $\mathcal{I} = 2$—see Figure 5.6a for the architectural design of HZ$_e$. Figure 5.9 shows a sequence diagram for HZ$_i$ during three adaptation cycles where the hybrid planner is executed in *every* adaptation cycle. As discussed earlier, compared to HZ$_i$, HZ$_e$ collects accumulated observations of system executions that can be used to estimate the operation conditions for the next interval.

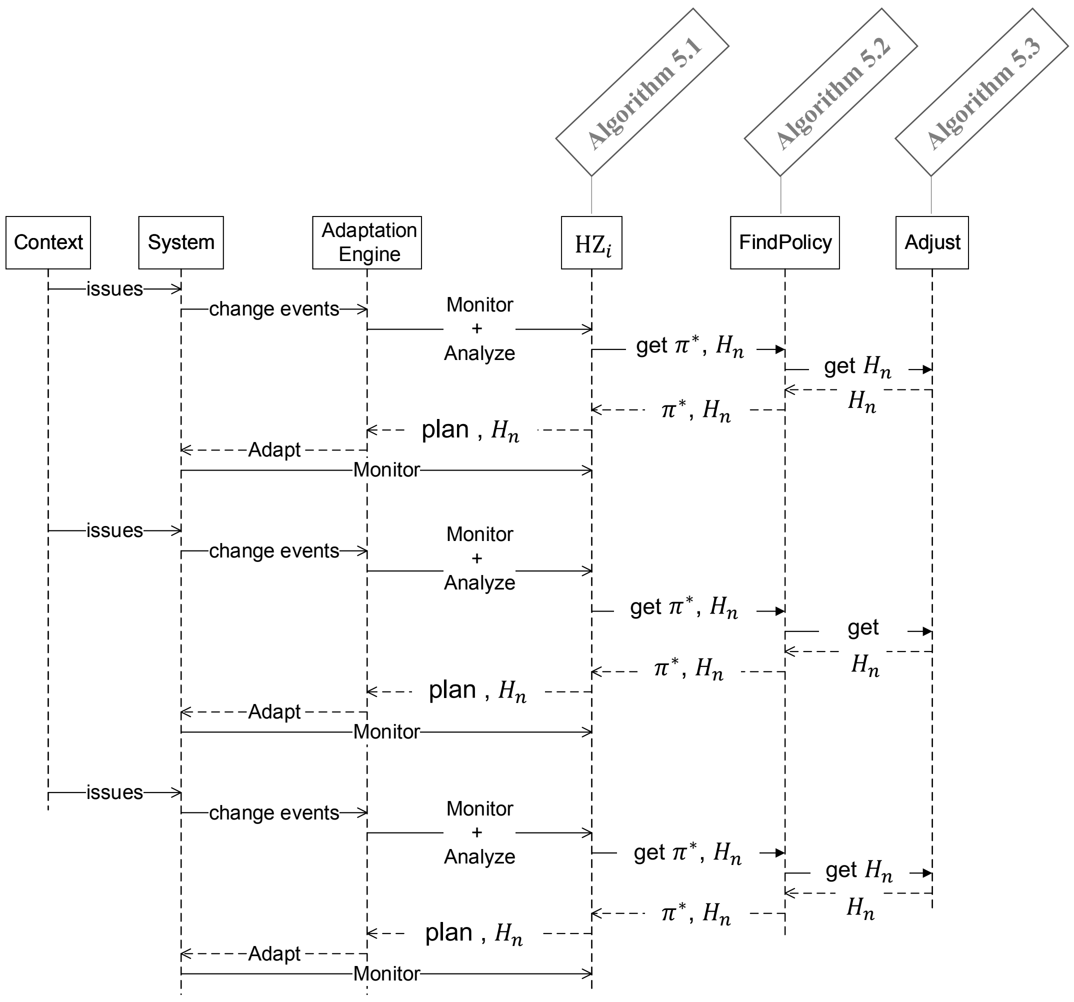


Figure 5.9: Sequence diagram for HZ_i execution.

HZ_i, however, due to its architectural design, is executed at each adaptation cycle, and makes control and switch decisions based on the observations over a relatively shorter period—see Figure 5.6b for the architectural design of HZ_i.

5.4 SUMMARY

When the complexity of the adaptation space exceeds the ability of individual adaptation solutions, to solely, fulfill both the quality and the cost requirements of the adaptation, hybrid adaptation mechanisms that coordinate, at runtime, multiple off-the-shelf adaptation polices are beneficial. Such solutions, piece-wise, exhibit properties of their constituents while avoiding a cost-intensive process of developing new custom solutions for the problem at hand. In this chapter, we introduced HypeZon, a hybrid planner for self-adaption with receding horizon. The scheme complements the proposed mechanisms for incremental architecture-based self-adaptation in this thesis by offering a hybrid planner that implements the Plan activity of the adaptation engine— see Figure 5.1 for an overview. HypeZon exhibits the characteristics of a generic solution for hybrid adaptation since it is designed to consider the employed adaptation policies as black-box and can coordinate arbitrary adaptation policies.

We discussed the use of receding horizon in HypeZon to adapt to the operation conditions and exhibit control flexibility at runtime. By considering hybrid adaptation as a case for meta-self-awareness, HypeZon supports systematic and consistent acquisition of a broad view on the adaptable software, the adaptation process, and the individual adaptation polices allowing for observing phenomena with global scope that are not visible at the same level as the adaptation loop, thus supporting more informed decisions for switching between the individual policies. The generic scheme for hybrid planning in HypeZon was instantiated via heuristic-based algorithms. We discussed through detailed algorithms how HypeZon coordinates a set of input policies to balance the quality-cost trade-off while opting for maximizing system objective satisfaction.

The scheme advances the state-of-the-art for hybrid adaptation by providing explicit architectural design. Conforming to meta-self-aware architectures, we proposed two designs for HypeZon that support explicit separation of concerns, i.e., adaptation and policy invocation, at the architecture level allowing for reusability, easier maintenance, and independent evolution of each level. Consequently, separate and independent mechanisms for observing, learning, and reasoning logic may be employed by each level.

Part III

EVALUATION AND CONCLUSION

In this part of the thesis, we discuss the employed application examples to evaluate the solutions. We outline the implementation framework and the experiment environment which we use for the evaluation. Afterwards, we comprehensively evaluate our incremental adaptation engine and Venus in a comparative study with alternative self-adaptation solutions. We assess the runtime performance, scalability, optimality, and robustness of Venus through a set of qualitative and quantitative experiments. Next, we present an instantiation of the proposed methodology for training utility-change prediction models on mRUBiS and evaluate the methodology by assessing the prediction error and runtime performance of the models. We evaluate the adaptation engine and Venus prior to the learning methodology. The reason is that, application of the methodology builds on existence of an adaptation engine and an adaptation mechanism which we demonstrate and evaluate first. Next, we evaluate HypeZon in a comparative study with an alternative solution for hybrid adaptation. Moreover, we discuss related work to contrast the contributions of this thesis with the state-of-the-art in engineering self-adaptive software; more specifically, we study how the state-of-the-art in planning mechanisms satisfy the requirements for architecture-based self-adaptation of dynamic architectures. Finally, we conclude the thesis and provide an outlook on the future work.

6 EXPERIMENTAL APPLICATION EXAMPLES

In this chapter, we present the two application examples, i.e., mRUBiS and `Znn.com`, that are used for the qualitative (only mRUBiS) and quantitative evaluation of the solutions proposed in this thesis. First, in Section 6.1 we summarize the description of the mRUBiS application that is already introduced in Section 2.5. Then, we present `Znn.com` in Section 6.2. Finally, we discuss the evaluation methodology and present the set of input traces/logs that we use for the experiments with both application examples in Section 6.3.

6.1 MRUBIS

As a running example throughout this thesis, we used the Modular Rice University Bidding System (mRUBiS) [414] to illustrate Venus and HypeZon. We have equipped the application with an adaptation engine based on a MAPE-K feedback loop that realizes the self-healing and self-optimizing properties—see Section 2.5. More specifically, we defined architectural pattern-based utility functions to represent the business objectives of mRUBiS in Chapter 3. In Chapter 4, we presented a MAPE-based description of Venus and its features based on an implementation of mRUBiS. Finally, in Chapter 5, we used a motivating scenario based on mRUBiS to illustrate the details of HypeZon. In the following, we summarize the description of mRUBiS initially introduced in Section 2.5.

mRUBiS is an online marketplace that implements the core functionality of an auction site: selling, browsing, and bidding. Technically, it is derived from RUBiS which is a popular case study and an open source benchmark to evaluate control theoretic adaptations as well as self-adaptive software systems with performance concerns [332]. mRUBiS extends RUBiS by adding new functionalities and modularizing its monolithic structure. The modularization enables the architectural adaptation of mRUBiS. The marketplace hosts arbitrary number of shops, i.e., tenants. The architecture of each shop is isolated from the architectures of the other shops as each shop is configured independently from the other tenants—see Figure 6.1 for an architecture of a tenant in mRUBiS. A shop in mRUBiS consists of 18 components that are individually deployed. All shops share the same component types but each shop has its own individually configured components. The architecture of a tenant consists of components to manage the products (Item Management Service), users (User Management Service), auctions and purchases (Bid and Buy Service), inventory (Inventory Service), rating of users (ReputationService), and components to authenticate users (Authentication Service) and to persist (PersistenceService) and retrieve data (Query Service) from the database. Employing a pipe of filter components that follows the batch sequential pipe-and-filter architectural style improves the results when users search for items, i.e., products. It iteratively adjusts the list of items obtained from the query service and removing items that are not considered as relevant for the specific user and search request.

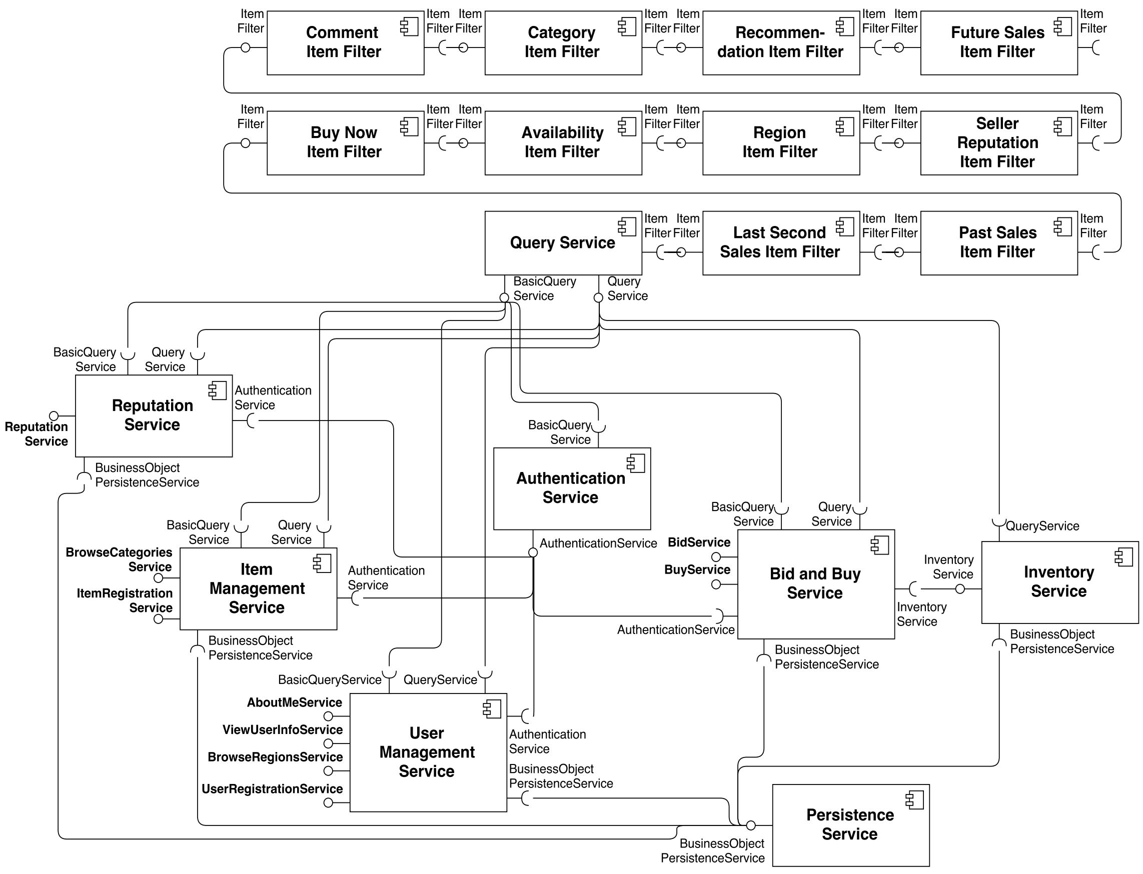


Figure 6.1: Architecture of a tenant in mRUBiS from [413].

The business objectives of a company running mRUBiS, i.e., obtaining high volumes of sales, is achieved via high availability of the shop services and low response times to client requests. In order to make sure that these goals are met, or in case of violation, the system is quickly brought back to a state where it satisfies its business objectives again, self-adaptation should be employed to automatically repair failures, i.e., self-healing by detecting, diagnosing, and recovering from disruptions [185, 343] and improve performance, i.e., self-optimization by reconfiguring the system [430]. The self-healing and self-optimizing properties are added to mRUBiS via employing a MAPE-K feedback loop as an adaptation engine.

In this thesis, we used mRUBiS as one of our application examples to investigate two different self-adaptation properties, i.e., self-healing and self-optimization of software systems. We equip mRUBiS with a MAPE-K feedback loop that uses an architectural RTM of mRUBiS. Specifically, the model represents the runtime architecture of mRUBiS according to the deployment of mRUBiS on an application server. Thereby, we consider both parameter and structural adaptation of software systems. mRUBiS application supports both types of adaptation. Restarting a component in mRUBiS is realized with parameter adaptation while replacing a component with an alternative requires structural adaptation that reconfigures the architecture. Moreover, the case study can be executed in combination with synthetic input traces that allow for investigation of the self-adaptive properties under a wide range of operation conditions. An individual input trace only captures one possible future for a simulated self-adaptive system and fails to cover a large and representative spectrum of the input space. Employing such a trace results in inconclusive output for the self-adaptive system under evaluation and lacks generality—we discuss this in the context of the evaluation methodology and input traces in Section 6.3.

6.1.1 *Self-healing and Self-optimizing mRUBiS*

We use the self-healing and self-optimization properties of mRUBiS to discuss Venus and HypeZon. To achieve high availability, the self-healing aims at repairing architectural failures that disrupt the operation of mRUBiS. For the mRUBiS architecture, four types of Critical Failures (CFs) are considered as introduced in [414] that should be handled by self-healing. They target Components that either crash and enter the UNKNOWN life cycle state (CF1), throw Exceptions exceeding a given threshold (CF2), are destroyed and removed from the architecture (CF3), and Connectors that are lost and removed from the architecture (CF4). The four CFs are the negative patterns that affect the system. The rule set $\mathfrak{R}$ includes the adaptation rules. As adaptation options, i.e., rules in Figure 2.8, mRUBiS supports restarting, redeploying, and replacing components, as well as recreating connectors. For the redeployment, there are two variants. The light-weight variant keeps the latest configuration while the heavy-weight variant resets the configuration parameters of the redeployed component.

In order to have low response times for the services, i.e., components, in mRUBiS, the self-optimization aims at automatically improving the performance of mRUBiS by architectural reconfiguration. We define one Performance Issue (PI1) that indicates the performance of a component in mRUBiS is below a threshold. Changes in the load of a component may bring mRUBiS to sub-optimal state that requires to be handed by self-optimization of the system. Significant increase in a component load reduces the performance of the component by overloading the active replicas of the component. Equation 3.9 shows how we measure performance. Moreover, reduced load may cause underutilization of the corresponding replicas for a component, thus reducing the performance. As adaptation options, mRUBiS supports adding and removing service replicas until the performance of the affected components reaches its desired value. Note that in case of extreme traffic, self-optimization may be restricted by the available resources thus failing to bring the component to its optimal state regarding performance.

Each adaptation rule in mRUBiS has three attributes: costs, utilityIncrease, and ratio—see the metamodel of mRUBiS in Figure 2.8. Costs attribute captures the expected execution time of the corresponding rule; utilityIncrease is the impact on the utility when applying the rule; ratio is the defined as utilityIncrease/Costs. mRUBiS can host arbitrary number of tenants, each containing 18 components. We define the overall system utility, i.e., the utility of the mRUBiS architecture G, conforming to the principals of additive utility functions [246, 340]—see Section 3.1.1. Hence, the utility of a tenant is the sum of the sub-utilities of all its constituent components as shown in Equation 5.1 and, similarly, the overall utility of mRUBiS is the sum of the utilities of its tenants—see Equation 3.2.

The system elements related to the self-healing and self-optimizing issues, i.e., the component life cycle (CF1), exceptions (CF2), components (CF3, PI1), and connectors (CF4) are represented in the mRUBiS RTM through the causal connection and thus observable in the running mRUBiS. The feedback loop operates on this reflection model that describes the runtime architecture of mRUBiS including these system elements. This model is a runtime instantiation of the mRUBiS metamodel presented in Figure 2.8.

As discussed in Chapter 4, Executable Story Diagrams (SDs) are used in this thesis to realize the rule-based adaptation scheme—see Section A.1 for more details on the SD formalism. Diagrams are similar to ECA rules—see Section 2.3.2—except that SDs are a visual language and based on graph transformations. The implementation of the analysis activity includes a code snippet that consumes the change events for the RTM. Based on these events, the SD interpreter is invoked to execute the corresponding analysis SDs. For instance, if the number of the exceptions in the `providedInterface` of a component in mRUBiS exceeds five, the model is updated by representing this as a CF2 attached to the `providedInterface` of a component in the RTM. Such an update causes a specific change event that indicates a potential occurrence of CF2. In Figure 4.5, we presented an analysis rule for detecting CF2 in mRUBiS based on SDs and the planning SD rule for addressing the CF2 was presented in Figure 4.6, respectively. Finally, once the `RestartComponent` rule is assigned to handle the CF2, we showed the rule for executing a component restart in Figure 4.7. For each of the CFs as well as PI1 that may potentially affect mRUBiS, we created similar SDs that together specify the analysis, planning, and execution activities of the MAPE-K loop. We presented the SDs addressing CF2 and `RestartComponent` in Section 4.3 due to their simplicity since the SDs for the rest of the issues and rules are more complex.

6.2 ZNN.COM

As a second case study, we employ `Znn.com` [91] that simulates a news service website based on real-world sites like CNN.com. Clients make content requests and a load balancer distributes requests across a server pool. The size of the pool can be dynamically adjusted to balance server utilization against request response time. Architecturally, `Znn.com` is a web-based client-server system conforming to an N-tier style—see Figure 6.2. Certain system and client information such as server load, request response time, and the connection bandwidth can be monitored. The goal of `Znn.com` is to provide short response times to clients while keeping the cost of the server pool within the budget limit.

The content requests are not uniformly distributed over time and demonstrate the *slashdot effect*, i.e., sudden and relatively temporary surges in traffic. As a result of the slashdot effect, the response time of the servers increases above the acceptable threshold and causes `latency` issues for the affected clients. `Znn.com` handles `latency` via enlistment of servers and reducing the quality of the content. The employed remedies,

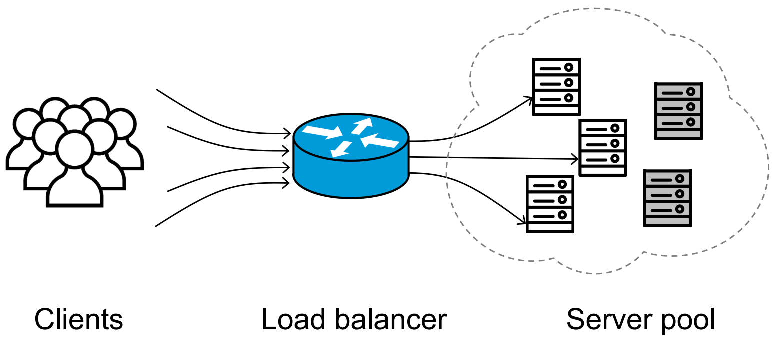


Figure 6.2: High-level view for `Znn.com` system.

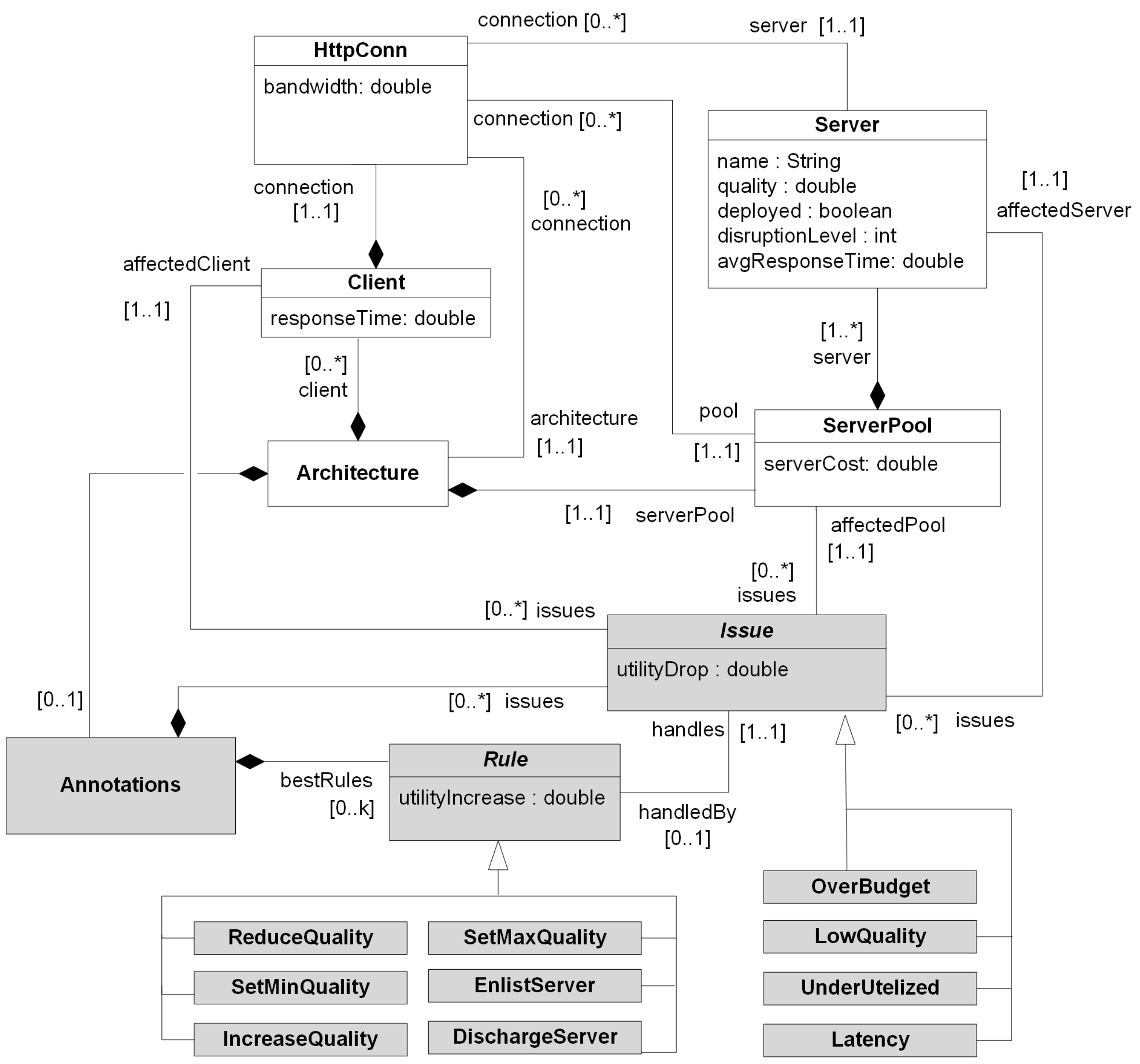


Figure 6.3: Simplified metamodel of `Znn.com`.

however, could cause `overBudget`, `underUtilized`, and `lowQuality` issues which are handled by discharging redundant servers and increasing content quality. Thus during execution of `Znn.com`, four potential issues may affect the system: `latency` in the response times, `overBudget` cost of adding servers to the pool, `underUtilized` servers, and `lowQuality` contents.

In order to fulfill system objectives, i.e., short response times and low operation cost of the server pool, we add self-adaptive properties to `Znn.com` via adding a MAPE-K feedback loop that uses an architectural RTM of the system. The model represents the runtime architecture of `Znn.com` and is presented in Figure 6.3 based on the widespread ECORE syntax [391] which defines valid model instances [44]. The metamodel of the RTM captures the `Znn.com` Architecture with a set of Servers constituting a ServerPool. Clients are connected to the Architecture via HttpConn links. Each established connection, i.e., an instantiation of HttpConn, has a bandwidth. Servers are parameterized by setting the value attribute of the properties, e.g., quality and disruptionLevel. The ServerPool includes at least one activated Server. All the Servers have the same operation cost that is stored as the

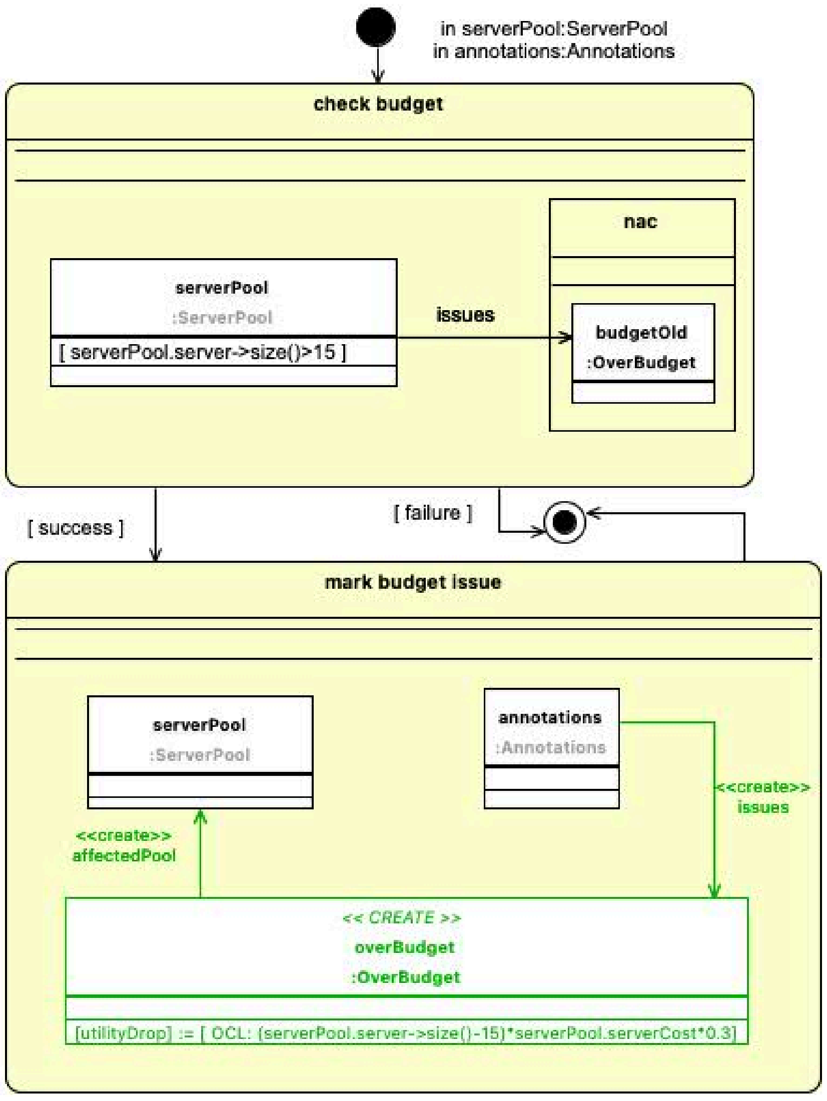
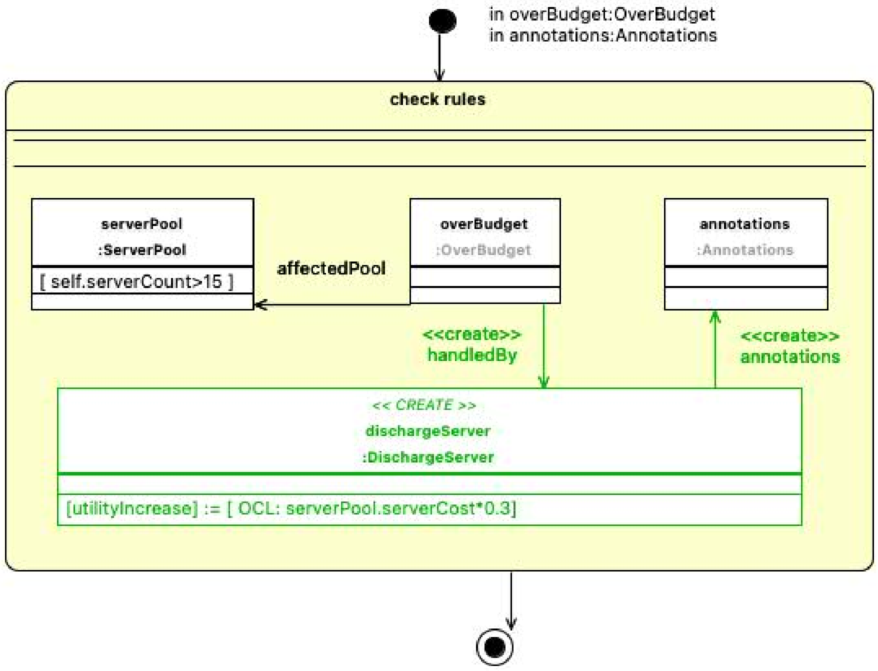


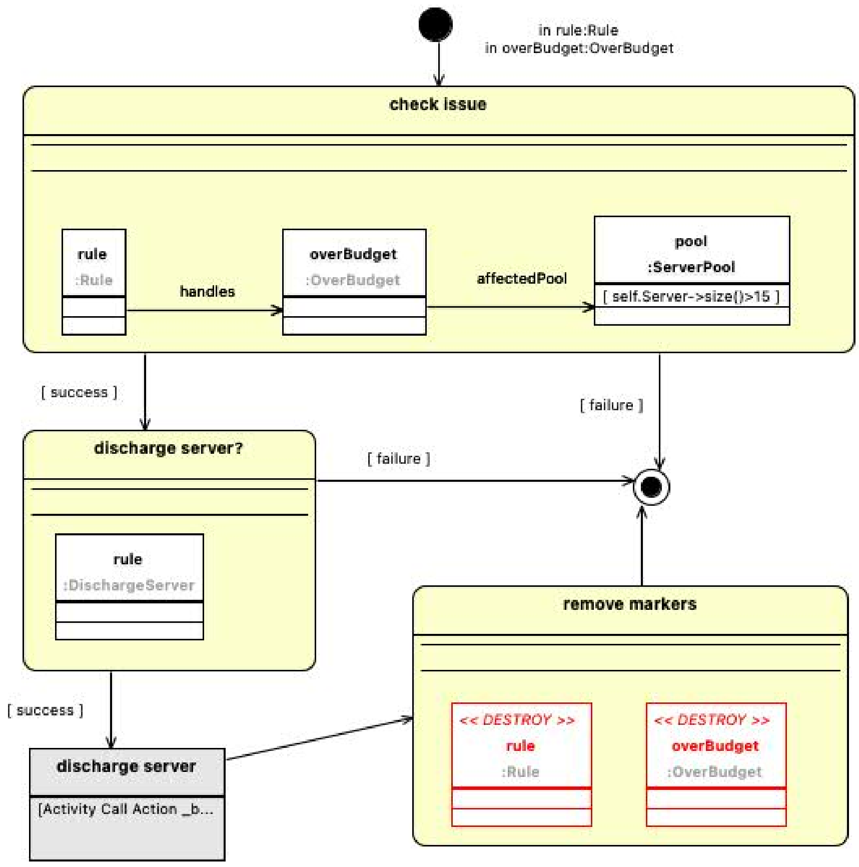
Figure 6.4: Analyze SD for `overBudget` issue.

serverCost attribute in serverPool. These elements allow us to describe the runtime architecture of `Znn.com` and the runtime issues, e.g., delays or sub-optimal configurations. The elements colored gray are relevant for self-adaptation of `Znn.com`. The activated Servers in the ServerPool may exhibit Issuess of type `underUtilized`, and `lowQuality`. As the number of the Servers grow above the budget limit, it causes `overBudget` issues in the ServerPool. A Client can face a `latency` issue caused by response time days. The issues that affect `Znn.com` are annotated in the RTM via model Annotations. The adaptation Rules that modify the configuration of `Znn.com` to address the issues include discharging and enlisting servers, increasing and reducing content quality, and setting the content quality of servers to the possible minimum and maximum.

The Monitor activity of the MAPE-K loop, owing to the causal connection between `Znn.com` and its RTM, incrementally updates the architectural RTM based on the changes of the running `Znn.com`. The model then reflects the current state of the system. Next, the updated RTM is analyzed for symptoms of the adaptation issues, i.e., `latency`, `overBudget`, `underUtilized`, and `lowQuality`. The employed Analyze activity is event-



(a) Plan SD for `overBudget`.



(b) Execution SD for `overBudget`.

Figure 6.5: SDs specifying planning and execution for `overBudget` issue.

/state-based as it is driven by change events resulting from updates of the architectural RTMl, i.e., of the state, during monitoring— see Section 4.3.3.

6.2.1 *Analyze, Plan, and Execute Activities with SD*

Similar to mRUBiS, we operationalize the MAPE activities in `Znn.com` via the SD rules; implementation of the Analyze activity includes a code snippet that consumes the change events for the RTM—see the source code in Section A.2. Based on these events, the SD interpreter is invoked to execute the corresponding analysis SDs. For instance, if the number of the activated Servers in the ServerPool exceeds the budget limit, e.g., 15 servers, the model is updated by representing this as an `overBudget` issue attached to the ServerPool in the RTM. Such an update causes a specific change event that indicates a potential occurrence of `overBudget`. Figure 6.4 shows the Analyze SD for identifying an `overBudget` issue. The SD has two input parameters: the ServerPool model element obtained from the change event that notifies about the exceeding number of the activated servers and the Annotations element that contains all the annotations of the RTM. The first Story Pattern (SP), titled *check budget*, checks if there are more then 15 servers in the ServerPool. The *Negative Application Condition* (NAC) element in the SP checks for old markings of the `overBudget` issue; if no existing marking is found, an `overBudget` issue is identified and the second SP annotates the affected ServerPool with an `overBudget` marker element; the SD terminates immediately. For each of the four issues shown in the metamodel of `Znn.com` in Figure 6.3, we specified SD rules that analyze the model for occurrences of the issues and annotate the RTM with the markings of the issue and the affected model element. The green parts in the SD are modifications to the RTM elements during analysis. The occurrence of the `overBudget` issue results in a drop in the ServerPool utility. The SD in Figure 6.4 creates the `overBudget` annotation in the RTM with the computed utilityDrop that points to the affected component, i.e., the ServerPool. We omit the details to avoid multiple annotations for the same issue.

For each of the identified issues, the planning activity of the MAPE loop assigns the proper rule to resolve the issue. `Znn.com` supports six adaptation rules as depicted in Figure 6.3. We specify mappings of the issues to the corresponding rules with SDs. Remember from Section 4.1 that a rule $r = (\mathsf{LHS}, \mathsf{RHS})$ is considered as applicable to an issue if the pattern P, defining the issue in the RTM, creates a match for the LHS, i.e., the pre-condition of r. For instance, the `dischargeServer` that discharges one server from the pool, is triggered when the number of activated servers in the pool exceeds 15. If the pre-condition holds, the rule `dischargeServer` becomes applicable. Figure 6.5a shows the Plan SD for `overBudget` where a `dischargeServer` rule is assigned to handle the issue. The assignment is realized via creating a `dischargeServer` marker element and associating it to the issue. The green parts are modifications to the RTM elements during planning. The issue points to the affected model element, i.e., the ServerPool. Similar to the issue markers, rule markers are annotations to architecture elements in the RTM. The result of the planning activity is an architectural RTM of the system with rule marker elements. These elements represent the adaptation rules that are assigned to the occurred issues and that should be executed by the Execute activity.

Finally, the Execute activity executes the adaptation rules that are assigned to the issues in the RTM. Following a models@run.time approach, the adaptation is first performed on the RTM of `Znn.com` and then synchronized to the system. This approach allows us to specify and execute the adaptation rules with SDs that operate at the model level. Figure 6.5b shows the SD for the execution of `dischargeServer`.The first SP checks the negative pattern once more, i.e., whether the ServerPool contains more than 15 servers; if not, the execution of the rule is aborted and the SD terminates. Otherwise, another SD (not shown here) is invoked via the activity call action node (in gray) to discharge one server from the pool and re-distribute its load among the remaining activated servers. In the next SP, having executed the `dischargeServer` rule, the corresponding issue and rule markers are removed from the model; the SD terminates afterwards.

The rest of the three issues, i.e., `latency`, `underUtilized`, and `lowQuality` are also detected and resolved via a set of SDs that together specify the Analyze, Plan, and Execute activities of the MAPE-K loop. In this section, we presented the SDs for the `overBudget` issue and the corresponding `dischargeServer` rule due to their simplicity. As discussed earlier in Section 4.3, the MAPE activities are driven by changes of the architectural RTM in terms of the markers, i.e., model annotations that have been added to the model by their predecessor activity; this supports an incremental processing as the execution of a rule starts with a rule marker that is directly associated to the issue marker and the affected model element. With this information, the SD interpreter is invoked by the Execute activity to execute the corresponding SD that realizes the adaptation rules. Thereby, the adaptation performed at the model level is incrementally synchronized to `Znn.com`.

6.2.2 *Utility Function for `Znn.com`*

The goal of employing self-adaptive properties in `Znn.com` is automating decisions that trade off multiple objectives to adapt the system. More specifically, `Znn.com` with self-adaptive properties exhibits automated system management to adjust the server pool size with respect to multiple objectives, i.e., cost and response time, and switch the content quality of each server. Response time, content quality, server utilization, and budget are the four objectives of `Znn.com`. These objectives are captured by four quality dimensions respectively. Each dimension is represented by a utility sub-function as described in Table 6.1 that can either be observed, e.g., response time, or obtained from the system configuration, e.g., server utilization. Utilities are assigned to particular values of the quality dimensions. Cheng et al. in [91] employ a discrete custom utility function in the form of (x_i, y_i) where the utility y_i is assigned to the value x_i of the quality attribute. For example, the discrete utility function $(low, 1)$ for response time means that low response time obtains a utility value equal to 1. We Extend the discrete custom utility functions introduced in [91] for `Znn.com` and replace them by linear utility sub-functions as described in Table 6.1.

RT in Table 6.1 is the estimated request response time and RT_{max} is set to 90 seconds, i.e., when `Znn.com` throws a *request timeout* exception and ends the session. A server in `Znn.com` can transfer content with three different qualities that are quantified as: (low, 0), (medium, 0.5), and (high, 1). Therefore, $Server.quality$ can have one of the 0, 0.5, or 1 values. $Server.utilization$ is the percentage of the server capacity that is in-use, and finally, $Server.cost$ is the operational cost of the server which can vary for different providers. Similarly to [91], we define the overall utility of the system as a weighted sum of the utility sub-functions. The weights w_i in Table 6.1 are extracted

Table 6.1: Utility sub-functions for Znn.com.

ID	Quality Dimension	Utility Sub-function	w_i
u_R	Response time	$u_R = 1 - \frac{RT}{RT_{max}}$	0.4
u_Q	Content quality	$u_Q = Server.quality$	0.2
u_U	Server utilization	$u_U = Server.utilization$	0.1
u_C	Cost	$u_C = Server.cost$	0.3

from [91]. For each state $s = (\bar{c}, \bar{o})$, the overall utility of `Znn.com` is defined according to Equation 6.1. For each target state s', the expected utility of the state is an estimation of Equation 6.1, i.e., $EU(s') = \hat{U}_{znn}(s')$.

$$U_{znn}(s) = \sum_{client} w_r u_R + \sum_{server} (w_q u_Q + w_u u_U - w_c u_C) \tag{6.1}$$

In Figure 6.4, the drop in the utility of the system caused by the `overBudget` issue is calculated at runtime and the related attribute of the `overBudget` element, i.e., utilityDrop, is defined using the *Object Constraint Language* (OCL) [355] assignment expression. The utilityIncrease of the corresponding rule, i.e., the impact of the rule application on the utility, is computed and assigned during the planning—see Figure 6.5a. The execution of the `dischargeServer` rule discharges one server from the pool, thus the the utility of the server pool is increased according to $w_c \times u_C$ in Table 6.1.

The internet phenomenon, known as the slashdot effect, introduced earlier, motivates the use of `Znn.com` instantiation in this thesis as it provides for multiple runtime scenarios where system configuration should be adapted at the architectural level to fulfill system objectives. Moreover, an instantiation of `Znn.com` in this thesis explores the composability of adaptations across different quality attributes, i.e., response time, content quality, server utilization, and cost.

6.3 EVALUATION METHODOLOGY AND INPUT TRACES

The experiments with mRUBiS and `Znn.com` are both based on simulated systems since we are not executing a real system in its real operational environment. Thus, the simulated systems are executed in combination with a set of *input traces* that are manually injected to the adaptable software. In this context, we conducted an SLR of state-of-the-art in evaluating self-healing systems in [176] and extended it in [177]. Our findings suggest that proper design and evaluation of self-adaptive software still remains an open issue, since critical elements for the design space of a self-adaptive system under evaluation, e.g., *fault models*, as identified by [270], are often missing. The observations revealed that while majority of the work that are concerned with performance of self-healing systems use simulation-based evaluation (97%), they typically do not employ *representative input traces* for evaluation. Workload related metrics such as response time, throughput, and working versus adaptivity time need to be measured to analyze the performance for self-adaptive systems—see [238]. Representative input trace is a (set of) input trace(s) with volatile characteristics that provide coverage of a large region of the potential operational environment of the simulated system. Therefore, the credibility and the scope of validity of the claims in the systems remain unclear.

In the specific context of self-healing systems that are concerned with runtime failures, taking into account that: (i) the characteristics of their operational environments are always only known to a limited extent due to the data sparsity caused by the rare nature of the failures; (ii) the characteristics of their operational environments are also subject to change over time due to runtime uncertainty; (iii) we are interested in a robust solution whose performance profile is generalizable to different operation conditions; it is necessary to consider multiple alternative *probabilistic input traces* that mimic the runtime uncertainty of a real-world operation context. In the face of our findings in [177] we recommended improving the efforts in evaluation of self-adaptive systems such that multiple representative probabilistic input traces are employed during evaluation. The reason is that, a single input trace for a system under evaluation only allows for evaluating the system for *one* (of the many) possible future(s) and lacks generalizability. In Figure 1.1 we showed an example of how evaluating an approach against only a single (or a non-representative set of) data points in the input space of the self-adaptive software yields inconclusive results—see how the best performing solutions in Figure 1.1a change as complexity increases.

The technical contributions of the thesis are evaluated according to an *evaluation methodology* that is developed based on our SLR. The methodology builds on the required improvements for the current practice in evaluation of self-healing systems introduced in [177] and suggests that conducting multiple reproducible experimental runs under controlled circumstances is required to obtain robust, conclusive, and reliable results from the evaluation. Generic credibility claims on evaluation of the system are only justified if the results are tested for robustness in the presence of large spectrum of the input space. Therefore our evaluation methodology entails using multiple (probabilistic) input traces with volatile characteristics that provide coverage of a wide spectrum of the input space for the system under evaluation. As discussed in the context of contribution **C**8 in Chapter 1 and confirmed by our SLR in [176, 177] (that is extended in Chapter 10), the evaluation in this thesis goes beyond the state-of-the art in providing coverage of a wide spectrum of the potential input space for self-adaptive systems. Next, we introduce the input traces that we use during evaluations with mRUBiS and `Znn.com`.

6.3.1 *Input Traces for mRUBiS*

As revealed by the foundational work on characterizing failure models in computer systems [77, 228, 400], failures are often not independently occurring, but correlated in time or space, referred to as failure *bursts*. The phenomenon is associated to the effects of failure propagation where a single failure in the system triggers a sequence of failures in other system components within a short period of time. While several fault tolerant algorithms make the assumption that failures occur independently [see 206, 436], this assumption contradicts the *bursty* failure models and neglects that occurrences of failure bursts often result in correlated availability behaviors of different components. Iosup et al. showed in [222] that ignoring the bursty character of failure models results in overestimating the transient reward rate by an order of magnitude, even if only as few as 10% of the failures conform to a bursty model.

Table 6.2: General characteristics of failure models.

	Deterministic Failure Model	**Probabilistic Failure Models**
FGS	Constant	Varies for each failure (group) occurrence
IAT	Large enough	Varies for each failure (group) occurrence
FET	0	Larger than 0

The *granularity*, *magnitude*, and *duration* of the failures are the main features that shape a failure model [270]. Thus, we characterize the failure models via the attributes *Failure Group Size* (FGS), *Inter Arrival Time* (IAT), and *Failure Exposure Time* (FET). FGS is the number of time- or space-correlated failures that occur approximately at the same time or within a short time span. IAT, also known as *Mean Time Between Failures* (MTBF), is the time between two occurrences of failure groups or bursts. FET is the time window during which time- or space-correlated failures affect the system; thus each burst occurs within the FET. See Figure 6.6 for an illustration of IAT, FGS, and FET.

In the following, we describe the two sets of input used for experiments with mRUBiS: *Deterministic* failure models and *probabilistic* failure models. A deterministic failure model defines the characteristics of the failure profile with scalar values and therefore, generates traces with deterministic characteristics for occurrences of failures. A probabilistic failure model, however, employs probability distributions to characterize a failure profile. Such a statistically defined failure model is either an outcome of fitting statistical models to recordings of real data, i.e., fitted to real data, or is synthetic, i.e., generated through computer programs instead of being composed through the documentation of the real-world events. The probabilistic failure models used for evaluation are further decomposed to *realistic* failure models that are fitted to real data, and *synthetic* failure models that are manual variations of the realistic failure models.

While the FGS and IAT of the failure traces based on probabilistic models vary during the trace, they remain constant in the deterministic traces. In the realistic failure profile models, $\mathsf{FET} > 0$ as failures take some time to propagate in the system; conversely, we assume $\mathsf{FET} = 0$ in the deterministic traces and all the failures in a group occur at once.

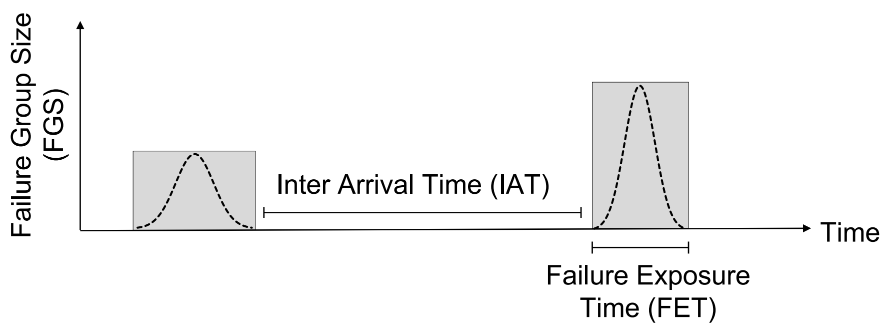


Figure 6.6: Failure group size (FGS), failure exposure time (FET), and inter arrival time (IAT) of failure models.

6.3.1.1 *Deterministic failure models*

We use deterministic failure models to generate four deterministic failure traces: X, X10, X100, and X1000. Each trace has a constant FGS, that is, the number of failures occurring together, of 1, 10, 100, and 1000 failures respectively. In the context of mRUBiS, different types of the adaptation issues, i.e., CF1-CF4 and PI1—see Section 2.5—are equally distributed for each trace as failures. For a FGS larger than one, that includes X10, X100, and X1000 traces, we assume that all the failures in one group, i.e., time-correlated failures, occur at once, thus FET is only an instant and is approximated to be zero. In the deterministic failure models, we assume a long enough IAT between occurrences of each group of failures such that all the current failures are handled by the current adaptation loop before the failures of the next group occur. The *failure density* of each trace is the overall number of failures injected by the trace. For example, considering four consequent executions of MAPE-K runs with X10, the failure density of the X10 trace is 40. Table 6.2 shows the general characteristics of the deterministic and probabilistic failure models introduced in this section.
Next, we present the two sets of probabilistic failure models, i.e., realistic and synthetic.

6.3.1.2 *Realistic failure models*

We employ three different realistic failure models provided by Gallet et al. [155] that are fitted to real-world failure traces. The models originate from different computer systems and differ in scale and volatility. Failure traces are derived from these failure models for a certain duration. Among the failure models introduced in [155], we selected three models with different failure densities and sizes of the source system: *Grid*5000, *LRI*, and *DEUG*. The *Grid*5000 model is based on the *Grid*5000 system in which a significant fraction of failures occurs in bursts. *Grid*5000 is an experimental grid environment with over 2500 processors and 1288 nodes [see 223] which is comparable to the size of mRUBiS with 100 shops, i.e., 1800 components. The event data for *Grid*5000 has been gathered over 1.5 years of monitoring [269]. The other two models, *DEUG* and *LRI*, are constructed from application-level traces of real enterprise desktop grids that contain bursts of failures and are collected over one month from about 100 (*DEUG*) and 40 (*LRI*) hosts [268].

All three failure models, conforming to probabilistic models principals, fit statistical distributions to IAT and FGS. Moreover, Gallet et al. [155] consider different window sizes for monitoring and detecting the failure occurrences in each model. This window size is equivalent to the FET illustrated in Figure 6.6. Thus, each burst occurs within duration of the FET. Gallet et al. [155] do not clarify how the failures are distributed within each burst, thus we assume that failures propagate following a normal distribution during each burst—see the normal distribution curves in each failure group in Figure 6.6. Table 6.3 lists the distributions proposed by [155] for the IAT and FGS for the three failure profile models along with the their FET.

We extracted a *short* and a *long* failure trace from each realistic failure model. As shown in Table 6.3, short traces include $n = 50$ occurrences of failure bursts while long traces include 1355, 2843, and 1678 bursts. Each of the traces has a different duration due to different IAT distributions in the corresponding failure model. The failure densities of the traces, i.e., the collective number of failures during the execution of the trace, is shown in Table 6.3 as well.

Table 6.3: Characteristics of realistic failure models (top) and generated traces (bottom).

	LRI	**DEUG**	**Grid5000**
FGS	LOGN(1.32, 0.77)	LOGN(2.15, 0.70)	LOGN(1.88, 1.25)
IAT (s)	LOGN(−1.46, 1.28)	LOGN(−2.28, 1.35)	LOGN(−1.39, 1.03)
FET (s)	100	150	250
No. of bursts n (short trace)	50	50	50
No. of bursts n (long trace)	1355	2843	1678
Duration (hrs) (short trace)	41.2	21.4	24
Duration (days) (long trace)	30	30	30
Failure Density d (short trace)	318	666	1116
Failure Density d (long trace)	7568	32895	25279

6.3.1.3 *Synthetic failure models*

To obtain a fair and meaningful comparison between the experiment results for the different failure models, we constructed three synthetic probabilistic failure models based on the *Grid*5000 model. The resulting traces share the same failure density $d = 1116$ as in the short trace of the *Grid*5000 in Table 6.3. The parameters of the *Grid*5000 failure model are modified to obtain variants with extreme characteristics; the synthetic variants allow for studying the impact of the extreme characteristics of the failure models on parameters such as scalability and optimality of different self-adaptive solutions. In addition, using more and extreme failure traces for the experiments allows us to evaluate the robustness of the solutions and their yielding results.

The short trace generated from the *Grid*5000 failure model includes $n = 50$ occurrences of failure bursts during 24 hrs— see Table 6.3. The synthetic failure models are constructed based on the *Grid*5000 (short trace) and are defined as follows: (1) the *Burst* model replicates the *Grid*5000 model with identical characteristics; (2) the *Uniform* model where failures are uniformly distributed, (3) the *Single* model with a single failure at each occurrence, and (4) the *Bigburst* model with only relatively large bursts. Based on each of the synthetic failure profile models we generated a *short failure trace* that includes failure arrivals for 24 hrs. The synthetic models and their short traces are presented in Table 6.4. Figure 6.7 shows the probabilistic FGS distribution for the *Grid*5000 model, henceforth Burst model; different FGS values that have been re-sampled to construct the synthetic models are marked in different patterns/colors in Figure 6.7.

Table 6.4: Characteristics of synthetic failure models (top) and their generated traces (bottom).

	Burst (Grid5000)	**Uniform**	**Single**	**Bigburst**
FGS	LOGN(1.88, 1.25)	N(22.85, 20.68)	1	N(238, 97.3)
IAT (s)	LOGN(−1.39, 1.03)	1728	77.4	N(3521.4, 5418.6)
FET (s)	250	250	N/A	250
No. of bursts n (short trace)	50	50	1116	6
Duration (hrs) (short trace)	24	24	24	24
Failure Density d (short trace)	1116	1116	1116	1116

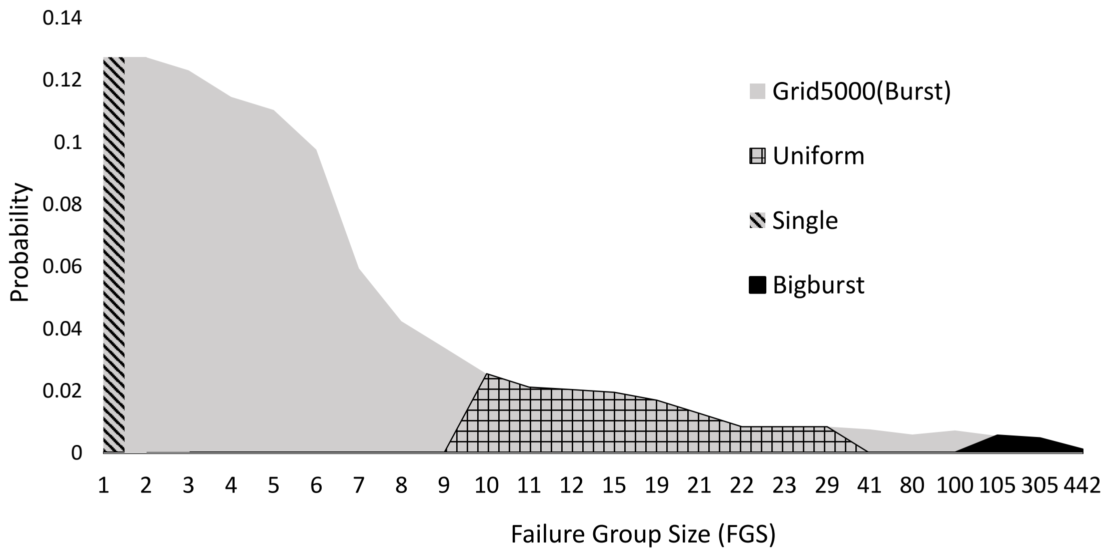


Figure 6.7: Probabilistic FGS distribution in *Grid*5000 (full curve) and sampled segments for constructing synthetic failure profile models.

Burst Model. The Burst model is identical to *Grid*5000, thus has the same statistical properties; the model is only listed here for consistency purposes. The description of the Burst (*Grid*5000) model and its generated trace is repeated in Table 6.4.

Uniform Model. The Uniform failure profile model is constructed based on the *Grid*5000 short trace—see the description of *Grid*5000 short trace in Table 6.3. Thus, Uniform has the same failure density $d = 1116$ and lasts 24 hrs. The model has a uniform statistical distribution for IAT and FGS—see Table 6.4 for the characteristics of the Uniform model and its trace.

To construct the Uniform model, we consider the original set of 50 failure groups extracted from the *Grid*5000 short trace as the set S. A normal sample distribution is extracted from S using statistical bootstrapping [94, 122]. For this purpose, we randomly re-sampled S using statistical bootstrapping and formed a new set S' of the mean values of each sample set. A normal distribution $N(\mu_{S'}, 2\sigma^2_{S'})$ is used to generate random values for FGS. The resulting set consists of uniformly distributed values within a certain margin extracted from the original set S. Applying this distribution, we generated a sequence of normally distributed values for FGS while keeping the same number of bursts as in the original *Grid*5000 trace. Thus, the Uniform model takes all the d failures within the 24 hrs and distributes them in n intervals, i.e., occurrences, by using the extracted uniform distribution. The IAT is the average of the IAT values in the *Grid*5000 short trace. Therefore, the Uniform model is a sequence of failure groups with normally distributed sizes which occur in equal intervals.

Single Model. To consider a naive failure profile model that—to the best of our knowledge [1]—has been used in the majority of the existing work on self-healing systems, e.g., [see 11, 75, 76, 116, 295, 334], we consider the Single failure profile model. In this model, failures are not correlated but arrive individually, thus $FGS = 1$—see Figure 6.7. In the failure trace extracted from this model, individual failures are equally distributed within the 24 hrs keeping the same failure density d as the original *Grid*5000

1 See our SLR on state-of-the-art in evaluation of self-healing systems [177].

short trace, i.e., 1116. The number of the occurrences of bursts, i.e., n, equals d since each failure group in the Single model includes exactly *one* failure—see Table 6.4 for the characteristics of the Single failure model and its resulting trace.

Bigburst Model. We also consider the other end of the spectrum, that is, occurrences of large failure bursts with 150 to 450 failures at each failure group. Similar to the construction of the Uniform model, we use statistical bootstrapping to extract a set from the original set S. To achieve large FGS values, only part of S that is above a certain threshold, i.e., FGS $\geqslant$ 100, is re-sampled for bootstrapping—see Figure 6.7. To keep the same failure density as the original *Grid*5000 short trace, i.e., 1116, the number of the failure groups are reduced to $n = 6$ since the failures occur only in large group sizes, thus the IAT increases accordingly. The IAT values in the Bigburst model are extracted via bootstrapping from the randomly re-sampled IAT values of the original *Grid*5000 trace where IAT $\geqslant 1000\,sec$—see Table 6.4 for the characteristics of the Bigburst failure model and its trace.

6.3.2 *Input Traces for* `Znn.com`

We consider two sets of input traces for experiments with `Znn.com`: *deterministic* traces and *realistic* traces. The traces include client requests from the web servers in `Znn.com`. The requests may result in different types of adaptation issues, i.e., `latency`, `lowQuality`, `overBudget` and, `underUtilized`. However, the input traces for `Znn.com` only include client requests and not the specific adaptation issues, the issues occur according to the system response to the input trace—see Section 6.2 for a reference of different issues and adaptation rules in `Znn.com`.

6.3.2.1 *Deterministic input trace*

The deterministic traces are constructed synthetically and include client content request from the servers in `Znn.com` with constant request arrival rate. Overall, we consider five different deterministic traces, i.e., TR_r, with r the request arrival rate per minute. For the five employed TR_rs, we consider $r = 1$, $r = 10$, $r = 100$, $r = 1K$, and $r = 10K$ requests per minute; each trace has a constant value for r, that is, the number of the requests that arrive together. Additionally, for traces that include more that one request per minute, i.e., $1 < r$, we assume that during an interval, all the requests arrive at once and in an instant, therefore, the IAT between two consequent groups of client requests is exactly one minute. The *load density* of each trace is the overall number of requests in the trace. For example, during 10 minutes simulation of `Znn.com` with the TR_{100} trace, in total 1000 client requests are injected to the system. Table 6.5 summarizes the general

Table 6.5: Characteristics of deterministic input traces for `Znn.com`.

	TR_1	TR_{10}	TR_{100}	TR_{1K}	TR_{10K}
Request arrival rate r (short/long trace)	1	10	100	1K	10K
Duration (min) (short trace)	60	60	60	60	60
Duration (hrs) (long trace)	24	24	24	24	24
Request Density d (short trace)	60	600	6K	60K	600K
Request Density d (long trace)	1,440	14,400	144K	1,440K	14,400K

characteristics of the deterministic traces used for Znn.com. In the experiments with Znn.com, for each TR_r, we consider a *short* (60 min) and a *long* (24 hrs) trace.

6.3.2.2 *Realistic input trace*

To construct realistic user behavior for accessing a cloud-based system such as Znn.com, we adapted a research dataset with online traffic, the daily traces of user requests from the FIFA WorldCup website [17], that is common in web analytics [85]. These traces are independent, day-by-day recordings of user website activity during the 1998 soccer championships. The reasons we used the realistic WorldCup dataset as the realistic input trace for Znn.com are twofold; first, the traces are considered as a benchmark for traffic in web analytics—see [17, 221, 434]). Second, the traces comply with the customary request arrival pattern in modern-day web applications, e.g., content delivery and video streaming platforms [see 66]), that are typically bursty, with periods of low load contrasting with occasional spikes caused by an important event. The traces contain the patterns for high-demand cloud systems as classified by Gandhiet al. [157] that include slowly varying, quickly varying, big spike, dual phase, and large variations.

Out of the 92 available traces, five were incomplete/corrupted; we selected nine traces (out of 87 remaining) for our experiments with Znn.com; the selected traces correspond to day 10, 20, 30, 40, 50, 60, 70, 80, and 90—see the plots illustrating the client request arrival patterns in Figure 6.8. The trace for each day contains timestamps representing IAT between two client requests, abstracting away the details of user requests to focus on their frequency. For our implementation of Znn.com to work, we scaled down each trace (keeping the trace pattern intact) such that it does not constantly exceed the maximum capacity of Znn.com in terms of its ability to serve requests per minute. Considering the size of ServerPool in our implementation of Znn.com, for the experiments with FIFA traces, we inject the system only with reasonable number of request. Thus, we equally scaled down the traces such that the maximum request arrival rate does not exceed 4000 requests per minute—see Day 60 in Figure 6.8.

Additionally, similar to the realistic traces in Table 6.5, we consider a *short* (Figure 6.8) and a *long* (Figure 6.9) instantiation of the FIFA traces. The *long* traces keep the original duration of the original trace, i.e., 24 hrs. To construct the *short* traces however, we scaled each trace (keeping the trace pattern intact) to the length of 60 minutes. We fixed the length of the traces to normalize aggregate utility values in the experiments for reward. We consider the *short* variant of all nine traces in Figure 6.8; out of the nine traces, however, we only consider the *long* versions for days 10, 50, and 90—see Figure 6.9. Long traces are also scaled down to the maximum of 4K request arrival rate in a minute.

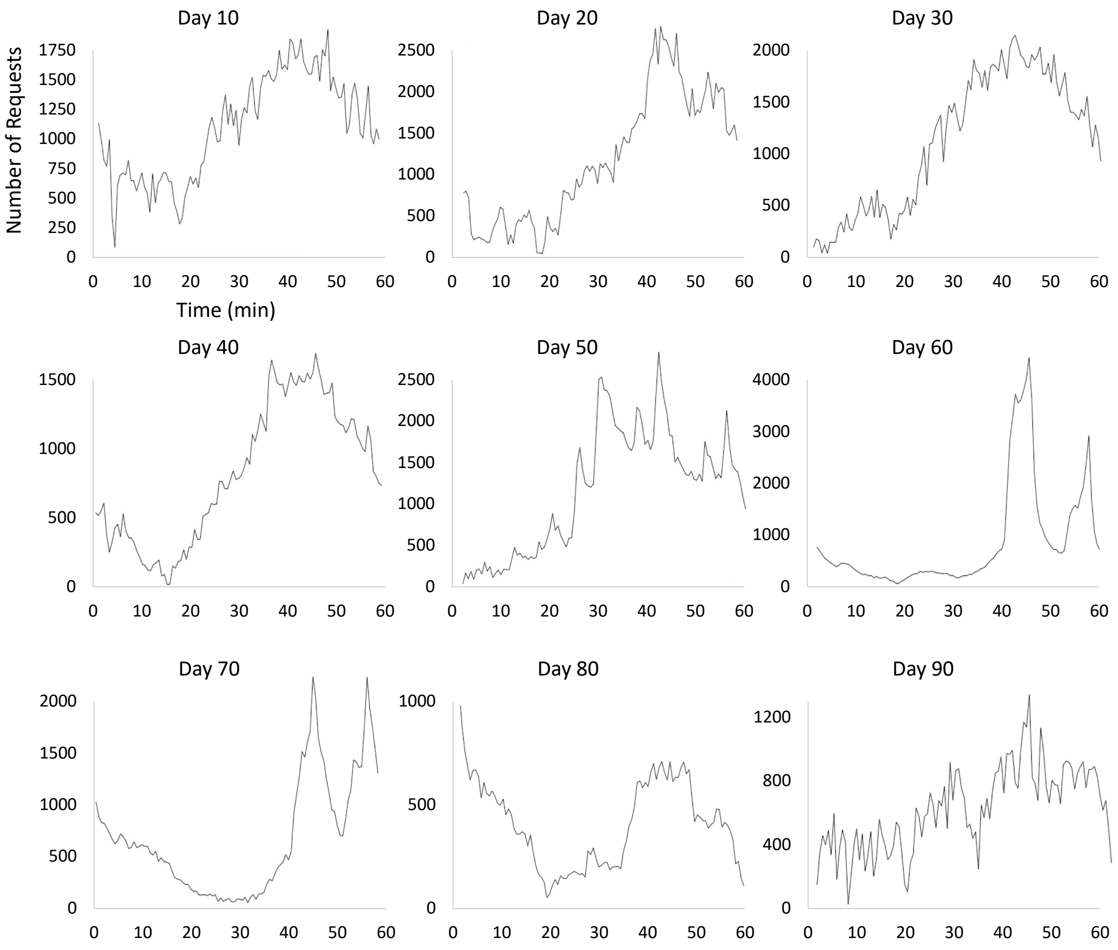


Figure 6.8: Plots of FIFA short traces.

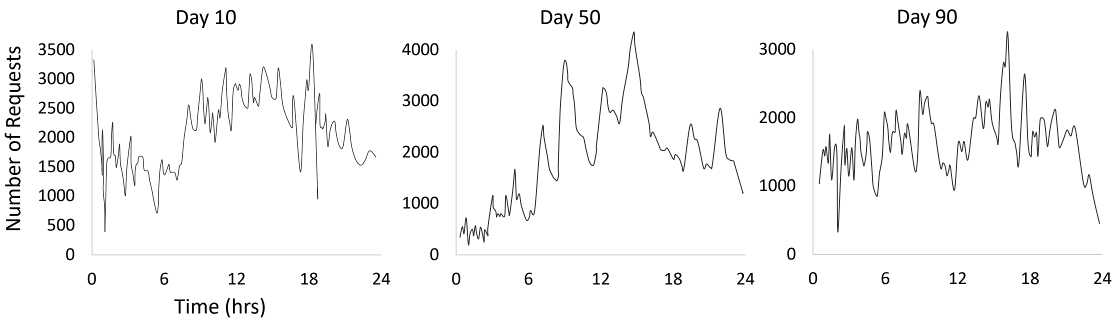


Figure 6.9: Plots of FIFA long traces.

7 EVALUATION OF VENUS

In this chapter, we evaluate Venus qualitatively and quantitatively, review the pieces of supporting evidence, and discuss how each helps to support the thesis claims introduced in Chapter 1. First, in Section 7.1, we discuss the implementation of our approach for incremental architecture-based self-adaptation and of the technical setting that we use to conduct experiments with Venus. Next, we introduce the alternative approaches for architecture-based self-adaptation in Section 7.2. Section 7.3 presents qualitative and quantitative evaluation of Venus using the mRUBiS application example in a comparative study with alternative solutions for self-adaptation. Section 7.4 includes evaluation of Venus using the `Znn.com` application example. In Section 7.5, we discuss the possible violation of the assumptions that are required for the validity of Venus and demonstrate the impact of the violations on the optimality and scalability of Venus. Finally, Section 7.6 presents the threats to the validity of the results and Section 7.7 summarizes the chapter and discuses the fulfillment of the requirements.

7.1 IMPLEMENTATION

Figure 7.1 shows a stack of elements constituting the technical setting that we use to implement Venus and run experiments. The implementation is in Java and based on the *Eclipse Modeling Framework* (EMF) [391]; EMF is an Eclipse-based code generation and modeling tool that provides the required MDE infrastructure for the model-driven engineering of self-adaptive software with RTMs. The implementation architecture is *generic*, i.e., it is not customized for one specific type of adaptable software (application), one specific self-adaptive property, or one specific (built-in) utility function. Being integrated in a MAPE-K feedback loop, Venus is realized via an external approach for self-adaptation where the adaptation engine dynamically observes and adjusts the software—see Section 2.3.2. Venus is embedded in an adaptation engine that employs a causally connected RTM as its knowledge. The adaptation engine uses RTMs of software architecture—see Section 2.4. Building on the MDE principles whereby creation and runtime evolution of causally connectedRTMs are supported, Venus uses models of software architecture at runtime for maintaining the adaptable software.

7.1.1 *Adaptable Software*

In order to achieve genericity, the *Adaptable Software* in Figure 7.1 is not limited to one specific adaptable software, thus supporting a broad spectrum of applications. The adaptable software platform represents the runtime environment of an executable software system. The *Application* and its corresponding *Input Trace(s)* are provided by the developers. The input trace simulates the operation condition for the application. Via adding a control feedback loop on top, the application is then instrumented to be managed as an adaptable software. We enforce no restrictions on the technology

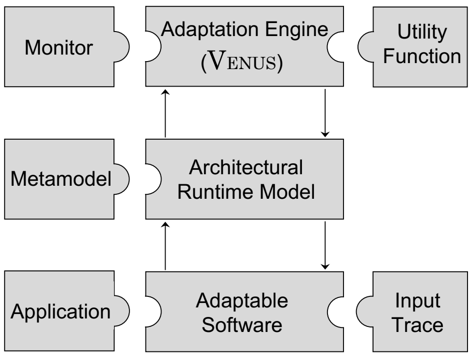


Figure 7.1: Architectural decomposition of VENUS implementation.

used to develop the application, however, we require the application to be provided as a component-based software consisting of multiple components and interconnections. Service-oriented architectures are examples of software systems with component-based designs [128]. The provided application, deployed on the corresponding application server, should be instrumented with sensors and effectors making the application observable and adaptable during runtime. Consequently, the application supports parameter and structural adaptation as its individual components, connection (link) between the components, and attributes of the components and links become subject to observation and adaptation.

We use an implementation of the mRUBiS simulator as an Eclipse plug-in that is coupled on top of EMF. For `Znn.com`, we simulated the architecture of the application example by means of the EMF models, i.e., without having to run the underlying causal connection and the adaptable software, and executed it in combination with the input traces—see Chapter 6 for more technical details about the application examples and the input traces. For both application examples, we use *injectors* that implement the injection of an issue into the *Architectural Runtime Model*. For instance, an injector can introduce a failure into the model for a self-healing scenario. Consequently, an injector simulates the occurrence of an issue such as a failure in the adaptable software and the synchronization of this issue from the software to the model. In our implementation, we are concerned with a higher level of abstraction such that feedback loops may operate on a higher-level, platform-independent RTMs to perform self-adaptation. Owing to the the causal connection between the system and its RTM, enabled by MDE tools such as EMF, we bridge the abstraction gap between the low-level, platform-specific API and the higher-level, platform-independent RTM. The causal connection between the model and a system refers to synchronizing an RTM and a running software system such that changes in one of them are reflected in the other—see Section 2.4.

7.1.2 *Architectural Runtime Model*

As shown in Figure 7.1, any user-defined *Metamodel* representing an application at a desired level of abstraction can be used to instantiate an RTM. The causal connection, supported by the sensor and effector API, is automated by the EMF tool and does not concern our implementation. Thus, for any supported application as the adaptable software, enabled by MDE, an *Architectural Runtime Model* of the software system is made available and causally connected to the adaptable software. We envision that, ideally, the adaptable software directly provides its sensor and effector capabilities as an RTM instead of a code-based API. The architectural RTM is an instance of the provided *Metamodel*, at the architecture abstraction level, and is used to monitor the adaptable software. Moreover, changes to the adaptable software can be generated by modifications of the corresponding architectural RTM. The causal connection synchronizes the RTM with the adaptable software by using the sensors and effectors. The synchronization is in both directions, i.e., from the adaptable software to the RTM and vice versa.

In the context of mRUBiS, we extended the (EMF-based) *CompArch* metamodel of the system provided by Vogel [414]—see Figure 2.9. The metamodel describes the architecture of the adaptable software in an abstract manner leaving out details that do not concern self-adaptation—see Figure 2.8 for a metamodel of mRUBiS. Similarly, for `Znn.com`, as shown in Figure 6.3, we defined a metamodel capturing the architecture of the adaptable software. The metamodel is generated based on (but not identical to) the `Znn.com` description provided by Cheng et al. [see 91].

7.1.3 *Adaptation Engine*

The topmost layer of the implementation layout in Figure 7.1 includes the *Adaptation Engine* that is implemented according to the MAPE-K feedback loop blueprints—see Section 2.3.2. Venus is integrated in a MAPE-based adaptation engine as described in Section 4.3.1. The adaptation engine uses the architectural RTM as its knowledge that is shared between its four activities. The adaptation engine implementation includes Java code snippets wrapped around the SDM tools[1] including an editor, an interpreter, and a debugger for the SDs—see Section A.1 for an introduction to the SDM tool. The source code for the initialization of the SDs in Java are included in Section A.2.

The SD interpreter is used to parse the architectural RTM and check if any adaptation issue exists. The adaptation engine is implemented in Java where each of the MAPE activities are realized via SDs—see Figure 6.4, Figure 6.5a, and Figure 6.5b for Analyze, Plan, and Execute SDs, respectively, for an `overBudget` issue in `Znn.com`.

The *Monitor* activity as shown in Figure 7.1 is implemented independently and external to Venus—see Section 4.3.2. To make the monitoring runtime-efficient, we use a change tracking mechanism enabled by the notification feature of the EMF, which provides notifications about the individual changes of any EMF-based models [391]. Thus, the monitoring activity consumes the EMF events notifying about changes of the RTM—see the source code in Section A.2.

The implementation of the remaining MAPE activities then consists of Java code snippets that consumes the change events for the architectural RTM. Based on these events, relevant checks are conducted via invoking the SD interpreter to execute the corresponding SDs. Finally, once the execution of the adaptation is completed, we compute the overall utility of the RTM, i.e., the architecture, and validate the model to check for any unhandled issue—see Section A.2 for the source code.

Venus employs utility functions to evaluate the desirability of each adaptation and steer the planning accordingly—see Figure 4.1 for an overview. Utility functions, either manually engineered and hard-coded in the adaptation logic at design time (see Section 3.2), or acquired as prediction models and plunged in to the system before or

1 Story-Driven Modeling (SDM) Tools: `http://www.mdelab.de/mdelab-projects/story-diagram-tools`– accessed 18 March 2023.

during its execution (see Section 3.3), can be defined by externally, thus independently of Venus—see Figure 3.1 for an overview. As shown in Figure 7.1, the implementation of the adaptation engine and Venus, concerning the required *Utility Function*, is generic and supports any plugged in utility function. In Table 3.2, we showed four different utility functions subscribing to different mathematical complexity classes for mRUBiS. Similarly, for any application of interest, a set of developer-provided utility functions are supported in the implementation of Venus.

In our implementation, injecting adaptation issues to the RTM triggers an execution of the adaptation loop. Different *injection strategies* as well as *Input Traces* may be provided by the developers—see Figure 7.1. A single run of the adaptation loop in our implementation consists of the following steps: (i) the monitoring activity consumes the input trace provided by the developer to simulate the runtime condition of the adaptable software; consequently, the changes obtained by the monitoring activity trigger an adaptation; (ii) the feedback loop performs the remaining adaptation activities to identify and handle the potential adaptation issues; in this context, issues are handled by adapting the architectural RTM. In a Models@run.time approach, an adaptation is prescribed and previewed in the RTM before it is executed on the adaptable software through the causal connection. We follow the same approach in the implementation. We emulate the adaptable software and the causal connection such that the adaptation engine can operate only on the RTM and regardless whether the actual adaptable software with the causal connection or a simulator is used.

7.2 ALTERNATIVE SOLUTIONS FOR ARCHITECTURE-BASED SELF-ADAPTATION

Solutions to self-adaptation of software systems can be categorized in various ways [275, 367]. In the experiments to evaluate Venus, we structure the spectrum of the considered self-adaptive approaches with respect to the following: (i) event- versus state-based processing of the system changes; (ii) employed policies for expressing and operationalizing the self-adaption goals during planning; (iii) temporal aspect of the decision-making process—see Section 2.3.2.

Static Solution. This approach leverages a deterministic, rule-based policy and employs ECA rules that directly map specific event combinations to the adaptation plans, i.e., (sequence of) adaptation rule(s). Moreover, it conforms to the principles of a static adaptation policy introduced in Section 2.3.2. The decision-making operates based on the design-time preferences whereby the adaptation rules are mapped to the states, i.e., conditions, based on design-time estimates for the resulting state. Therefore the estimations for the desirability of the outcome configuration are agnostic to using any runtime observations $\bar{o}$. The decision-making in the Rainbow framework [90] is an example of a static adaptation policy as it only uses configuration attributes $\bar{c}$ to *estimate* the expected utility for its actions. However, due to their minimal runtime computation, static solutions for adaptation do not introduce considerable runtime overhead during planning [322]. The considered static approach, henceforth *Static*, uses deterministic priorities for adaptation rules agnostic to their impact on the overall system utility at runtime. In the context of our application examples, the costs and utilityIncrease of the adaptation rules are defined at design time, hence, a rule is assigned to each issues in a deterministic manner. The utilityDrop caused by each adaptation issue is also estimated at design time suggesting a deterministic order to addressed the issues at runtime.

Solver Solution. The second alternative approach has utility-based policy formalism that uses IBM ILOG CPLEX[2] optimizer [216] for selecting the adaptation rules during planning. The approach, henceforth *Solver*, requires an objective function and a set of constraints (if applicable) as inputs. The objective function, e.g., the overall utility function that quantifies the desirability of system states, captures the system goal satisfaction. Next, taking the system constraints into account, *Solver* plans the adaptation such that the plan optimizes the objective function. In the context of mRUBiS and `Znn.com`, *Solver* employs the utility functions in Table 3.2 (Linear) and Equation 6.1 as its objective functions for the sequence of rule applications. The tasks of assigning proper adaptation rules to the adaptation issues and ordering them for execution are defined as optimization problems for *Solver*. Moreover, regarding the temporal aspect of *Solver*'s decision-making, similar to VENUS, it conforms to the principles of a dynamic decision-maker and is concerned with the runtime assessment of the conditions affecting the adaptable software, e.g., impact of the adaptation rule on the system utility and cost of adaptation—see Section 2.3.2. Consequently, while providing for more control flexibility at runtime, as discussed by McKinley et al. [302], *Solver* faces additional challenges such as finding optimal or partially optimal solutions for decision-making problems, dealing with uncertainty and incompleteness of the observations available at runtime, and addressing the scalability and fault-proneness of its decision-making mechanism. rather

Venus Solution. VENUS computes the impact of different adaptation rule applications at runtime using a utility function. To resolve an issue, it selects the rules with largest impact on the overall utility; in case of identical impacts, rules yielding the lowest costs are selected. The execution order of the adaptation issues is determined by the ratio of utilityIncrease/costs in mRUBiS and utilityIncrease in `Znn.com`, which are all dynamically computed based on the runtime observations regarding the affected element by the issues. VENUS is event-based and uses change events as adaptation triggers; the changes events result from updates of the RTM, i.e., of the state. VENUS also supports the state-based execution of the adaptation engine as it holds a global view on the adaptable software through maintaining RTMs that represent the state of the system.

Static and *Solver* have different planning phases while they are both integrated in an adaptation engine that is executed incrementally —as in VENUS. Therefore all three approaches listed above have similar and incremental monitoring, analysis and execution phases. For instance, *Solver* uses the optimizer only to solve the planning problem of selecting the best adaptation rules and ordering them for execution. Possible ways of how to monitor and modify the system/RTM are completely pre-defined and identical for all three approaches. Moreover, analogously to VENUS, both *Static* and *Solver* solutions support event-/state-based adaptation as they operate based on consuming change events and maintain RTMs that capture the state of the system. In order to investigate the impact of the incremental execution of the activities in the adaptation engine that integrates *Static*, *Solver*, and VENUS, we employ a baseline approach titled *Batch*. The approach is integrated in an adaptation engine that does not capture the change events, therefore, is executed in a state-based manner.

Batch Solution. The activities of the feedback loop in this approach, henceforth *Batch*, work in a state-based manner on the RTM that captures the state of the system. Thus,

2 The optimizer is suitable for Mixed Integer Programming (MIP). A MIP is a problem where some of the decision variables at the optimal solution are constrained to be integer values. Integer variables make an optimization problem non-convex, and therefore, non-trivial to solve—see [358].

Table 7.1: Positioning three solutions in the landscape of self-adaptation solutions.

Property of Interest	**Solution**			
	Static	*Solver*	Venus	*Batch*
Policy	Rule-based	Utility-based	Rule-& utility-based	Rule-based
Temporal aspect of decision-maker	Design-time	Runtime	Runtime	Design-time
Execution of adaptation loop	Incremental	Incremental	Incremental	Non-incremental
Event- /state-based	Both	Both	Both	State-based

the monitoring activity does not process any change events, rather evaluates the complete RTM to identify whether changes of the model have occurred and analysis is required. The analysis, in turn, checks the complete RTM for any symptoms of the issues and annotates them in the model. The planning activity searches the complete model to detect the annotations made during analysis. For each of them, it changes the model according to a predefined, deterministic, rule-based adaptation policy to address the corresponding issue (similar to *Static*). The execute activity removes the annotations from the model and terminates the feedback loop. An overview of the considered approaches is given in Table 7.1.

All four solutions are implemented according to the technical setting presented in Section 7.1; they use an architectural RTM of the adaptable software that is maintained by the EMF tool and the activities of the adaptation engine are implemented based on the SDM tool.

7.3 QUALITATIVE AND QUANTITATIVE EVALUATION WITH MRUBIS

In this section, we evaluate Venus through several comparative studies using mRUBiS application example from Section 2.5 and Section 6.1. We compare Venus to three alternative self-adaptation solutions introduced in Section 7.2. Through the experiments, we investigate three features of Venus that are captured by individual contributions in Section 1.4: *runtime performance* and *scalability* through *incrementality* (**C**1), *optimality and runtime overhead* (**C**5), and *robustness* (**C**6). The qualitative evaluations in this chapter are according to the evaluation methodology introduced in Section 6.3 and provide coverage of a wide spectrum of the input space for self-adaptive software systems (**C**8).

The deployment of the four solutions, i.e., *Static*, *Solver*, Venus, and *Batch* follows the implementation architecture presented in Section 7.1. In the context of *Batch*, the adaptation engine in Section 7.1 is executed in a state-based manner, thus is non-incremental. The different solutions operate on an architectural RTM of the employed application, i.e., mRUBiS, while the input traces are injected—see Figure 7.1 for a reference. In the experiments with mRUBiS, the RTM is maintained by a simulator that emulates the running system and the causal connection. Thus, connecting and synchronizing the RTM and the running system does not concern the self-adaptation solutions.

All four self-adaptation solutions follow the MAPE-K blueprint and therefore consist of four adaptation activities. However,they differ in the planning phases as discussed in Section 7.2. At the beginning of each experiment run, we playback the relevant input trace; for this purpose, we customize the mRUBiS simulator with injectors that introduce adaptation issues to the RTM. One experiment run consists of three steps: (1) the simulator injects adaptation issues to the RTM, (2) an execution of a MAPE-K loop is initiated during which the adaptation solution analyzes the RTM, plans an adaptation, and executes the changes on the architectural RTM, (3) the simulator analyzes the adapted RTM for success of the adaptation and the overall utility of the system.

The experiments for scalability and optimality with mRUBiS share the same experiment design; we used mRUBiS with the Linear utility function in Table 3.2 that is manually engineered in the code. Specifically, mRUBiS can host different number of tenants (1 to 1000), each containing 18 components with different criticality and connectivity values. The utility of a tenant is the sum of the sub-utilities of all the components in the tenant—see Section 6.1.1 for a reference on self-healing and self-optimization with mRUBiS. To determine the injected traces of failures, we use deterministic, realistic, and synthetic failure models introduced in Section 6.3.1. We inject issues of type Critical Failures (CF) and Performance Issue (PI) to mRUBiS which are the negative patterns that affect the system. The rule set $\mathfrak{R}$ includes the adaptation rules; each rule has three attributes: costs, utilityIncrease, and ratio—see the metamodel of mRUBiS in Figure 2.8 for details of the supported issues, adaptation rules, and their attributes. Costs refers to the expected execution time of the rule, utilityIncrease is the impact on the utility when applying the rule, and ratio is defined as utilityIncrease/costs.

7.3.1 *Evaluation of Runtime Performance and Scalability*

To provide the supporting evidence for **C1**, i.e., the runtime performance and scalability of Venus (see Section 1.4), we run Venus on a spectrum of architecture sizes and input traces, providing sensitivity analysis of the results. We compare Venus against the alternative solutions on different sizes of the mRUBiS architecture and measure the execution times. Consequently, we can investigate the runtime performance and the scalability of the adaptation loop for increasing sizes of the architecture; in addition, the number of the adaptation issues are increased to evaluate the system under more intense conditions.

mRUBiS simulates the behavior of a marketplace that hosts multiple shops as tenants; each tenant has 18 components and an architecture as shown in Figure 6.1. Using mRUBiS with sizes larger than one tenant implies having an RTM that includes multiple tenant architectures in one model. Thus, for each additional tenant on the marketplace, the architectural RTM contains additional 18 components. Consequently, the adaptation engine has to manage now a larger system and therefore operate on a larger RTM. Thus, scalability of the self-adaptation solution is key in handling large architectures.

Experiment Design. To measure the performance, we execute the four solutions on top of the simulator (see Figure 7.1) and consider the cycle of (1) injecting adaptation issues to the RTM, (2) executing the MAPE-K loop to handle the issues in the model, and (3) validating and evaluating the architecture after the adaptation. Step (1) is performed by the simulator and based on a developer-defined *injection strategy*. The injection strategy determines the type and the number of the CFs and PIs that are injected as well as the affected elements. The solutions perform step (2), and step (3) is achieved via an external code snippet that checks the the adapted RTM with respect to the success and the overall utility of the adaptation. The analysis performed in step (3) provides feedback whether the solutions actually addressed the injected CFs or PIs. Thus, for runtime performance of the solutions, we measure only the times for step (2). In the experiments with all four approaches, we report the complete loop execution times

Table 7.2: Average MAPE-K loop execution time (ms).

	X				X10			
# Comp.	*Batch*	*Static*	Venus	*Solver*	*Batch*	*Static*	Venus	*Solver*
18	97.2	3.1	3.3	7.4	101.1	26.6	30.6	71.9
180	1500.1	3.1	3.3	7.2	1504.8	27.9	31.5	77.1
1800	18043.8	3.2	3.3	7.3	18048.2	26.1	31.1	78.2
18000	202435.7	6.2	7.4	12.4	202439.2	37.1	41.7	99.6

	X100				X1000			
# Comp.	*Batch*	*Static*	Venus	*Solver*	*Batch*	*Static*	Venus	*Solver*
18	–	–	–	–	–	–	–	–
180	1511.3	42.1	46.6	246.5	–	–	–	–
1800	18055.9	45.4	58.3	251.1	18070.1	183.1	189.7	3341.5
18000	202446.4	63.6	68.2	310.7	202461.8	350.8	400.1	3840.4

while in the experiments where the *Batch* approach is excluded, we only report the planning times; the reason is that *Static*, *Solver*, and Venus have the same monitoring, analysis, and execution phases and are only different with respect to their planning.

For one measurement run, we execute one solution (*Static*, *Solver*, Venus, or *Batch*) together with one failure trace from Section 6.3.1 for one size of the mRUBiS architecture (1, 10, 100, or 1000 tenants, that includes, 18, 180, 1800, or 18000 components). For the experiments, we only consider meaningful combinations of the architecture size and number of the failures. Thus, we do not inject a large number failures to small architectures, e.g., we do not present any data where more than 10 (100) failures occur in a system with 18 (180) components. Within one measurement run, we repeat the cycle of steps (1), (2), and (3) 300 times or until the measurements stabilize, i.e., the standard deviation of the execution time is less than five percent of the measured average execution time. We report the mean values of the measurements. The measurements were performed on one machine[3] and follow the benchmarking guidelines in [378].

7.3.1.1 *Runtime performance of adaptation engine*

To compare the complete loop execution time and scalability of the solutions, we executed them on mRUBiS with 18, 180, 1800, and 18000 components. We used the X, X10, X100, and X1000 failure traces from Section 6.3.1.1. One experiment run includes a *single* MAPE loop run, one self-adaptation approach, one size of the architecture, and one input trace. Table 7.2 shows the average MAPE-K loop execution time of *Batch*, *Static*, Venus and *Solver*.

Compered to the other three approaches, *Batch* has significantly larger loop execution times; the difference becomes more severe as the size of the model increases. For X1000 and 18000 components *Batch* has a complete loop execution time equal to 3.4 minutes (202461.8 ms) while the worst execution time among the remaining three approaches belongs to *Solver* (for the same experiment; 18000 components - X1000) with 3.8 seconds (3840.4 ms). Another observation is that the execution times of the approaches that leverage incremental executions of the MAPE activities, i.e., *Static*, Venus, and *Solver*, are either almost insensitive to the increasing size of the architecture, or are affected

3 All experiments and simulations have been conducted on a machine with OS X 10.10, Intel processor 2.6 GHz core i5, and 8 GB of memory.

only with a linear rate while the size of the architecture increases exponentially; the latter applies only to the experiments with 18000 components. The results in Table 7.2 suggest that the incremental execution of the adaptation engine in *Static*, VENUS, and *Solver* results in scalability of the adaptation solutions—see the changes in the execution times of the approaches as the size of the model increases. The good scalability of these three solutions is caused by the event-based processing of the changes; the adaptation engine never processes the whole model, rather only the events that capture a very small amount of information obtained once from the model in the monitoring step, e.g., the identifier and life cycle state of a single component. Technically, the analysis and planning steps only partially refer to segments of the model (state-based processing), that is, parts that are referred to by the change events; therefore, they do not access the whole model and are not affected by the size of the model.

Consequently, we may observe that state-based, non-incremental solutions for self-adaptation such as *Batch* do not scale for large systems which calls for incremental approaches to process large models. Such an incremental approach is enabled by supporting event-based processing of the changes, analogously to the proposed incremental adaptation engine in this thesis that integrates VENUS. In this context, EMF itself suffers from scalability issues when querying the contents of large models [see 40] or modifying large models. This explains the almost stable loop execution times in the incremental approaches for models with 18-1800 components in Table 7.2 and the increased execution times for 18000 components. Therefore, those solutions that process more often and extensively the EMF-based model might suffer from such issues. This applies to all four solutions as all of their operations process (parts of) the model.

While the loop execution time in *Batch* heavily depends on the size of the architectural model as the complete model is searched to detect the changes, the number of the failures in the models, characterized by the input trace, does not affect the performance of *Batch* nearly at the same scale as the size of the model. For example, for 18 components, while the execution time of VENUS increases by 827% from the experiments with X to the experiments with X10 (3.3 to 30.6), the execution time of *Batch* increases only by 4% (97.2 to 101.1). *Static* and *Solver* show similar performance patterns as VENUS. For comparisons between the three incremental approaches we refer to the following section since the differences are attributed to the planning activity as they share the same monitoring, analysis, and execution phases.

We exclude *Batch* from the rest of the experiments in this chapter for the following reasons; (1) *Batch* only serves as a baseline approach to investigate the impact of incremental execution of the adaptation engine on the performance and scalability of the approaches; (2) the planning time in *Batch* is significantly affected by its state-based execution, rather than its employed decision-making policy which impedes a comparison of the policy performance in the individual solutions; (3) *Batch* uses the same deterministic, rules-based policy for decision-making as *Static*, thus the same observations regarding the optimality and reward can be made from the decision-maker in both approaches; (4) the excessive runtime overhead of *Batch* renders the assessment regarding the optimality and reward inclusive as the extreme performance degradation preempts any (potentiality) sub-optimal impact that may be attributed to the decision-maker.

Table 7.3: Average planning time (ms).

	X			X10			X100			X1000		
# Comp.	*Static*	VENUS	*Solver*	*Static*	VENUS	*Solver*	*Static*	VENUS	*Solver*	*Static*	VENUS	*Solver*
18	0.8	0.9	5.0	10.4	14.4	55.7	–	–	–	–	–	–
180	0.7	0.9	5.0	9.7	13.6	59.1	14.2	17.7	219.5	–	–	–
1800	0.6	0.7	4.8	10.6	13.5	58.2	13.8	26.7	211.1	54.5	60.1	3216.6
18000	0.7	0.7	4.9	10.1	13.9	71.9	21.8	26.4	271.5	127.8	171.3	3611.9

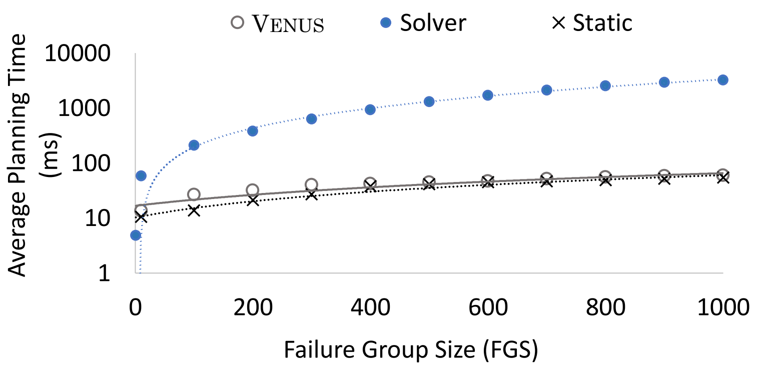
7.3.1.2 *Runtime performance of planning*

Single MAPE-K loop run. To compare the planning time and scalability of the solutions, we executed them on mRUBiS with 18, 180, 1800, and 18000 components; X, X10, X100, and X1000 failure traces are used. One experiment run includes, a *single* MAPE loop run, one self-adaptation approach, one size of the architecture, and one input trace. Table 7.3 reports the average planning time of *Static*, VENUS and *Solver* during the same experiments shown in Table 7.2.

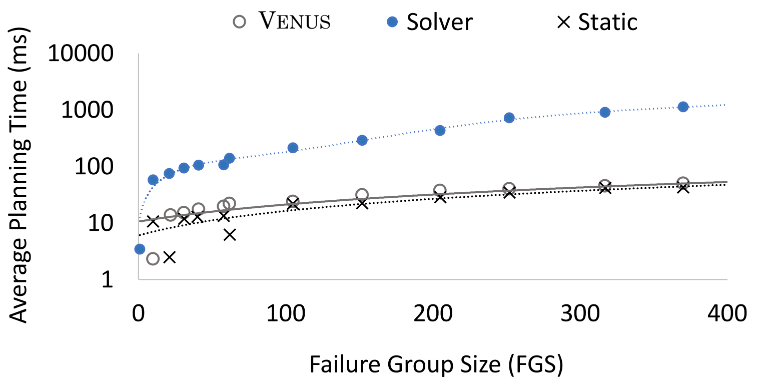
At the first glance, comparing the results in Table 7.2 to Table 7.3 suggests that for the three incremental approaches, the planning time constitutes a large fraction of the loop execution time, e.g., 94% in *Solver* for X1000 and 18000 components. As the number of failures affecting the system increases in Table 7.3, the planning time typically increases as well. However, the experiments suggest that the increase of the planning time is more visible for *Solver* that has to solve larger optimization problems. For all the combinations of numbers of the failures and architecture sizes in Table 7.3, we consider the planning time of *Static* as the *base value*; the planning time of *Solver* is at least 447% (10 failures, 18 components) and at most 5802% (1000 failures, 1800 components) larger than the *base value*. In case of VENUS, the minimum difference compared to the *base value* is 9% (1 failure, 18000 components) and the maximum is 92% (100 failures, 1800 components). Our experiments confirmed that while for a given architecture with failures, both *Solver* and VENUS reach the same final configuration, i.e., adapt the system similarly, *Solver* has an extreme planning overhead in terms of execution time for the planning phase for large numbers of failures and large architectures—see planning time of *Solver* for X1000 and 18000 versus that of VENUS in Table 7.3.

Figure 7.2a shows the planning time of the three approaches for the deterministic failure traces. Note that the vertical axis is in logarithmic scale. To achieve a more precise regression, we added additional data points to the measurements presented in Table 7.3. For this purpose, we constructed additional deterministic failure traces with FGS of 200, 300,..., 900 from the deterministic failure models and repeated the experiments with new traces. The measured data points in Figure 7.2a are visualized as markers and the regression lines represent the best fitting curves between the data points. As the number of the adaptation issues in one MAPE loop increases, the planning time for *Solver* has a polynomial growth while both *Static* and VENUS maintain a linear growth of their planning times. The results suggest that in contrast to *Static* and VENUS, *Solver* does not scale for large architectures and large FGSs.

The deterministic traces in experiments with single MAPE loop provide for a systematic evaluation during which runtime performance of the approaches is investigated for various FGSs and different sizes of the architecture. However, the results are inconclusive and serve only as preliminary evidence for runtime behavior of the self-adaptation



(a) Single MAPE execution, deteremintsic failure traces are used.



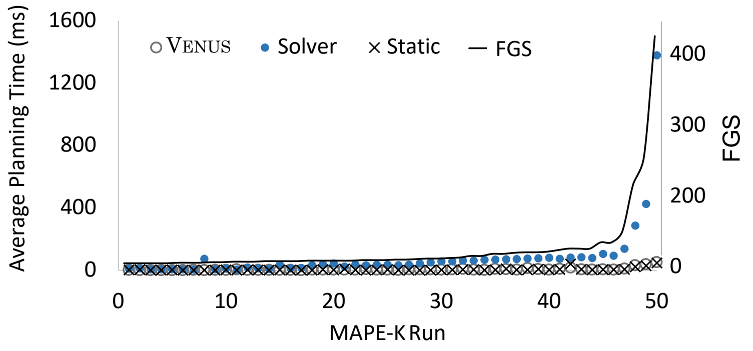
(b) Multiple MAPE executions, realistic and synthtic failure traces are used.

Figure 7.2: Planning time of self-adaptation approaches.

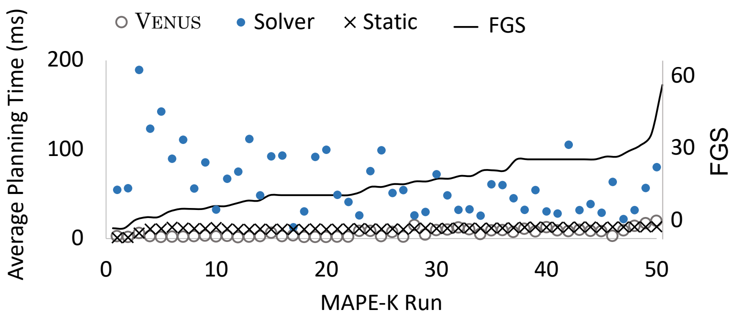
solutions and the reason is twofold; first, as depicted in Table 6.2, deterministic failure traces assume long enough IAT between two groups of failures. Consequently, the MAPE loop has enough time to handle all the issues currently affecting the system before the next group of failures occur. However, there is no guarantee that this assumption always holds and it is a simplification of realistic failure traces. Second, the results of the evaluation are difficult to generalize as they are only based on a single deterministic failure model. In addition, this failure model is deterministic, thus only captures one possible future for the simulated system and fails to cover a large and representative spectrum of its operation input space. We have shown in our work [177] that employing one failure trace as input for evaluation of self-healing systems only supports a single experiment run and does not justify any claim on the robustness and generality of the results. Generic credibility claims are only justified if the results are tested for robustness in the presence of a large spectrum of the input space during multiple reproducible experiments.

Multiple MAPE-K loop runs. In the following, we investigate whether using the traces extracted from the *probabilistic* failure models, i.e., realistic traces from Table 6.3 and synthetic traces in Table 6.4, confirm the scalability results that we obtained for the *deterministic* model earlier. We used mRUBiS with 100 shops (1800 components). The results are as follows:

Figure 7.2b shows the planning time for VENUS, *Static*, and *Solver* for the six *short traces* listed in Table 6.3 and Table 6.4. The traces based on *LRI*, *DEUG*, *Grid*5000, and Uniform failure models invoke 50 executions of the MAPE loop, while the Single model



(a) Short trace from Burst model is used.



(b) Short trace from Uniform model is used.

Figure 7.3: Planning time during 50 MAPE-K executions for short traces from Burst (top) and Uniform (bottom) failure models.

generated a trace with 1116 and Bigburst model generated a trace with only six bursts—see Section 6.3.1 for detailed description of the employed short traces. For each of the three solutions, we consider the measurements from all six traces collectively and show the average planning time for the observed FGSs independent of the constituent failure trace. The vertical axis in Figure 7.2b is in logarithmic scale. These results are the averages of the planning time measurements in *ms* over 300 repetitions for each of the six traces.

Employing failure traces of the Single or Uniform model results in a large population of data points, i.e., planning time measurements, for FGS < 50—see Figure 6.7 for FGS distributions in Uniform and Single models. Therefore, in Figure 7.2b, to avoid optimizing the regression curve for the range of $1 \leqslant \text{FGS} < 50$, we randomly sampled the data points in the range FGS < 50 for all three approaches. The measured data points in Figure 7.2b are visualized as markers and the regression lines represent the best fitting curves between the points. Consistent with the results for the deterministic failure traces in Figure 7.2a, compared to *Solver*, Venus has a lower overhead in terms of planning time; Venus has similar planning times as *Static*, whose runtime planning efforts are almost negligible—see definition of the static decision-making in Section 2.3.2; the observation also holds for large FGSs. Both *Static* and Venus have linear growth in planning time as the FGS increases. However, the planning time of *Solver* increases with a polynomial gradient as the FGS increases. Therefore, we can confirm the tendency observed for the synthetic failure traces in Figure 7.2a by the results shown for the probabilistic failure traces in Figure 7.2b; *Solver* does not scale well in contrast to the other two approaches, while VENUS and *Static* behave closely in terms of their planning time for different FGSs.

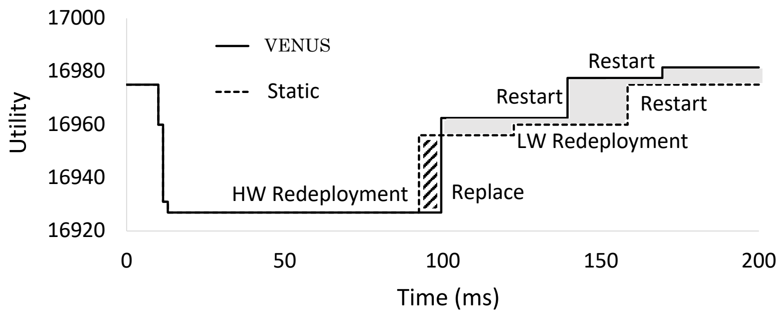
Figure 7.3 shows the planning time of the approaches for short traces generated from the Burst (top) and Uniform (bottom) models (see Table 6.4). The measured data points are shown as markers while the black curves in both charts indicate the number of the issues, i.e., FGS, for each data point. Once again, the results confirm the similar runtime performance of VENUS and *Static* for both traces. In Figure 7.3a, the last MAPE-K run includes 436 issues which results in a large planning overhead in *Solver*, while the planning times in *Static* and VENUS are only slightly affected, i.e., less than 5%.

7.3.2 *Qualitative Assessment of Reward and Optimality*

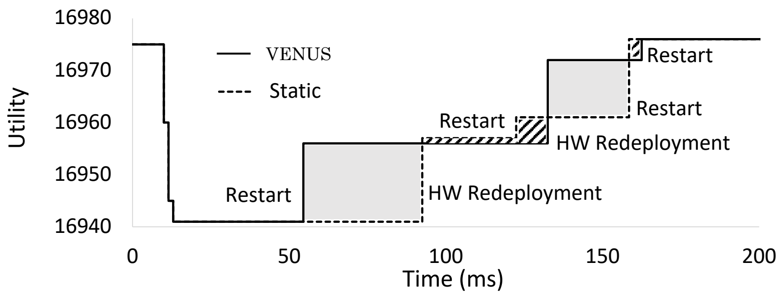
In the following, we present a set of qualitative experiments through which we qualitatively assess the reward and optimality of VENUS (**C**5) in a comparative study using the mRUBiS application example. The employed mRUBiS simulator contains 100 tenants, i.e., 1800 components. We use the deterministic failure traces in Section 6.3.1.1 to inject failures to the simulator. We compare the runtime decision-making and performance of VENUS against the ones in the alternative solutions in terms of utility. At each point in time, we can compute the overall utility of the mRUBiS architecture G via the corresponding utility function $U(G)$—see more details on utility of the system architecture in Section 3.1. In the context of mRUBiS, the utility of the architecture is the sum of the utility of all its tenants—see Equation 3.2. Similarly, for each tenant in mRUBiS, we assume utility independence between its constituent components and define the utility of a tenant as the sum of the utility of its 18 constituent components. To compute the utility of a component we use the *linear* utility function from Table 3.2.

Single MAPE-K loop run. The aim of this experiment is to separately evaluate the two main steps of VENUS for a *single MAPE-K run*: (1) selecting the best adaptation rules, and (2) ordering them for execution. The experiments start with occurrences of three failures of type CF1, CF2, and CF3 causing the utility of the system to drop. These utility drops are followed by a single run of the adaptation loop that resolves the three CFs by executing three repair rules. We show that VENUS makes the optimal decision during rule matching for CFs by selecting the rule that results in the maximum increase in the overall utility. In contrast, the *Static* approach fails to do so and hence is non-optimal. We also show how the order in which the adaptation rules are executed impacts the achieved reward, i.e., accumulated utility over time.

When a match for an issue is detected, VENUS computes the utilityIncrease and the costs of all possible matches among the applicable adaptation rules. The effect on the utility achieved by each rule application remains in the system while the cost of a rule has only a short-term effect on the reward; the cost of a rule application is the time required to execute the rule and realize the expected increase of the utility. Thus, in VENUS, rules with the highest utilityIncrease are prioritized over those with smaller increase but less costs. The type of the issue and the specific affected component determine the utilityIncrease of rule applications but the costs are defined at design time.



(a) Non-optimal rule selection.



(b) Non-optimal rule ordering.

Figure 7.4: Lost reward of *Static* compared to VENUS due to non-optimal rule selection (top) and non-optimal rule ordering (bottom).

Figure 7.4a shows the utility of *Static* and VENUS in the experiment with three issues. *Static* fails to reach the maximum final utility due to non-optimal rule selection. Both *Static* and VENUS select CF3 to be resolved first; *Static* performs a Heavy Weight (HW) Redeployment while VENUS Replaces the affected component and reaches a higher utility—see the first increase of utility in Figure 7.4a. As the first rule, *Static* selects a rule with less cost, i.e., HW Redeployment, thus achieves the utility increase earlier than VENUS—see the hachured region representing the improvement of utility in *Static* over VENUS—but it obtains a lower reward compared to Replace rule in VENUS. The impact of this non-optimal rule selection by *Static* remains in the system during the whole experiment and results in a loss reward for *Static* shown as the gray regions in Figure 7.4a. As the second decision, *Static* resolves CF1 via a Light Weight (LW) Redeployment while VENUS resolves CF2 via Restart which has a higher impact on the utility. As the last decision, *Static* resolves CF2 via Restart and reaches the same increase of utility as VENUS in its second rule execution, but with a delay, thus less reward. VENUS finishes the adaptation by repairing CF1 via a Restart rule. *Static* is slightly faster in resolving the three issues due to avoiding all the runtime computations. The gray and hachured regions represent respectively the lost and gained reward of *Static* compared to VENUS. The result of the qualitative experiment in Figure 7.4a suggest that the reward gained by *Static* as a result of smaller runtime overhead and choosing the cheaper HW Redeployment rule over the Replace rule does not compensate for the reward loss caused by the non-optimal decisions in *Static*.

To back our claim for optimality of VENUS, we show in Figure 7.4b that the optimal rule execution order in VENUS contributes to obtaining maximum accumulated utility over time. VENUS prioritizes rule applications with larger impacts on the utility while *Static* decides for the priorities of rule executions based on the design time estimations.

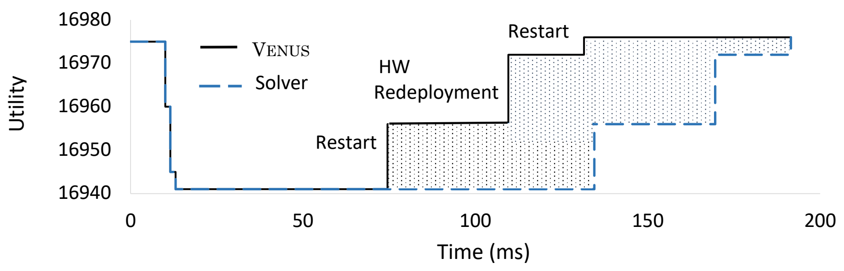


Figure 7.5: Lost reward of *Solver* compared to Venus due to longer planning time.

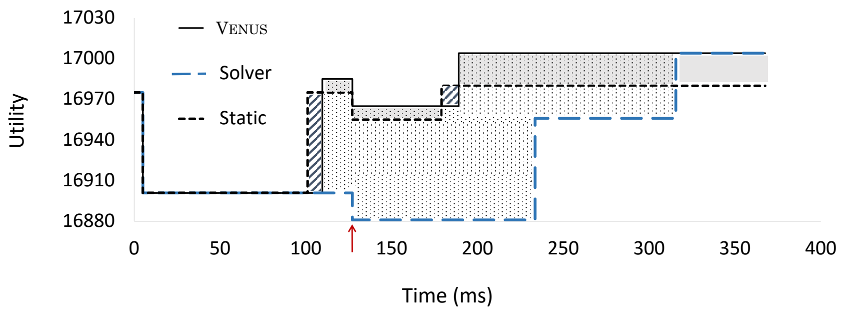


Figure 7.6: Lost reward of *Solver* compared to Venus due to longer planning time and short IAT.

This can be done considering the type of the issues. A reasonable order of rules based on the three issues in our example is: (1) removals of components (CF3), (2) crashes of components that, however, might still be operating to a certain extent (CF1), and (3) occurrences of Failures in terms of exceptions (CF2). This ordering, however, fails to take into account the actual utility of the affectedComponent which is a function of criticality, connectivity, and reliability—see Table 3.2 for the definition of the Linear utility function. The values of these attributes can dynamically change such that they are only known at runtime. Figure 7.4b illustrates a case where *Static* fails to address the issues in the correct, i.e., optimal, order. In this experiment, despite the fact that *Static* makes the optimal decision regarding the rule selection—which is not always guaranteed, see Figure 7.4a—and achieves the same final utility as Venus, *Static* loses reward due to its sub-optimal ordering of the rules (gray regions in Figure 7.4b) and gains only a slight improvement over Venus because of the smaller overhead of its planning time (hachured region).

In Figure 7.4b, Venus first repairs CF2 and *Static* repairs CF3; *Static* applies a HW Redeployment reaching a slightly higher utility but with considerably larger costs than Venus that Restarts the affected component. Because of the large cost of the rule selected by *Static*, the solution loses utility that is equal to the area of the first gray region; in contrast, it gains more utility over Venus, equal to the hachured area, due to repairing a different issue first. As the second repair decision, *Static* resolves CF1 by a Restart while Venus resolves CF3 by a HW Redeployment. Finally, *Static* resolves CF2 by a Restart and reaches the same increase in utility as Venus, but loses utility over time due to the sub-optimal rule execution order. Venus executes the repair of CF1 via a Restart rule as its third repair decision because in this scenario, compared to CF2 and CF3, repairing a CF1 has a smaller impact on the utility, thus is the last rule to be executed.

We conducted the same experiments as in Figure 7.4b with three CFs for a qualitative comparison between *Solver* and VENUS. Both solutions make identical decisions regarding the rule selection and the ordering of the rules, thus they both reach the optimal final configuration and achieve the same final utility—see Figure 7.5. However, *Solver* reaches the final utility value after a considerable delay due to its computational planning overhead, which depends on the size of the architecture and number of the issues. Despite the fact that both solution, at each point of decision-making, choose the same rules with identical costs and utilityIncrease, and that both propose similar orders for rule execution, compared to VENUS, *Solver* gains less reward. The planning overhead of *Solver* causes a delay resulting in reward loss compared to VENUS—see dotted area in Figure 7.5.

Multiple MAPE-K loop runs. This qualitative experiment investigates the impact for characteristics of input traces on the reward for self-adaptation. We use mRUBiS with 100 shops (1800 components) and X10 failure trace (see Section 6.3.1.1). We inject the simulator twice with X10 that randomly injects ten failures of type CF1, CF2, CF3, CF4, and PI1 as a failure group that cause a drop in the utility of mRUBiS. Consequently, the MAPE-K loop is executed twice. The first utility drop is followed by one MAPE-K execution. The adaptation solutions plan for and resolve all the existing failures. The feedback loop is not reentrant, thus in the case that new failures occur during the current MAPE-K run, they are ignored until the execution of the loop is finished. Therefore, the planner will not take these failures into account. Nevertheless, the occurrence of failures still causes drops in the utility of the system even if they are not yet considered by the adaptation loop and are queued to be processed by the next loop.

Figure 7.6 shows the reward of *Static*, VENUS, and *Solver* during two MAPE-K loop executions with X10. The first drop in the utility is followed by the first MAPE-K execution where all the approaches plan for repairing the failures. Both *Static* and VENUS are fast enough to resolve the failures before the second group of failures occur—the second failure occurrence time point is depicted by a red arrow in Figure 7.6. Due to the longer planning time in *Solver*, the scheme misses the on-time detection of the second group of failures. Therefore, the utility drop caused by the new group of failures remains in the system until they are detected and resolved during the second MAPE-K run. In contrast, *Static* and VENUS, owing to their short planning times, manage to detect and resolve the first group of failures and obtain the increase in the utility before the second group of failures occur.

Both *Solver* and VENUS obtain the same maximum final utility while VENUS obtains more utility over time due to faster execution of MAPE loop. The dotted areas in Figure 7.6 represent the lost reward of *Solver* compared to VENUS. Therefore, we can conclude that, similar to the optimization-based solution, i.e., *Solver*, VENUS chooses the adaptation rules that bring back the system to the maximum possible utility after the adaptation. However, the results suggest that this is not the case for *Static*—see the lower final utility for *Static* in Figure 7.6. The gray regions represent the lost utility of *Static* compared to VENUS and the hachured regions depict the utility gained by *Static* over VENUS. This experiment clarifies that in situations where the IAT is shorter than the repair time, there will be an additional loss of reward if the planning or the subsequent execution phase overlap in time with the occurrences of failures; this is due to the fact that recent failures in the system are temporarily neglected, while their impacts remains in the system until until they are resolved by the next loop, thus the system performs with a lower utility.

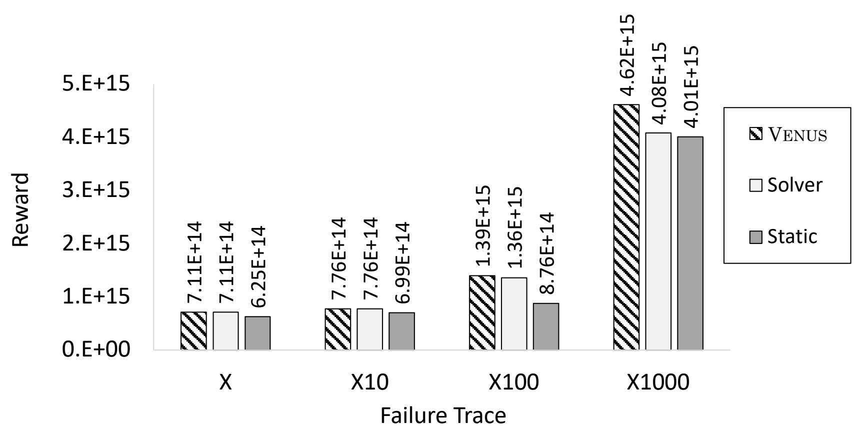


Figure 7.7: Reward of the three approaches over 50 MAPE-K runs.

7.3.3 *Quantitative Evaluation of Reward and Optimality*

In the following, we quantitatively evaluate the optimality of Venus in terms of utility and reward of the adaptation (contribution **C**5). We conduct a comparative study with *Static*, that employs design-time decision-making policy, and *Solver*, that employs an optimizer to plan an adaptation based on runtime circumstances—see Section 7.2. As discussed earlier, we exclude *Batch* from the rest of the experiments in this chapter because the approach uses the same policy for planning as *Static*. The overall utility of the mRUBiS is the sum of the utility of all its tenants—see Equation 3.2. Similarly, the utility of a tenant is the sum of the utility of its 18 constituent components. The utility of a component is computed absed on the *linear* utility function from Table 3.2.

Experiment design. To compare the reward of the three approaches, we execute them on mRUBiS with 1800 components. First, we conduct an experiment employing deterministic failure traces with long enough IAT, where the current MAPE-K loop has enough time to handle all the existing issues before the next group of failures affect the system. Next, we investigate the reward of the self-adaptation solutions under more intense and unpredictable runtime conditions. For this purpose, we use probabilistic failure models. This way, we can evaluate the obtained reward and optimality of the solutions for various failure traces with varying IAT and FGS. During the experiments, the different CF types and PI1 are equally distributed for each trace.

Experiments with deterministic traces. We inject failures to mRUBiS according to X, X10, X100, and X1000 input traces. The experiment includes exactly 50 MAPE-K loop executions. As we consider longer simulation runs, we observe rather large reward values. The reward of each approach is measured for each of the four traces. We consider a time span of 24 hours during which 50 groups of failures, characterized by the input traces, occur. The 50 groups of failures are uniformly distributed over the 24 hours resulting in an IAT of 1728 seconds (24 hours divided by 50). As summarized in Table 6.2, the employed deterministic failure traces impose long enough IAT. We manually confirmed that 1728 seconds constitutes a long enough IAT between two groups of failures in the worst case, i.e., X1000, and for the slowest approach, i.e., *Solver*.

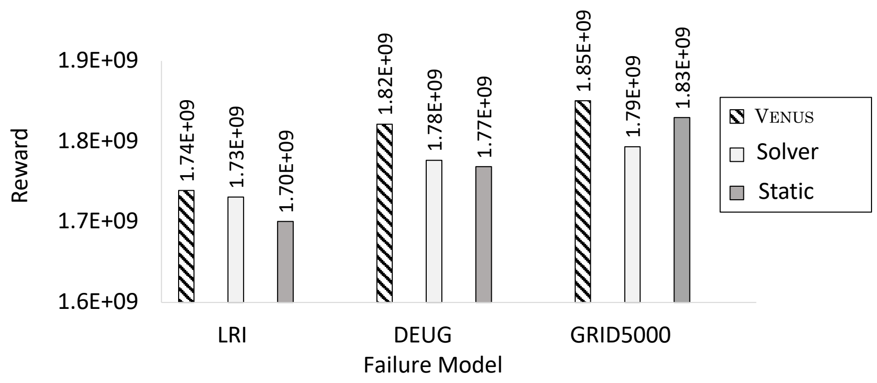


Figure 7.8: Reward for *short* traces of realistic failure profile models.

Figure 7.7 shows the reward of the approaches for the four failure traces. As the FGS increases, the difference between the reward of Venus and *Solver* becomes larger. This is due to the planning overhead of *Solver* as discussed in the context of Figure 7.5. Larger numbers of failures cause larger overhead and thus a larger loss of reward for *Solver*. This effect, however, is not visible in traces with relatively smaller FGS, e.g., X and X10.

For all the traces in Figure 7.7, compared to *Solver* and Venus, *Static* has the least amount of reward. Non-optimal decisions and wrong ordering of the adaptation rules in *Static* cause reward loss as discussed earlier in the context of Figure 7.4 and Figure 7.5. The loss of reward due to non-optimal decisions can be very severe for long system execution times since the system is performing at a sub-optimal utility level for a considerably long time. *Solver* has a larger reward than *Static* for all the traces in Figure 7.7. This is partially due to the fact that the adaptation loop always has enough time, i.e., the IAT is large enough, to resolve the current failures within one feedback loop run before the next group of failures occurs. Additionally, unlike *Static*, *Solver* aims for optimal choice of the adaptation rules with respect to the UtilityIncrease and optimal ordering of the rules to maximize its final utility and reward. Venus aims for the same optimal decisions as *Solver*, however, in the context of Venus, its relatively faster planning results in timely adaptations, thus higher reward.

Experiments with probabilistic traces. The following experiments compare the reward of the self-adaptation approaches using probabilistic failure models. Figure 7.8 shows the reward of the approaches for the *short* traces generated based on *LRI*, *DEUG*, and *Grid*5000 models—see Table 6.3 for details of the traces. Since these failure models have different characteristics, e.g., the generated traces have different failure densities, the results cannot be compared across different models. However, we can compare the results in terms of the reward achieved by the three self-adaptation solutions for each model separately. As previously shown, for large FGSs, compared to Venus and *Static*, *Solver* requires considerably more time for planning. Consequently, for *Grid*5000 short trace with 1116 failure density, *Solver* achieves the lowest reward compared to the other two approaches—see Figure 7.8. In contrast, for *DEUG* and *LRI* short traces with a total of 666 and 318 failures respectively, *Solver* performs slightly better than *Static*. Thus, the different failure densities of the traces influence the reward and optimality of the approaches under test.

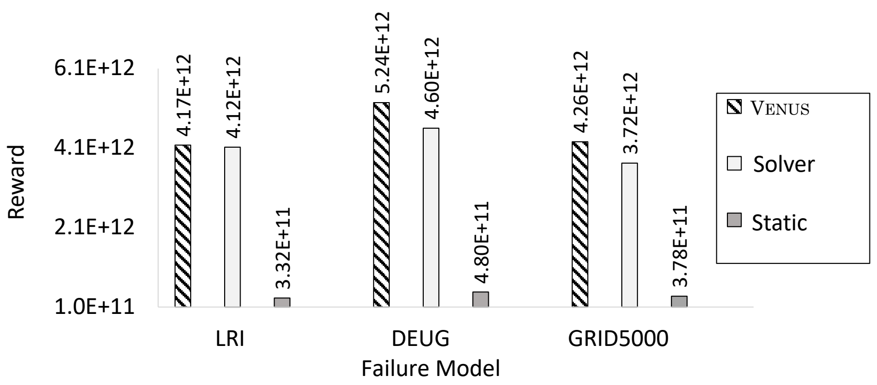


Figure 7.9: Reward for *long* traces of realistic failure profile models.

Moreover, if the IAT is shorter than the time required to resolve the failures, it causes more severe loss of reward—see Figure 7.6 for a qualitative demonstration. This effect applies in particular to *Solver*, due to its time-intensive planning, which is a further reason that this approach achieves a considerably lower reward compared to the alternatives for the *Grid*5000 with high failure density while performing better with *LRI* and *DEUG* with smaller failure densities—see Table 6.3 for detailed characteristics of the failure traces. For the *LRI* model that has the lowest failure density, the reward of *Solver* in Figure 7.8 is close to Venus while their difference is larger for the *DEUG* and *Grid*5000 traces with larger number of failures.

Figure 7.9 shows the reward for the *long* traces generated from the *LRI*, *DEUG*, and *Grid*5000 models. The traces simulate approximately a period of 30 days, allowing for observing the longer execution of the approaches. Particularly, we observe that *Static*, compared to Venus and *Solver*, severely losses reward for all the failure models. As discussed in the scope of the qualitative experiments in Section 7.3.2, the impact on the reward, caused by the non-optimal decisions of *Static*, can permanently remain in the system. Therefore, the reward loss due to non-optimal decisions remains and propagates through 30 days of system execution. Similar to the results in Figure 7.8, Figure 7.9 shows that *Solver* achieves less reward compared to Venus due to the planning overhead. However, the interesting point that comes to light thanks to the long system execution period, i.e., 30 days, is that the reward achieved by *Solver* is considerably larger than *Static*. The reason is that the reward loss in *Solver* is due to the planning overhead and seems to be compensated for over time, while the reward loss in *Static* remains in the system because the scheme does not always obtain the maximum final utility after the adaptation. Conversely, *Solver* eventually brings the system to the maximum utility level after the adaptation.

As different traces have different failure densities, the reward of the solutions for one trace cannot be compared to the one for a different trace. To enable a comparison across traces, we use traces with *equal* failure densities. Therefore, for the following experiment we use the synthetic failure models from Table 6.4 to generate traces with *equal* failure density. As confirmed by the qualitative assessment (Section 7.3.2) and quantitative experiments for reward, characteristics of the input traces influence the reward of self-adaptation. The results are presented in Figure 7.10. For the Single model, both *Solver* and Venus obtain the maximum reward because it is a simple model, i.e., at each arrival of failures, there is only a single failure to handle. Thus, *Solver* is not concerned with

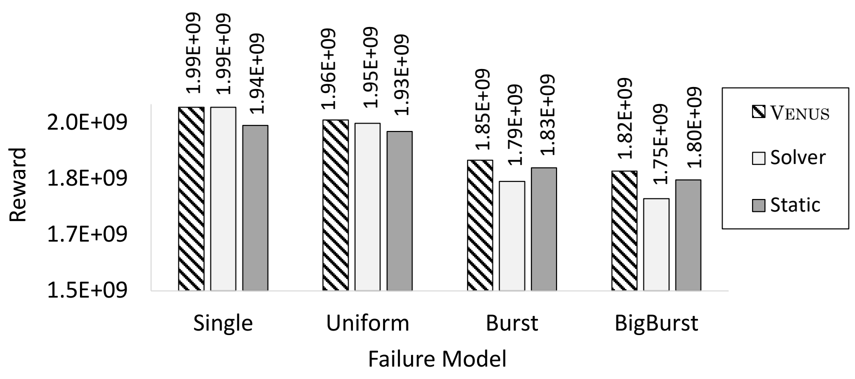


Figure 7.10: Reward for synthetic traces with equal failure densities.

planning overhead. *Static* however, achieves relatively less reward for Single model; this is due to sub-optimal choice of rules in *Static*, this effect corresponds to the impact of non-optimal decisions demonstrated by the qualitative experiment in Figure 7.4a. In the experiments with Uniform model, *Solver* achieves more reward than *Static* but less than Venus. For the trace generated based on the Uniform model, while the the average FGS is smaller than in Burst and Bigburst models, Venus still manages to outperform *Solver* in terms of reward do to its scalability and faster execution times.

As the FGS increases in the experiments with Burst and Bigburst models, all three solutions obtain less reward compared to the experiments with Single and Uniform models. Venus and *Solver* are both affected by their planning overhead for large FGSs, thus attain less reward. In case of *Static*, the reward loss compared to models with smaller FGS can be associated to the impact of non-optimal ordering of the rules during execution. We investigate this phenomenon in qualitative assessment presented in Figure 7.4b.

The quantitative evaluation of Venus in comparison to a deterministic, rule-based solution and an optimization-based solution for adaptation suggest that for the two groups of experiments, i.e., with deterministic traces imposing long enough IAT, and the probabilistic models with different characteristics regarding FGS, IAT, FET, failure density, and duration, Venus always outperforms—in terms of the adaptation reward—both *Static* and *Solver*. Only in the experiments with Single model, Venus and *Solver* obtain the same reward (see Figure 7.10) because no costly planning is required when there is only a single issue to resolve.

The experiments with multiple probabilistic traces allowed for investigating the performance and reward of Venus while the adaptable software is operating under operation conditions that were subject to extreme and sudden changes. Additionally, following the evaluation methodology introduced in Section 6.3, Venus has been evaluated while exposed to a large spectrum of input traces with divers characteristics. The results confirmed that Venus consistently showed scalability, cost-efficiency, and optimality during multiple reproducible simulation scenarios. These observations provide the supporting evidence for robustness of Venus as a solution for software self-adaptation. As discussed in Section 1.4, according to the IEEE standard [227], the robustness of a software system is defined by the degree to which the system operates correctly in the presence of exceptional inputs or stressful environmental conditions. In this section, using the mRUBiS application example, we demonstrated the robustness of VENUS, as is customary [see 306], via bombarding the self-adaptive software with valid and exceptional inputs and verify the success criteria, i.e., *if it does not crash or hang, then it is robust*—see **C**6 in Section 1.4.

7.4 QUANTITATIVE EVALUATION WITH znn.com

As the second case study for evaluating VENUS, we adopted `Znn.com`, a cloud-based load balancing system introduced in Section 6.2. Similar to the scheme of Section 7.3, we compare VENUS to the two alternative self-adaptation solutions, i.e., *Static* and *Solver*—see Section 7.2. With the help of the comparative study, we aim to investigate two features of VENUS that are captured by the individual contributions in Section 1.4 , i.e., *runtime performance* and *scalability* (**C**1), and *optimality* in terms of reward (**C**5). The qualitative evaluations in this section are according to the evaluation methodology introduced in Section 6.3 and provide coverage of a wide spectrum of input space for self-adaptive software system (**C**8).

We used `Znn.com` in combination with a large spectrum of deterministic and realistic input traces presented in Section 6.3.2. The implementation and deployment of the three solutions, i.e., *Static*, VENUS, and *Solver* replicate the steps introduced in Section 7.3. The different solutions operate on an RTM of `Znn.com`—see Figure 7.1 for a reference of the implementation. We simulated the architecture of the application example by means of the EMF models, i.e., without having to run the underlying causal connection and adaptable software. Thus, similar to the experiments with mRUBiS, connecting and synchronizing the RTM and the running system does not concern the self-adaptation solutions. We customize the `Znn.com` RTM with injectors that add client requests to the system, thus introducing adaptation issues in the RTM.

We conducted controlled experiments by keeping all the experimental parameters constant except the self-adaptation solution and the traces used as inputs for `Znn.com`. We controlled the parameter values to isolate the effects of the self-adaptation solution on the utility of the `Znn.com` architecture as well as the request response time for clients. One experiment run with `Znn.com` includes: (1) injection of client requests to the RTM, (2) an execution of the employed self-adaptation solution realized with a MAPE-K loop, and (3) validating the RTM with respect to success and utility of the adaptation.

Experiment Design. We instantiated a version of `Znn.com` with 10 severs, where only 5 are considered within the budget. Thus, during the Analyze activity of the MAPE loop, the threshold for `overBudget` issue is set to 5, i.e., instead of 15, as shown in Figure 6.4. We operate the serves in `Znn.com` with three levels of quality—see Section 6.2. Table 7.4 shows the cost and capacity parameters for servers in `Znn.com` for different content quality. In our experiments, the cost of servers that are within the budget, i.e., the first 5 servers, can be covered by the revenue of handling approximately 20% of its maximum capacity with optimal quality, or alternatively, the revenue of handling 45% of their maximum capacity with medium quality, or, finally, 95% of its maximum capacity with low content quality. Servers 6 to10 that are above the budget, however, can compensate their operation cost by handling approximately 34% and 75% of their maximum capacity with optimal and medium quality, respectively. The over budget servers, as shown in Table 7.4, cost more than within the budget servers and cannot compensate for their operation cost by providing low quality content.

Table 7.4: Server cost/capacity parameters in `Znn.com`.

Cost (per min)		**Capacity (serve per min)**		
within budget	above budge	low quality	medium quality	high quality
0.6	1	800	600	300

Table 7.5: Average planning time (ms) for deterministic traces in `Znn.com`.

Trace (*short*)	**# Req. per min**	*Static*	VENUS	*Solver*
TR_1	1	0.56	0.61	5.72
TR_{10}	10	8.71	10.73	49.76
TR_{100}	100	12.51	15.46	3,071.07
TR_{1K}	1K	80.26	99.34	51,521.62
TR_{10K}	10K	210.34	262.23	780,746.26

The overall utility of `Znn.com` at state $s = (\bar{c}, \bar{o})$, i.e., $U_{znn}(s)$, is defined according to Equation 6.1—see Section 6.2 for a reference on self-healing and self-optimization with `Znn.com`. The request arrival traces for `Znn.com` are generated based on the synthetic as well as realistic web traffic logs introduced in Section 6.3.2 and are employed as the input traffic for `Znn.com`. The traces include clients web content requests from the web servers over the course of 60 minutes for *short* traces and 24 hours for *long* traces. Injecting client requests causes issues of type `latency`, `overBudget`, `lowQuality`, and resolving them may cause further optimization issues, e.g., `underUtilized`. Similar to mRUBiS, the issues are captured as negative patterns that affect the system. The rule set $\mathfrak{R}$ includes the adaptation rules; each rule has a utilityIncrease attribute that is the impact on the utility cased by the corresponding rule application—see the meta-model of `Znn.com` in Figure 6.3 for details of the supported issues, adaptation rules, and their attributes. We execute the adaptation loop at each minute, unless an in-progress loop exceeds the one-minute time window. In this case, an execution of the consequent loop immediately follows. We only allow for a single server boot-up at a time, however, multiple servers can be active simultaneously. For one measurement run, we execute one solution (*Static*, *Solver*, or VENUS) together with one input trace from Section 6.3.2. Within one measurement run, we repeat the cycle of steps (1), (2), and (3) for 300 times or until the measurements stabilize, i.e., the standard deviation of the execution time is less than five percent of the measured average execution time. We report the mean values of the measurements. The measurements were performed on one machine.

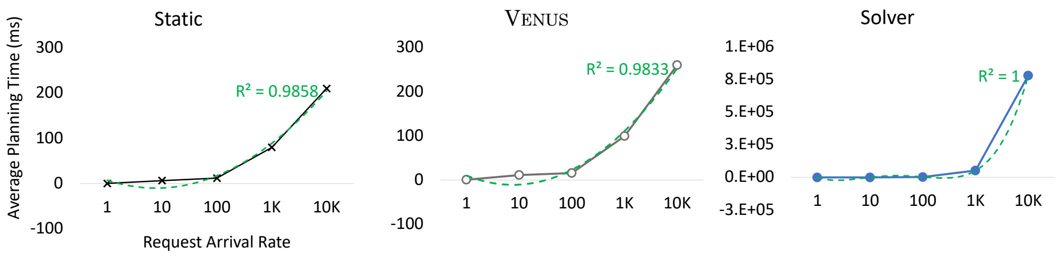


Figure 7.11: Average planning time with polynomial regression (in dashed green).

7.4.1 *Evaluation of Runtime Performance and Scalability*

To compare the planning time and scalability of the solutions, we executed them on `Znn.com` in combination with the deterministic traces from Section 6.3.2.1—see the results in Table 7.5. The experiments in this section provide the supporting evidence for contribution **C1** in Chapter 1. As the request arrival rate in the traces increases, `Znn.com` is more likely to exhibit adaptation issues such as `latency`, `lowQuality`, and `overBudget`, therefore, the planning time of the approaches increases accordingly since there are more issues to address. However, as shown previously in our experiments with mRUBiS (see Table 7.3), compared to VENUS and *Static*, *Solver* demonstrates more significant increase of the planning time for large number of issues as it has to solve large, time-intensive optimization problems.

Figure 7.11 shows the plots of planning times of the approaches from Table 7.5 with respect to the request arrival rates; for each plot, Figure 7.11 also shows the best fitting linear (polynomial) regression in dashed green lines. R-squared, i.e., R^2, is between zero and one and is the sum of squared residuals in the regression indicating a goodness-of-fit measure for linear regression models; the larger the R^2, the better the regression model fits the observations [386]. As the request arrival rate increases by the orders of magnitude, i.e., from 1 request per minute to 10K requests per minute, the planning time for *Static* and VENUS grows with $O(n^2)$, i.e., polynomial with degree two ($R^2 \approx 0.98$), however, this growth rate for *Solver* is $O(n^4)$ with $R^2 = 1$—see Figure 7.11.

Both the qualitative and quantitative evaluation of *Static* in Section 7.3 confirmed that, in comparison to *Solver* and VENUS, thanks to its minimal runtime overhead, *Static* shows minimum planning time. We observed the same pattern in the experiments with `Znn.com` presented in Table 7.5. Thus, taking the planning time of *Static* as the *base value*, VENUS yields in the worst case 25% (TR_{10K}), and in the best case 9% (TR_{10K}) longer for planning. The average planning time in *Solver*, however, in the worst case is 371,083% (TR_{10K}), and in the best case is 921% (TR_{10K}) more than *Static* for the same trace.

Figure 7.12 shows the client response times (in seconds) for the same experiments with deterministic traces as Table 7.5. In the experiments with TR_{10K}, *Solver* shows only timeout for the requests, i.e., response times are above 90 *sec*, thus not included in Figure 7.12. In Figure 7.12, the boxes represent the middle portion of the measured data points, i.e., the response times values for a solution-trace pair. The horizontal lines inside each box represent the median. The whiskers show the minimum and maximum response times. Take the chart with 100 requests as an example; the median for *Static*, VENUS, and *Solver* is 3.06, 4.2, and 10.9 *sec* respectively. This indicates that half of the requests have response times below 3.06, 4.2, and 10.9 *sec* in *Static*, VENUS, and *Solver*, respectively, while the other half have higher response times. Additionally, the maximum response time value, represented by the upper end of the whiskers, is 6.27 *sec* for *Static*, 7.02 *sec* for VENUS, and 23.32 *sec* for *Solver*. The "×" marks inside the boxes represent the mean values for the response time. *Solver* has a greater mean than the median in all the experiments; this means, the distribution of the response times in *Solver* is *positively skewed*, i.e., towards larger values. Apart from the experiment with 1 request, where the mean and the median are aligned, the mean values in VENUS are less than the median; this indicates that the distribution of response time values in VENUS are *skewed negatively*, i.e., towards smaller values. *Static* has appropriately aligned mean and median values in the experiments with 10, 100, and 1K requests; in the experiments with 1 and 10K traces, response times in *Static* are negatively skewed.

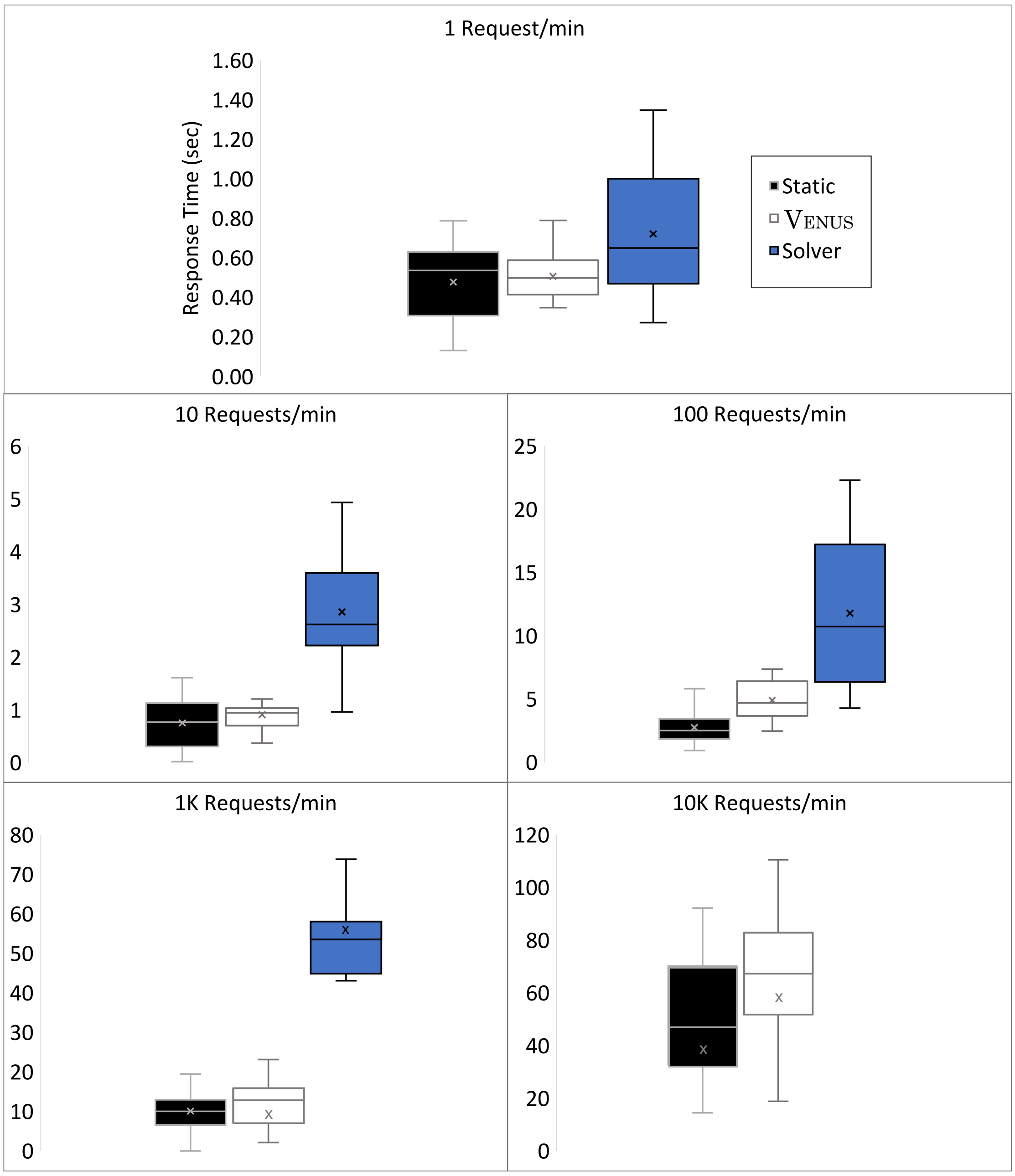


Figure 7.12: Response time (sec) for deterministic traces in `Znn.com`.

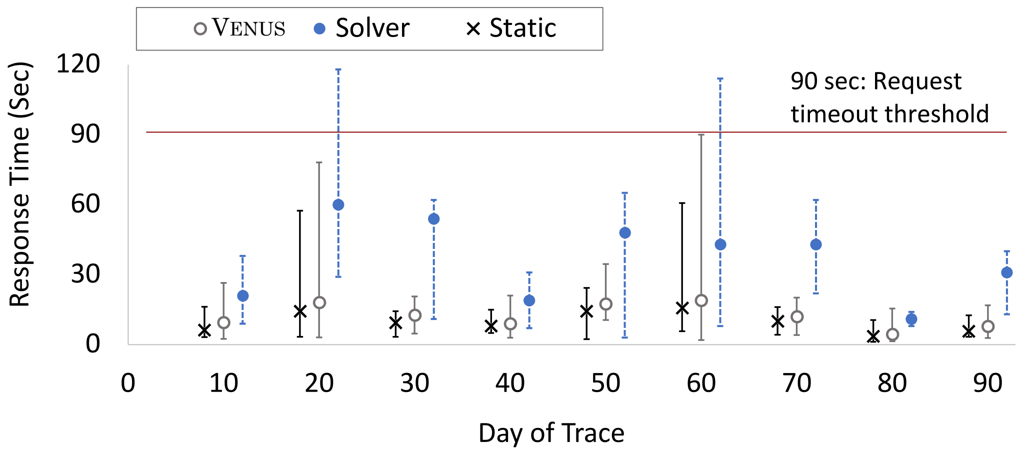


Figure 7.13: Response time for realistic traces in `Znn.com`.

The deterministic input traces for `Znn.com` enable a systematic evaluation of the performance and scalability of the self-adaptation solutions whereby runtime performance of the approaches is investigated for various, controlled arrival rates for client requests. Next, we compare the approaches in a study where we use the nine realistic FIFA traces, presented in Section 6.3.2.2 and visualized in Figure 6.8, as input traces for the system. Figure 7.13 presents the average request response times for clients in `Znn.com` for the nine short traces (60 min) from Figure 6.8. The x-axis represents the trace based on the corresponding day, e.g., the most left bars are with the Day 10 trace. Each measurement shows the average response times (bullets) as well as the maximum and minimum response times (vertical bars) of the clients in the experiments with *Day* i trace. Among the nine traces, Day 60 shows larger response time values for all the three solutions. The request arrival distribution of Day 60 in Figure 6.8 suggests that this trace has the largest burst of client requests, i.e., 4000 requests per minute. This is while the second largest burst contains approximately 2500 requests—see plots of Day 20 and Day 30 in Figure 6.8.

In the experiments with Day 60 and Day 20, *Solver* causes request timeout, i.e., above 90 sec. The same experiment shows the largest values for the maximum response times in *Static* and VENUS. Day 80 has the smallest average response time as well as the smallest maximum and minimum response times for all the approaches. The reason is that, the corresponding trace has a relatively small request burst size, i.e., 650 requests per minute. The results in Figure 7.13 are inline with the response time measurements in Figure 7.12; *Static* and VENUS have response time values in approximately the same range. This can be explained by the planning time of the two solutions, presented in Table 7.5, that are at most only 25% different.

7.4.2 *Evaluation of Reward and Optimality*

We compare the accumulated utility (reward) of VENUS to *Static* and *Solver* during the 60 minute executions of `Znn.com` with the *short* FIFA input traces from Section 6.3.2.2. We studied the response time of the clients when we played back the traces used in the experiments for Figure 7.13. For each experiment, we used a single trace from Day i with 60 min duration and an adaptation solution, i.e., *Static*, VENUS, or *Solver*. We inject the client requests to `Znn.com` every minute according to the corresponding input trace. We execute the MAPE-K loop in one-minute intervals, unless an in-progress loop exceeds this time. At the end of each adaptation, we measure the accumulated utility of the system during the last one minute. We use the utility function in Equation 6.1 to compute the system utility. The experiments in this section provide supporting evidence for contribution **C**5 in Section 1.4.

In Figure 7.14, we report the normalized reward of the three approaches over the course of 60 $\mathtt{min}$. The reward graphs for all three approaches have similar patterns. Day 60 yields minimum normalized reward across the solutions. We observed in Figure 7.13 that Day 60 also has worst response time values, in terms of maximum, minimum, and average across all traces. We associate this to the characteristics of request distributions over time—see Figure 6.8. While trace of Day 60 has a fairly small request arrival rate until minute 38, i.e., below 600 requests per minute, the reward values suggest that the large request burst between minute 38 to 50 significantly reduces the accumulated utility of the system during 60 $\mathtt{min}$.

Another interesting observation regarding the reward charts involves the experiments where *Static*, while often exhibiting sub-optimal behavior due to its deterministic decision-making (see Section 7.2), outperforms *Solver*. The aforementioned experiments belong to Day 20 and Day 60—see Figure 7.14. We showed earlier in Figure 7.13 that for Day 20 and Day 60, *Solver* demonstrates the highest response time values as well as request timeouts; consequently, low response times affect the utility of the system—see the definition of u_R in Table 6.1. Additionally, the results for the normalized reward confirm that VENUS outperforms both *Static* and *Solver*. This is interesting while, as the response time measurements in Figure 7.13 show, for all nine experiments, VENUS exhibits larger response times than *Static* (in terms of both the average and the maximum). We attribute this to the utility-driven decision-making policy in VENUS that performs adaptations yielding higher utility, thus compensating for its slightly higher client response times over *Static*. Regarding *Solver*, large planning times, as depicted in Table 7.5, significantly affect the client response times, thus the reward.

In contrast to *Static*, *Solver* exhibits similar pattern to VENUS, in terms of variations in the reward values. This can be explained by the fact that the decision-making in *Static*, compared to *Solver* and VENUS, by principle is less sensitive to runtime operation conditions—see Table 7.1. Consequently, while this reduces the planning overhead for the approach, thus avoiding steep drops in system utility, *Static* also overlooks the opportunities, only surfacing during runtime, to improve its reward due to its runtime-oblivious planning. Take for example the change of the reward in the approaches during transition from Day 50 to Day 60, and from Day 70 to Day 80 in Figure 7.14; we observe large variations of the normalized reward for *Solver* and VENUS, while *Static* maintains a more steady reward values during both experiments.

7.5 POSSIBLE VIOLATION OF ASSUMPTIONS

In the context of VENUS, we made several assumptions (listed in Table 4.1) and discussed that these assumptions are usually justified for rule-based, self-healing systems and for the limited scope of rule-based, self-optimizing systems—see Section 4.4.4. To further assess the robustness of VENUS, in this section, we investigate how violation of the assumption (A2) impacts the reward and scalability of VENUS. Among the six assumptions we made for the validity and applicability of VENUS, the reason we chose to quantita-

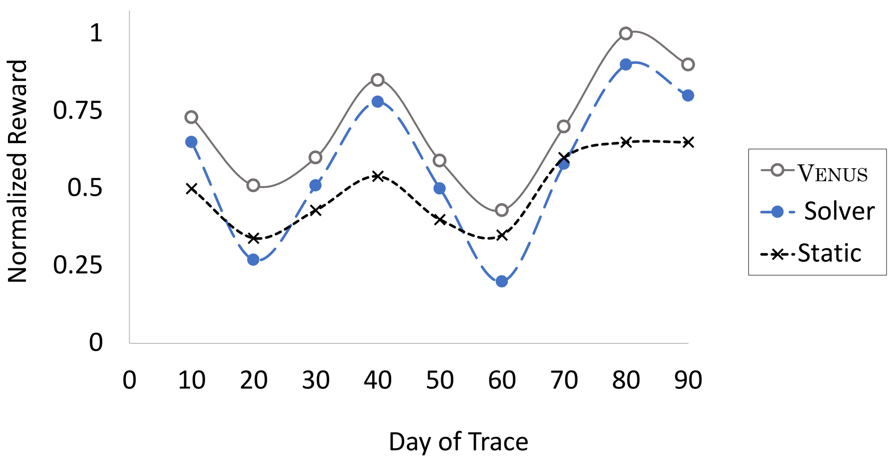


Figure 7.14: Normalized reward for realistic traces in `Znn.com`.

tively investigate the assumption (A2) is that the rest of the assumptions are either always justified for the targeted class of problems and systems, e.g., assumption (A1), or demonstration of their violations via controlled experiments required development effort that is beyond the scope of this section, e.g., assumptions (A3)- (A6).

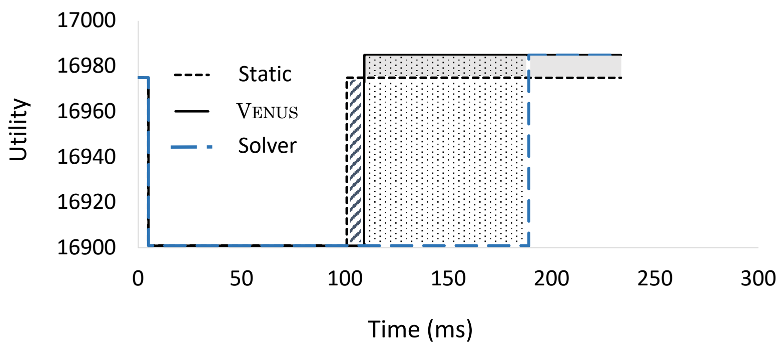
In the following, we first empirically investigate how violating the assumption (A2) affects the utility and reward of VENUS; next, we discusses how violations of the rest of the assumptions in Table 4.1 affect the scalability and runtime performance of VENUS.

7.5.1 *Violation of A2: Impact on Reward*

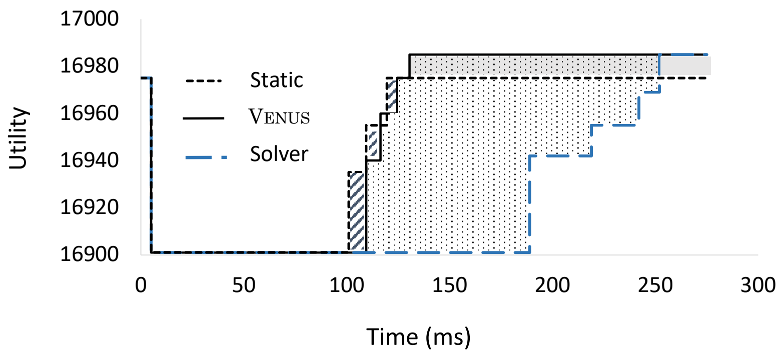
Assumption (A2) requires that adaptation rules are *deterministic* and *effective*, i.e., an adaptation rule is always able to resolve the corresponding issue. However, there might be cases where the rules do not succeed in resolving the issue, thus violating assumption (A2).

Qualitative assessment. We first show through a scenario with *probabilistic adaptation rules* how violating assumption (A2) affects system utility. A probabilistic adaptation rule has a success likelihood for resolving the adaptation issues. If a rule is not successful, the issue remains in the system and is dealt with during the subsequent MAPE-K run(s). To study the impact of ineffective rules on the utility, we randomly injected multiple failures of type CF1- CF4 and PI1 to mRUBiS with 1800 components; as showed in Figure 7.15, the failure injection causes a drop in the utility and is followed by an execution of the MAPE-K loop via one of the three self-adaptation approaches.

In Figure 7.15a, the first MAPE-K loop execution resolves *all* the existing failures. In this scenario, the adaptation rules are 100% effective, entailing they successfully resolve the corresponding issues, i.e., assumption (A2) holds. Thus, after the adaptation is complete, the utility of mRUBiS is restored to *the same* value (*Static*) or possibly *higher value* (VENUS and *Solver*) than before the failure injection. Figure 7.15b repeats the same experiment, except that the success likelihood of the adaptation rules is reduced to 50%. After the first MAPE-K run, the adaptation approaches fail to restore the utility of mRUBiS due to partially ineffective rules; as a result, the unresolved issues remain in the system, delaying the envisioned utility increase as otherwise would have been obtained, and require additional attempt(s), i.e., executions of the MAPE-K loop. Each MAPE-K run has one attempt of executing the rules. Consequently, the point in time when all the ex-



(a)



(b)

Figure 7.15: Reward of mRUBiS in experiments with probabilistic rules.

isting issues in the system are resolved and the system utility is restored is postponed causing the system to operate with reduced utility in the meantime, thus obtaining less reward.

The dotted area in Figure 7.15 shows the lost reward of *Solver* compared to VENUS. The gray (hachured) areas represent the reward loss (gain) of *Static* over VENUS. In general, the longer a failure remains in the system, for instance, because of the ineffective adaptation rules, the larger is the reward loss. Moreover, as confirmed in Section 7.3, *Solver* has a longer planning time, thus requires more time to resolve the failures compared to the other approaches. Unsuccessful execution of the adaptation rules results in potentially larger number of issue to be delegated to the subsequent execution(s) of the MAPE-K loop. The qualitative assessment in Figure 7.15 suggest that compared to VENUS and *Static*, *Solver* is more strictly affected by the violation of the assumption (A2) as ineffective rules entail more frequent planning by the approaches.

Quantitative evaluation. We now demonstrate the impact of violating assumption (A2) on the reward and optimality of VENUS via a set of qualitative experiments. The experiment studies how this phenomenon affects the two alternative solutions in comparison to VENUS. The experiment starts with the random injections of multiple failures of type CF1- CF4, and PI1 to mRUBiS with 100 tenants (1800 components). The short trace extracted from *Grid*5000 model is used for simulation—see Table 6.3. We consider different success likelihoods of 100%, 75%, 50%, 25%, and 5% for all adaptation rules. If a rule is not successful, the dropped utility is not restored as the issue still remains in the system. As the success likelihood decreases, the solutions obtain less reward since the unresolved issues remain in the system for longer time until they are finally removed.

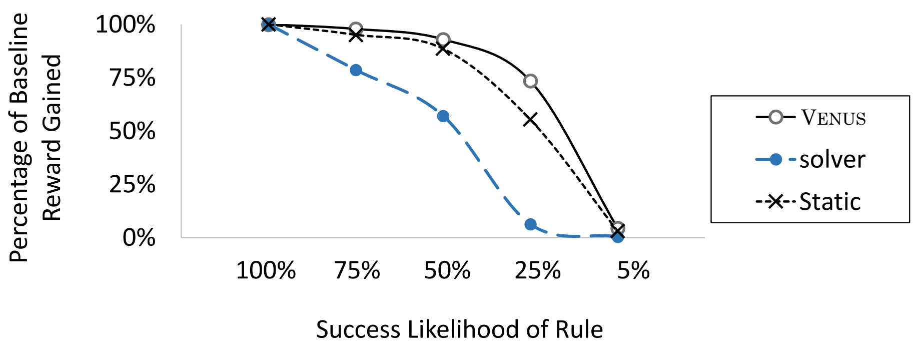


Figure 7.16: Effect of probabilistic rules on reward.

Figure 7.16 shows how different success likelihoods of the adaptation rules affect the reward of the approaches; the *baseline reward* is the system reward when rules have 100% success rate.

As the rules fail to resolve the issues, the MAPE-K loop keeps being re-invoked to resolve the remaining failures during additional runs. Meanwhile, new failures might occur, increasing the total number of the failures to be addressed. Consequently, the accumulated unresolved failures, in addition to the newly occurred ones, constitute a larger FGS, i.e., compared to the case of rules with 100% success rate. Figure 7.16 shows that Venus, for rules with 75% and 50% likelihood, obtains 98% and 93% of the baseline reward, respectively; this value drops to 73% for the 25% success likelihood. The reward for *Static* is affected similarly to Venus. This is due to the fact that the subsequent execution(s) of the MAPE-K loops in both *Static* and Venus are adequately prompt, thus the failures do not remain long in the system until they are eventually resolved and the utility of the system is restored. Probabilistic adaptation rules affect the reward of *Solver* relatively more severely; after an unsuccessful attempt to resolve the issues, the subsequent MAPE-K loop in *Solver* deals with an increasingly large optimization problem to plan for the old and the new failures. More frequent planning is required to resolve the remaining issues and the expected reward of *Solver* drops faster than the one of Venus and *Static*.

In this section, Venus is executed outside its intended operation condition, i.e., when the assumption (A2) does not hold. The empirical results suggest that while violating assumption (A2) negatively affects the reward of Venus, the approach exhibits robustness in that it continues to be functional and effective. The results in Figure 7.16 show that Venus maintains 98% and 93% of its optimal reward in experiments with rules that have 75% and 50% success likelihood; this can be attributed to the scalable and fast planning in Venus. The experiments in this section provide the supporting evidence for contribution **C6** that indicates Venus is a robust approach as the scheme preserved an acceptable percentage of its optimal performance while being executed outside its envisioned operation conditions—see Section 1.4.

7.5.2 *Violation of Assumptions: Impact on Scalability*

We discuss in the following how violations of the assumptions listed in Table 4.1 affect the scalability of Venus, or any generic self-adaptation approach—see Section 4.4.4 for discussions on justification of assumptions for Venus.

The reduced success likelihood of the adaptation rules requires additional MAPE-K loop executions during the IAT to resolve the remaining failures in the system. Unless the IAT is very short, such that not enough attempts in additional MAPE-K runs can be achieved, the violation of assumption (A2) will not affect the scalability of the self-adaptation because the size of the planning problem does not change. A violation of assumption (A3a) indicates that adaptation rules have side-effects in terms of introducing new issues upon execution. This side-effect introduces additional failures to the originally detected ones. Therefore, it can impact scalability by increasing the FGS. This effect impacts the runtime performance of any self-adaptation solution as it entails every rule execution might cause additional issues that the MAPE-K loop has to address. However, self-adaptation solutions that have time-intensive analysis and planning phases, e.g., *Solver*, are most likely to suffer from this effect in terms of their runtime performance. The quantitative evaluation of Venus in Section 7.3 confirmed that Venus has a reasonably low planning time for combinations of large architectures and large FGS—see Table 7.3 and Figure 7.2.

The violation of assumptions (A3b) and (A4) imply that there are dependencies between the adaptation rules. Detecting and resolving such dependencies can complicate the planning phase, thus negatively affecting scalability. To address this problem, more exhaustive solutions, e.g., model-checkers, are required. However, the dependency between the adaptation rules is directly related to the rules and not to the employed self-adaptation solution. Therefore, the impact will equally affect any solution for self-adaptation, particularly, if they all use the same technique to resolve these dependencies.

Violation of assumption (A6) can potentially reduce the scalability of Venus because it entails a more costly pattern-matching process. Finally, a violation of assumptions (A1) and (A5) will not have any impact on the scalability of the adaptation; as discussed in Section 4.4.4, the violation of assumption (A1) does not hold for the case of self-healing systems in general and for rule-based, self-optimizing systems. The violation of assumption (A5) indicates the need to extend the context for the adaptation rules which happens at design time and does not impact the runtime scalability of the adaptation solutions.

7.6 THREATS TO VALIDITY

Internal validity. Threats to internal validity concern how we performed the experiments and interpreted the results [429]. To address such threats, we systematically investigated the alternative approaches in a comparative study by using the controlled simulation environment for the application examples, i.e., mRUBiS [414] and `Znn.com` [90]. We conducted controlled experiments by keeping all the experimental parameters constant except for the self-adaptation solution and the traces used as inputs for the application example. We controlled the parameter values to isolate the effects of the self-adaptation solution on the quality objective of interest, i.e., performance, scalability, optimality, and robustness.

This chapter served two main goals; (i) studying the effect of the incremental execution of the adaptation engine on the adaptation performance; (ii) evaluating the impact of different planning mechanisms on the scalability and the reward of the adaptation. To study the effects of incrementality, the baseline, non-incremental solution shared the same implementation setting and used the same architectural RTM and pattern-matching tool as the incremental alternatives, the only difference was how the approaches captured the changes of the RTM and processed it. To focus on the effects of planning, the three compared self-adaptation approaches shared the identical monitoring, analysis, and execution phases of the MAPE-K feedback loop and they used the same architectural RTM and utility function. To guarantee a fair comparison of the approaches and to take variations of the measured execution times into account, the experiments followed our evaluation methodology and are driven by the input traces, constructed from the failure profile models or real world traces, that enable replicating a simulation for the different approaches and over multiple runs. Thus, we used various failure profile models and traces to investigate the effects of the incrementality and planning mechanisms on the scalability and the reward. For instance, to investigate scalability in the experiments with mRUBiS, we focused on increasing the failure group size (FGS), the rate of the injected issues through the traces, and the size of the system architecture.

We conducted multiple experiments with two different application examples that aimed at either qualitative assessment or quantitative evaluation of scalability and reward. The experiments considered single or multiple MAPE-K runs and satisfied either all or only a subset of the assumptions so that the results and their interpretation were always focused to concrete questions without confounding different effects and aspects of our overall evaluation. Finally, we followed the benchmark guidelines proposed by [378] in all experiments to obtain trustworthy measurements and results.

External validity. Threats to external validity may restrict the generalization of our evaluation results outside the scope of our experiments [429]. Such threats are the particularities of the two systems under adaptation, the specifications of the input traces, the choices of the utility functions, and the choice of the alternative self-adaptation approaches. To mitigate these threats, we used mRUBiS and `Znn.com` as the system under adaptation that allow the injection of generic architectural failures and the repair of these failures by generic architectural adaptation rules. We can consider both application examples as generic and representative exemplars for architecture-based self-adaptation. To characterize how the performance issues occur and propagate in practice, besides the synthetic and the deterministic traces, we especially used realistic ones as well as the probabilistic failure profile models that stem from real-world systems [17, 155]. We also extended these realistic models to cover edge cases such as single, isolated failures or large failure bursts with up to about 450 failures (mRUBiS) or 10K requests (`Znn.com`). Thus, our evaluation results hold for real-world failure behavior (mRUBiS) and user web access behavior (`Znn.com`).

The threat of using a specific utility function is negligible in our opinion since the same function is used for all alternative self-adaptation approaches in the study. Thus, the utility function does not cause any effect that differs between the approaches, which could otherwise influence the results; hence, we expect similar results with any other utility function. We compared Venus to two other self-adaptation approaches and one baseline solution in our experiments; consequently, while the relative results cannot be generalized to other approaches, the alternative solutions in the comparative study cover the edge cases in the solution space of self-adaptation: *Static* scales very well but often achieves non-optimal rewards while *Solver* typically achieves optimal rewards but does not scale due to the costly solving of the optimization problem. *Batch* precludes incremental execution of the adaptation and serves as a baseline to shed light on the impact of the incremental adaptation; nonetheless, considering these edge cases, we can conclude that Venus is both scalable and optimal in creating plans for self-adaptation.

Finally, a major threat to external validity is the use of simulated systems instead of real systems. Our SLR of the state-of-the-art on evaluating self-healing systems in [176, 177] revealed that simulation is the only means to evaluate the performance and reward of self-healing systems. Additionally, based on the conducted SLR, in this thesis, we employed an evaluation methodology to advance the state-of-the-art in the evaluation of self-healing systems—see Section 6.3. In this context, we can categorize the state-of-the-art in the self-healing systems as work that relies on simulation and does not use input (failure) traces at all[4], or work that rely on simulation and use input (failure) traces, although these traces are not (based on) real-world traces[5]. Thus, we can, at first, conclude that simulation is the common practice in the literature to evaluate self-healing systems, which confirms the general finding for self-adaptive systems by Weyns et al. [426]. Secondly, our SLR indicates the lack of appropriate methods to evaluate self-healing systems because to the best of our knowledge and effort we did not find any approach with performance claims that provides a complete failure profile either as representative real-world test traces or models for occurrences of failures—see [177]. This distinguishes this thesis from the current state-of-the-art, in the sense that we used realistic failure profile models for occurrences of failures and real world input traces. This allows us to systematically and extensively evaluate our approach using real-world data. Thus, our comparative study goes beyond the state-of-the-art in evaluation for self-healing and self-adaptive systems.

Construct validity. Threats to construct validity comprise situations where the used metrics do not measure the construct, i.e., the concept [429]. The major threats to construct validity are the correctness of the simulation environment, our implementation of the alternative self-adaptation approaches, our adaptation of the realistic input traces, and our construction of the input traces from these models. To address these threats, we use mRUBiS and `Znn.com` as our simulation environments that have been accepted as the two most well known exemplars by the research community on self-adaptive software and that have been extensively tested by students in the scope of our courses and lectures on self-adaptive software. Moreover, the implementations of all the self-adaptation approaches have been tested with both mRUBiS and `Znn.com` simulators while the adapted input traces and the traces constructed from the failure profile models have been double-checked by the fellow colleagues during scientific collaborations leading to publications.

Statistical conclusion validity. Threats to conclusion validity are factors that may lead to drawing incorrect conclusion about a relationship in the observations made during an experiment. In order to mitigate such threats, we employed a large set of realistic and

4 Examples of such work are [69, 123, 205, 316, 350, 366, 370].

5 Such approaches either use observed and manually adjusted failure traces, e.g., [165, 197, 224], probabilistic or simple random failure traces, e.g., [7, 80, 337], or deterministic failure traces, e.g., [11, 75, 76, 116, 193, 203, 295, 334] .

probabilistic input traces with diverse characteristics to reduce bias in the observed outcome. During the comparative studies, the alternative solutions have been implemented according to the same implementation guidelines; the experiments were executed under identical, controlled, and reproducible conditions; the reported results were average of relatively adequate number of executions.

7.7 SUMMARY

In this chapter, VENUS was evaluated with respect to the quality attributes of performance, scalability, optimality, and robustness. The evaluation included qualitative and subjective assessments of the quality attributes while we provided quantitative data, obtained based on rigorous evaluation of VENUS. The experiments were based on an evaluation methodology and provided coverage of a wide spectrum of the input space for self-healing systems. We conducted a sensitivity analysis of the results in this chapter by investigating VENUS over a spectrum of architecture sizes, input traces, and alternative solutions in comparative studies. To assess the generalizability of the results, large variety of the input traces along with two different application examples were used. The results confirmed that VENUS consistently showed scalability, cost-efficiency (timeliness), and optimality during multiple reproducible simulation scenarios. These observations provide the supporting evidence for robustness of VENUS as a solution for software self-adaptation. The timeliness, scalability, and utility of VENUS are demonstrated to remain robust during multiple simulated experiments whereby the self-adaptive software was bombarded with realistic and extreme inputs. The robustness of VENUS was also demonstrated through executing it outside its intended operation condition, i.e., when certain validity assumptions were violated.

In the context of VENUS, this chapter provided supporting empirical evidence for the following contributions stated in Section 1.4:

C1 Incremental execution of the adaptation loop, thus attaining scalability

C5 An optimal solution for self-adaptation with negligible runtime overhead

C6 A robust solution

C8 Coverage of a wide spectrum of self-adaptation problem space beyond the state-of-the-art

7.7.1 *Fulfillment of Requirements*

Based on the overall discussion and evaluation of VENUS, we now summarize its capabilities by discussing its coverage of the requirements for architecture-based self-adaptive software presented in Chapter 1—see Table 1.1.

We investigated the *optimality* of VENUS in terms of objective satisfaction captured by its utility function. In effect, we showed that, similar to an optimization-based solution that uses a constraint solver for planning, VENUS adapts the system to the optimal configuration in terms of final system utility and in the meantime, maximizes the accumulated utility, i.e., reward, thus fulfills $\mathcal{R}1$. $\mathcal{R}1$ indicates that the solution satisfies system quality objectives at a desirable level.

We showed that, in comparisons to a pure deterministic, rule-based adaptation policy that employs design-time preferences with no runtime overhead, planning with VENUS adds only negligible runtime overhead. The response time and utility of the adaptation were shown to be insensitive to the increase in the size of the architecture. The *timeliness* of VENUS was demonstrated to be affected only linearly (quadratically in the worst case) versus exponential growth of the number of the adaptation issues. Additionally, via a comparative study with a baseline solution that uses the same implementation setting as VENUS, but executes the adaptation in a state-based manner, we investigated the impact of incremental execution of the adaptation engine that integrates VENUS. Our solution for event-based detection and processing of the adaptation issues, incremental utility-change computation and planning, and finally, incremental execution of the rules constitute an adaptation engine that is amenable to incremental execution. The quantitative evaluation in this chapter showed improvements of the runtime performance and *scalability* of *any* adaptation mechanism that is integrated in the proposed adaptation engine. The results confirm that *Static*, *Solver*, and VENUS, independent of their planning policy, have significantly lower execution times in comparison to a non-incremental, state-based solution. Therefore, VENUS fulfills $\mathcal{R}2$ and $\mathcal{R}3$ that indicate the solution is cost-effective and scalable, respectively.

VENUS pursues a utility-driven, rule-based scheme to plan the adaptations. Using utility functions to steer the adaptation provides for balancing the quality-cost trade-offs by VENUS at runtime. In this chapter, we showed that the proposed scheme maximizes the reward of the adaptable software while significantly reducing the operation cost in terms of planning time. Additionally, being embedded in the proposed adaptation engine, which is executed incrementally, further reduces the execution cost of VENUS. Consequently, we have shown in this chapter that VENUS explicitly captures and optimizes the quality-cost trade-off, thus fulfills $\mathcal{R}4$.

8 LEARNING UTILITY-CHANGE PREDICTION MODELS: APPLICATION AND EVALUATION

In this chapter, using mRUBiS application example, we first present an instantiation of the proposed methodology in Section 3.3.3 to learn prediction models for utility-change in self-adaptive systems. We elaborate on the steps to design and execute, respectively simulate, the adaptable system to collect data for machine learning and elaborate on the process of learning the prediction models in Section 8.1. More specifically, we answer the following question via investigating the prediction models with respect to their prediction error and runtime effort:

Q 8.1 *Is it possible to systematically learn a prediction model for the utility-changes in a utility-driven, rule-based, self-adaptive system (particularly when the detailed knowledge of the system is not available)?*

Next, in Section 8.2, we provide a quantitative evaluation of the prediction model performance with respect to the reward during adaptation. Finally, we study the similarity metrics and their accuracy in steering the choice of best performing prediction models before executing the models on a running system and discuss the threats to the validity of the results. We evaluate the methodology in Section 8.2 concerning the following two questions:

Q 8.2 *Do the prediction models lead to a system performance that approximates an analytically-defined optimum (real loss lower than 10%), considering that different arbitrary bounds could be chosen to trade between model accuracy and model runtime effort?*

Q 8.3 *Can the suggested methodology properly* select *the best prediction model without requiring the models to be deployed on a real system?*

Finally, we summarize the chapter and discuss the fulfillment of the requirements in Section 8.3.

8.1 APPLICATION

In Section 3.3.3, we presented a methodology to train prediction model $\hat{U}^*_\Delta(\tilde{c}, \tilde{o}, a)$ that properly approximates the utility-change function $\hat{U}_\Delta(\bar{c}, \bar{o}, a)$—see Figure 3.9 for an overview of the steps of the methodology. In the following, we use the mRUBiS application example to elaborate how each step of the methodology is applied to an exemplary self-adaptive software system.

8.1.1 *Step 1: Data Generation*

We equipped mRUBiS with both self-healing and self-optimizing capabilities as described in Section 6.1.1. The implementation of the adaptable software replicates the

Table 8.1: Failure traces used for steps of methodology.

	FGS	IAT(s)	Used in
*Grid*5000	$\mathrm{LOGN}(1.88, 1.25)$	$\mathrm{LOGN}(-1.39, 1.03)$	Step 1- 3
LRI	$\mathrm{LOGN}(1.32, 0.77)$	$\mathrm{LOGN}(-1.46, 1.28)$	Step 4
DEUG	$\mathrm{LOGN}(2.15, 0.70)$	$\mathrm{LOGN}(-2.28, 1.35)$	Evaluation

implementation setting presented in Section 7.1. To generate data for machine learning, a failure trace based on *Grid*5000 failure model is used—see Table 6.3. We use different failure traces, i.e., *LRI* and *DEUG*, for different steps of the methodology; this is important to guarantee independent outcomes among the steps of model building (step 1-3 in Figure 3.9), model selection (step 4 in Figure 3.9) and model evaluation. Table 8.1 shows where we use each failure trace. For convenience, Table 8.1 contains the Inter arrival time (IAT) and failure group size (FGS) distributions of each model originally presented in Table 6.3. To have a good coverage of the configuration space, we equally inject the failures to components and connectors in mRUBiS.

Each failure causes a drop in the utility. The random adaptation engine randomly assigns each reached configuration and observation to an applicable adaptation rule r. Applying an adaptation rule resolves the corresponding failure and increases the utility. The overall utility for an instantiation of the mRUBiS architecture is the sum of the utility of the tenants. The utility of a tenant is the sum of the utility of its constituting components. The utility-changes are *observed* via measurable quality attributes that we can measure (observe) independent of the model $S(\bar{c}, \bar{o})$—see Section 3.3.3.1.

During data generation, we execute the adaptable system, i.e., mRUBiS with 100 tenants, together with a *randomly operating* adaptation engine on top. We use *Grid*5000 model to generate an input trace for the simulator. In our implementation architecture—see Figure 7.1—instead of Venus, we employ a random adaptation solution where assignment of issues to the applicable rules is random, in contrast to Venus that decides based on the utilityIncrease and costs of the rules.

Four different variants of mRUBiS with four different utility functions are employed. Table 8.2 summarizes the employed variants of mRUBiS and the descriptions of the utility function $\hat{U}$ encoded in each variant. The mathematical formulas of the utility functions $\hat{U}$ are detailed in Table 3.2. These functions are only employed on mRUBiS as a *black-box* model to provide the *feedback* for machine learning—see [361]. The details of the $\hat{U}$ functions, providing the utility-changes, are hidden to the machine learning method. Finally, we generate the data by executing each variant of mRUBiS with a

Table 8.2: Variants of mRUBiS.

mRUBiS Variant	**Description of the Encoded $\hat{U}$**
Linear	$\hat{U}$ only includes linear elements.
Saturating	There are saturation effects considered in $\hat{U}$.
Discontinuous	$\hat{U}$ includes discontinuous but linear steps.
Combined	$\hat{U}$ combines all three cases above.

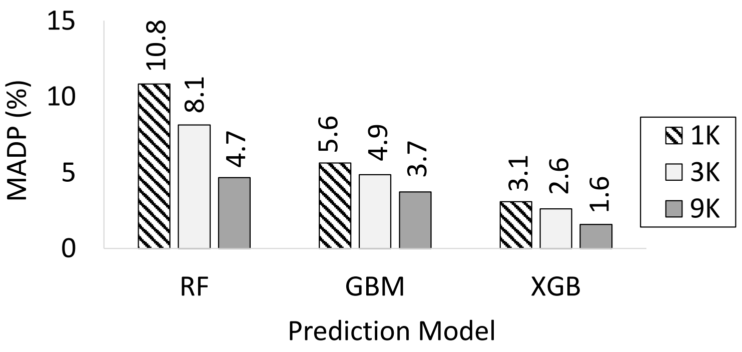


Figure 8.1: MADP for Combined utility function across prediction models trained with all dataset sizes.

sample configuration and observation along with the selected rule r and the utility-change.

8.1.2 *Iterate Step 2, 3, and 1: Training, Validating, and Preparing*

This section answers Q8.1 for a spectrum of utility function complexities, machine learning methods, and dataset sizes, i.e., providing sensitivity analysis of the results. Table 3.2 and Table 8.2 present the different utility functions that are considered for mRUBiS. The machine learning methods are Random Forest (RF) [55], Gradient Boosting Models (GBM) [153], and Extreme Gradient Boosting Trees (XGB or XGBoost) [86]—see Section 3.3.3.2. We used 1K, 3K, and 9K data points as different data set sizes. All the datasets are publicly available [see 174]. We saved 30% of each of the datasets for validation and used the remaining 70% for training and testing, i.e., 10-fold cross-validation—see Section 3.3.1.

We enable model validation by enforcing a usual data splits between training and validation. The split with lowest error corresponded to 70% for training/testing and 30% for validation. We also investigated data splits of 80/20 and 90/10, which did not show better results for the larger datasets—see Appendix B. The prediction models are trained and validated using 10-fold cross-validation over a range of hyper-parameter values. Among different prediction models, we choose the ones with the smallest *Root Mean Square Error* (RMSE). The hyper-parameters that presented larger impact on RMSE were the number of trees, the number of elements on the leaf nodes, and the maximum depth of the tree—See Appendix B for a detailed results of this tuning process.

To validate the prediction models, we compute the error between the actual and the predicted utility based on Equation 3.16. This error is further normalized, as MADP (see Equation 3.16), to be compared across different datasets. MADP is minimized by allowing the decision trees to select all the available input features. This is accomplished by tuning various hyper-parameters such as number of nodes in the leaves and maximum depth of trees. This is confirmed in the models exported to *predictive model markup language* (pmml) format[1].

The validation results for the Combined variant across the three prediction models, named after the learning method used to train them, show that larger dataset sizes correspond to lower MADP values—see Figure 8.1. The same pattern is observed for the

1 `http://dmg.org/pmml/v4-0-1/GeneralStructure.html`– accessed 18 March 2023.

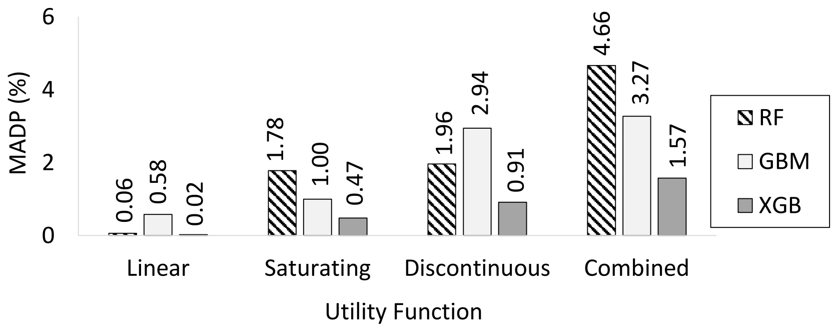


Figure 8.2: MADP across utility functions and prediction models trained with 9K datasets.

other three utility functions, i.e., Linear, Saturating, and Discontinuous. This suggests that we can optimize for accuracy by adopting the larger dataset 9K. We also investigated larger datasets, i.e., 10K, 100K, but the prediction error saturated after 10K data points.

Results in Figure 8.2 show that all three prediction models present increasing MADP as the complexity of the utility function increases. Except for the Discontinuous variant, GBM and RF models exhibit MADP values that are less than 0.5% different. The MADP value for the XGB is twice to three times smaller than the ones from the GBM and RF. This suggests that the XGB model would probably perform better when deployed on the running system. However, it is not clear whether GBM or RF would perform better for all the utility functions.

To check the runtime effort of the prediction models, discussed in Section 3.3.3.6, we used 9K datasets for training. While automatic feature selection and hyper-parameter tuning allowed us to minimize prediction error, it increases the size of the decision trees, i.e., the running effort. Therefore, we tuned the trees to keep runtime effort within one order of magnitude of each other. Take for instance the RF model that has the highest prediction error; we further minimized MADP for this model, henceforth *RF-Heavy* model, by respectively reducing the minimal number of elements in the leaves, from 10 to 5, and increasing the number of trees from 100 to 200. As a result, the RF-Heavy model presented a reduction in MADP from 4.66% to 4.09% for the Combined variant, but the runtime effort increased in more than two orders of magnitude (from 67,065µs to 8,723,761 µs). Hence, the model with slightly larger MADP is chose. This is performed for all the other models—see the results in Figure 8.3. The choice of learning

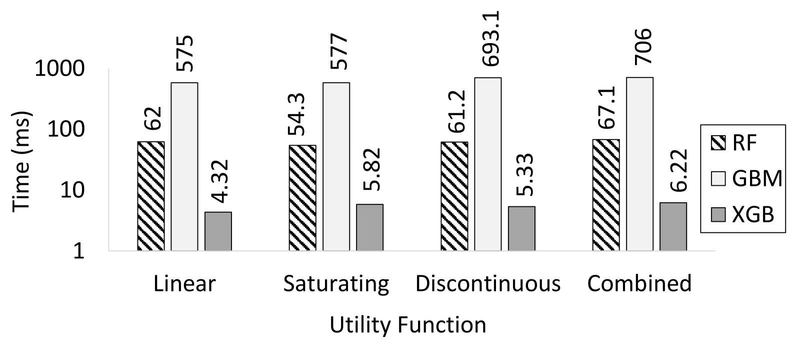


Figure 8.3: Runtime effort across utility functions in logarithmic scale (9K dataset is used).

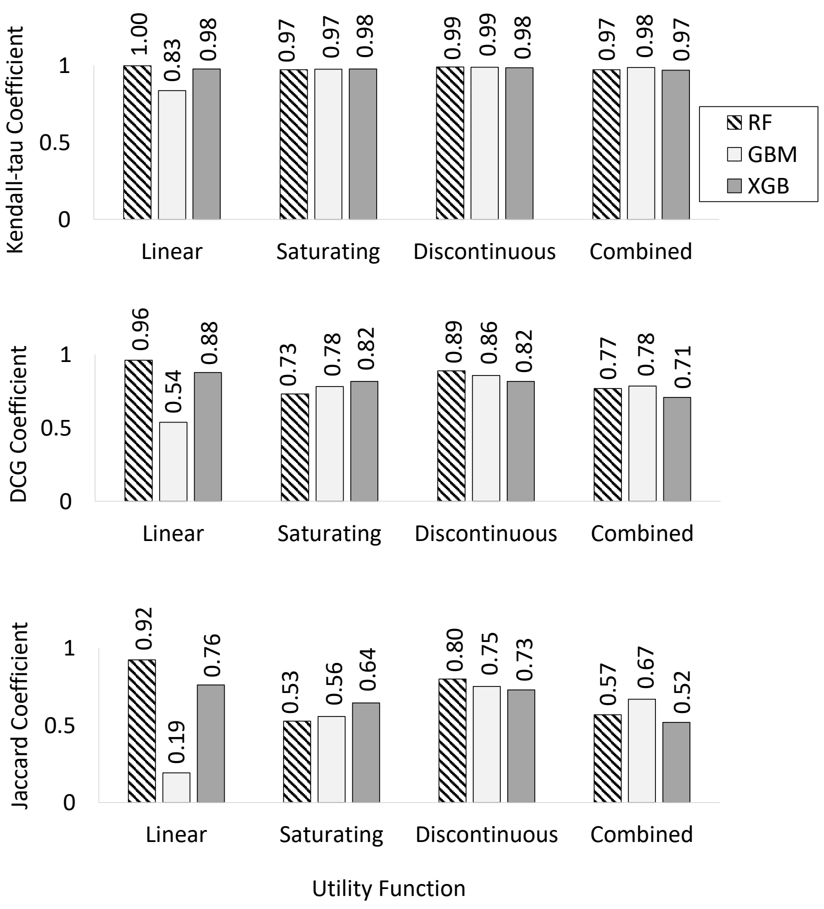


Figure 8.4: Similarity aggregation values for optimal and predicted decisions produced by selected prediction models.

method has a larger impact on runtime effort than the choice of utility function. Since distinct utility complexities imply different sets of features, one possible explanation for the results in Figure 8.3 is that, the number of features has a smaller impact on model size than the hyper-parameter tuning.

Answering Q8.1: We confirmed that (i) larger dataset sizes provide lower MADP values for all prediction models and (ii) prediction models can be tuned to minimize prediction error and runtime effort.

8.1.3 *Step 4: Select Prediction Model*

In this section, we select the prediction models based on their produced adaptation decisions as outlined in Section 3.3.3.7. We select the models based on the similarities between the optimal and the predicted list. The predicted lists are produced by executing the prediction models on the mRUBiS variants with 50 adaptation cycles, i.e., 50 MAPE-K loop executions, sampled from the *LRI* failure trace—see Table 8.1 for description of the trace. For each predicted list, we did the following steps to generate the corresponding optimal list: (1) inject the exact samples of *LRI* trace to mRUBiS and observe the utility-changes, i.e., $\hat{U}_{\Delta}$, (2) manually order the list in a descending manner regarding the utility-changes.

Table 8.3: SAM values for prediction models based on 9K data set.

Variant/Prediction Model	RF	GBM	XGB	Selected Model
Linear	0.959	0.521	0.871	RF
Saturating	0.742	0.770	0.812	XGB
Discontinuous	0.891	0.864	0.842	RF
Combined	0.809	0.812	0.731	GBM

The similarity measures, shown in Figure 8.4, present three distinct patterns: proportionate low values for GBM-Linear function, less discriminative values for Kendall-tau, and the same relative values for the DCG and Jaccard metrics. The lower values for GBM-Linear utility can be explained by the large difference in MADP between GBM versus RF and XGB models—see Figure 8.2. GBM model has MADP respectively from 10 to 30 times larger than of the RF and XGB models. This shows that a prediction error considered small, i.e., 0.58%, can still have a large impact on the quality of the adaptation decisions. The reason of this large impact is the simplicity of the Linear utility function, which produces utility-change values with a lower variance if compared to the more complex functions. Hence, even small errors are sufficient to alter the order of the adaptation decisions. This might suggest a surprising trade-off. While simpler utilities are easier to learn, i.e., small prediction error, the adaptation mechanisms deployed with these simpler utilities are orders of magnitude more sensitive to error variance in these models.

Kendall-tau values do not discriminate the prediction models as well as the DCG and Jaccard metrics. Except for the Linear utility, for all the other utility functions, the similarity differences for Kendall-tau are close to 1%—see Figure 8.4. Conversely, the DCG and Jaccard values range from at least 2% to 15%. Although these differences are not the same, DCG and Jaccard rank the prediction models in the same way. Ultimately, computing the similarity aggregation metric (SAM) values according to Equation 3.20 across all models, with $w_i = 1$, allows identifying the best performing prediction model—see highlighted cells in Table 8.3.

8.2 EVALUATION

We evaluate the quality of the results to answer the two remaining questions Q8.2 and Q8.3. The quality is relative to an optimal and a baseline benchmark. We report on the accuracy of the criteria used for model selection. To guarantee a fair comparison, we detail the data and system evaluation setup.

8.2.1 *Ground Truth and Experiment Design*

The existence of the *ground truth*, i.e., an optimal steering strategy, is usually not feasible for systems that come with a black-box performance model [383]. Conversely, in experiments with a simulator, instead of a real system, we have the benefit of accessing all the configurations and the possibility of undoing rule executions.

Comparison: A simulator at first enables us to deploy multiple prediction models under the same conditions and therefore, compare their performance in a fair manner. Secondly, in a simulator, we can observe the utility-changes that result from applying a rule and undoing the application step, i.e., acquiring the ground truth. This allows us to determine the *best* and the *worst* rule application and emulate accordingly the best possible strategy (Optimal) as well as the worst possible one (Baseline). Overall, we thus can consider for the comparison, the Baseline, the Optimal strategy, and the considered prediction models RF, GBM, and XGB. To facilitate model comparison, we normalize the reward values between zero and one as follows:

$$\mathit{Normalized\ Reward}\,(\mathit{mod}) = \frac{\mathit{Reward}\,(\mathit{mod}) - \mathit{Reward}\,(\mathit{Baseline})}{\mathit{Reward}\,(\mathit{Optimal}) - \mathit{Reward}\,(\mathit{Baseline})} \tag{8.1}$$

Experiment Design: The ground truth is generated by running four simulations of mRUBiS (100 tenants each), i.e., one simulation for each variant in Table 8.2[2]. With the same setup, we also determine the optimal strategy (Optimal) and the baseline (Baseline) that will serve as benchmark to evaluate the prediction models built and selected by the methodology. To mitigate bias in the evaluation, we inject the simulator variants with a real world failure trace *DEUG* that was not used in the steps 1-4. For each variant, we inject 50 bursts of failures that result in 50 executions of the MAPE-K loop. The number of the failures at each burst, i.e., the FGS, and the IAT follow the distributions in Table 8.1.

Variants: Table 8.2 describes the encoded utility function $\hat{U}$ in each mRUBiS variant and the details of each function can be found in Table 3.2. Note that the functions in Table 3.2 only presents the relevant fragment of $\hat{U}$ for each utility-change $\hat{U}_{\Delta}$ to be estimated by the prediction models.

Linear variant: We equip one of the mRUBiS simulators with a linear function with feature interaction [153], i.e., a polynomial of degree one.

Saturating variant: This variant of mRUBiS uses the Saturating utility function from Table 3.2 that extends the Linear variant by introducing the quality attribute performance to the formula.

Discontinuous variant: The Discontinuous variant conforms to the definition of the Discontinuous utility function for mRUBiS in Table 3.2 that includes disconnected intervals in the utility function.

Combined variant: Finally, we construct the fourth variant of mRUBiS with the Combined utility function that is a combination of the three formerly employed utility functions, thus the most complex variant. See Section 3.2.3 for more details on constructing utility functions for mRUBiS.

8.2.2 *Evaluating Prediction Model Performance - Q8.2*

To evaluate prediction model performance, we measure the normalized reward of the adaptations applied to the simulator variants and the runtime effort of the models. The normalized reward is calculated according to Equation 8.1 for two scalability scenarios: different datasets sizes and mRUBiS architectures sizes. Prediction models trained with larger dataset sizes acquire larger values of normalized reward. Figure 8.5 shows results

2 The experiments have been conducted on a machine with OS X 10.13, Intell processor 2.6 GHz core i5, and 8 GB of memory.

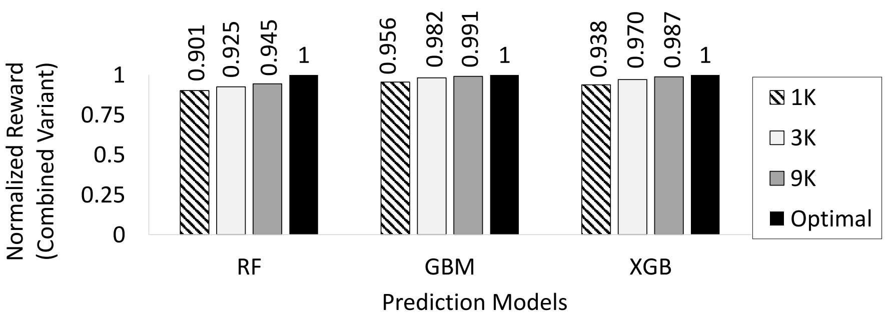


Figure 8.5: Normalized reward across prediction models for Combined variant computed with *DEUG* trace.

for the Combined variant. The same pattern is observed for the other three prediction models across all four variants—see Appendix B for the additional charts. The results in Figure 8.5 confirm our earlier finding for the prediction error, presented in Figure 8.1; the results also suggest that selecting the larger datasets, i.e., 9K, is a correct decision. The maximum reward loss of the 9K models compared to the optimal reward in Figure 8.5 is 5.5% and belongs to the RF model in the Combined variant (Normalized reward = 0.945).The reward loss in the other variants is below 5.5%—see the charts for Linear, Saturating, and Discontinuous variants in Appendix B.

As the architecture size grows, the normalized reward does not vary above 10%—see Table 8.4 for normalized reward of XGB with the 9K dataset for different sizes of architecture; similar patterns are observed for the RF and GBM models for dataset size 9K. To show this, we deployed the prediction models on mRUBiS simulators with different architecture sizes, i.e., 100, 500, and 1000 tenants. In addition to the *DEUG* trace, we injected the simulator variants with *LRI* trace. These results confirmed that the performance of the prediction models scales with architecture size. This also might suggest that we could transfer prediction models learned in smaller architectures to larger ones. This of course would require that the range of configuration and observation attributes, i.e., features, remain the same across architecture sizes.

The runtime effort scaled for the prediction models across all simulator variants. We evaluated this by measuring 20,000 executions of each prediction model in each simulator variant. The standard deviations of the outcome of each of these executions were below 0.5%. This confirms that the findings presented in Figure 8.3 are accurate.

Answering Q8.2: For the final prediction models trained with 9K datasets, the worst reward loss compared to an analytically defined optimum was 5.5%, i.e., a considerably smaller error than the 10% error bound.

Table 8.4: Normalized reward of XGB-9K for different architecture sizes across variants.

	DEUG Failure Trace				*LRI* Failure Trace			
# Tenant	Linear	Saturating	Discontinuous	Combined	Linear	Saturating	Discontinuous	Combined
100	0.998	0.996	0.990	0.987	0.997	0.996	0.992	0.990
500	0.997	0.997	0.990	0.987	0.997	0.996	0.992	0.991
1000	0.997	0.995	0.989	0.986	0.997	0.996	0.991	0.990

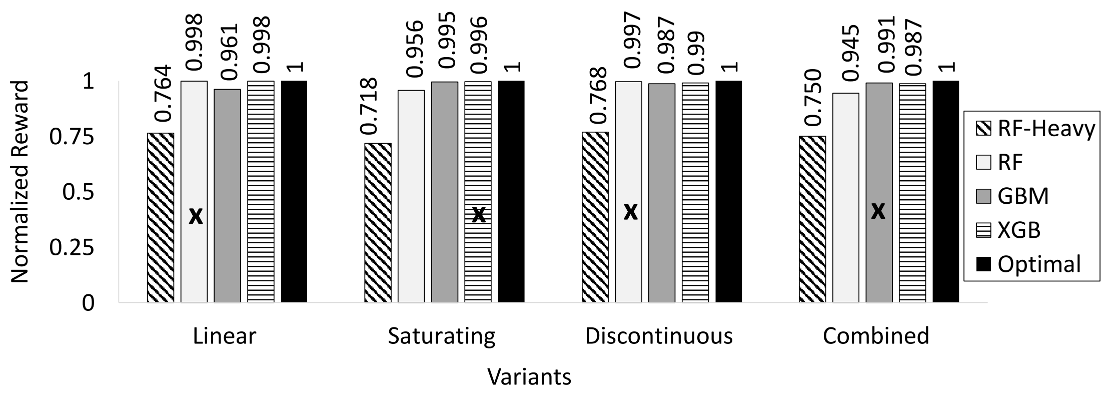


Figure 8.6: Normalized reward across prediction models (9K dataset is used).

8.2.3 *Evaluating Prediction Model Selection - Q8.3*

Figure 8.6 shows that prediction models with higher similarity metric values also obtain larger normalized reward. The prediction models that are selected based on the SAM values are marked with an "**X**"—see Table 8.3 for the SAM values. Comparing the normalized reward across all models in Figure 8.6, we can confirm that the similarity metrics lead to the choice of best performing models within each variant, i.e., largest normalized rewards. Moreover, although the RF-Heavy model has better prediction error (MADP) than the other RF model, the RF-Heavy model presents more than 10% loss in reward across all variants (see Figure 8.6). This confirms that during model tuning (step 1.2) and validation (step 3), it is sensible to trade slightly higher error rate for a smaller runtime effort.

Another observation is that the prediction model with the smallest runtime effort, i.e., XGB (see Figure 8.3), does not achieve the highest normalized reward according to the results in Figure 8.6. One possible explanation could be the effect of the different failure trace distributions. The average IAT between the groups of failures in *DEUG* trace, i.e., the idle time window to execute self-adaptation, is $1,543.26$ *sec*—see Table 8.1. To execute an adaptation that resolves all the failures injected in one cycle, the slowest prediction model, i.e., GBM model for the Combined variant in Figure 8.3, takes on average 38.18 *sec*; this means that, even for the slowest model, the overhead for each adaptation is on average 38.18 *sec*. Therefore, given the IAT of $1,543.26$ *sec*, this overhead does not have a long lasting impact on the reward. However, the impact of the overhead can be different in the presence of failure traces with different distributions.

Answering Q8.3: The best performing prediction models according to the results shown in Figure 8.6 are the same as the ones suggested by the SAM in Table 8.3. The selected models presented reward loss of less than 1% relative to the optimal reward. This confirms that our methodology, in (step-4), properly *selects* the best prediction models. Moreover, each of the lower performing models are correctly ranked, since they present the same ranking positions by SAM values in Table 8.3 as by normalized reward in Figure 8.6. This is possible because during the model tuning iterations, that is, steps 2, 3, and 1, the prediction models that could have potentially outperformed the alternatives were not discarded—see Section 8.1.2.

8.2.4 *Threats to Validity*

Internal validity. The similarity metrics that we considered for answering Q8.3 can be overly optimistic in the way that they are insensitive to the ranking of the components with the same utility-change, because we do not count as a mismatch the cases of components with the same utility-change, i.e., same ranking position. However, the case of repeated utility-change happened only for the Linear varinat of the utility function, which already presented low prediction errors. Using different similarity metrics or considering component types in addition to the utility-change value could mitigate this threat.
External validity. While our evaluation includes sensitivity analysis of the experiment outcomes, generalization of the results beyond the scope of this thesis is the main threat to external validity. Four factors hinder the generalization of our findings: considered utility models, size/quality of the employed input traces, the specification of the components/tenants, and the available adaptation rules. For a different setup of these factors, the current prediction errors might be unacceptable (even when small). This might happen in the case that adaptation decisions must present variances in utility that are smaller than the prediction errors. Applying the prediction models to different systems with different architecture styles can mitigate this threat.
Construct validity. The main threat to construct validity is the correctness of the implementation of the mRUBiS simulator. Errors in the mRUBiS simulator with respect to computing the utility of the architecture might lead to wrong measurements of the utility-change as the ground truth, thus affecting the predictions. The simulator is a exemplar from the *Symposium on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS) community that is publicly available [414] and we have several past experiences with mRUBiS from earlier work [175, 180, 181]. Slight error in calculating the distance by the employed similarity metrics might result in completely different rankings and affect the ordering of the rules. The distance metrics might not always reflect the ranking errors. For instance, if each utility prediction, instead of an absolute value, is a probability distribution, we would need a different metric that takes into account the uncertainty of the utility while ranking.
Statistical conclusion validity Slightly different input traces can cause completely different adaptation outcomes. Therefore, we carefully generated and tested failure traces to be *Independent and Identically Distributed* (IID) across the different ranges of dataset sizes and utility functions. This involved testing the generated traces to detect inconsistent output from the simulator, e.g., zero utility values. Inconsistencies are part of the inherent randomness of a simulation of a real system. Another threat was the implicit assumption that the data was IID while we did not make a parametric assumption, e.g., normality or heteroskedasticity. This threat is mitigated by the fact that the employed simulator is memory-less across the execution rounds and the traces are injected independently and uniformly across the architecture.

8.3 SUMMARY

This chapter presented an application followed by a quantitative evaluation of our proposed methodology to train and select prediction models for utility-change in rule-based, self-adaptive systems. We investigated if it is possible to learn instead of constructing the system utility that steers rule-based adaptation in self-adaptive systems with dynamic architectures. Our approach presented in Section 3.3 was to design a methodology that extends the standard machine learning process. The results in this chapter were promising and can be summarized in the answers to three questions:

Q8.1: *Is it possible to systematically learn a prediction model for the utility-changes in a utility-driven, rule-based, self-adaptive system (particularly when the detailed knowledge of the system is not available)?* it was possible to systematically learn a prediction model for the utility-changes even when the detailed knowledge of the system was not available.

Q8.2: *Do the prediction models lead to a system performance that approximates an analytically-defined optimum (real loss lower than* 10%*), considering that different arbitrary bounds could be chosen to trade between model accuracy and model runtime effort?* the obtained prediction models led to a performance that was equivalent to the optimal. The maximal loss of performance compared with the optimal solution was 5.5%, which is considerably lower than the 10% upper-bound used as a stopping criterion in our methodology. This suggests that it might still be possible to obtain good performing models by accepting models with prediction errors larger than 10%. We also showed that the prediction models scaled for larger architectures and over multiple adaptation cycles.

Q8.3: *Can the suggested methodology properly* select *the best prediction model without requiring the models to be deployed on a real system?* our methodology always selected the prediction model with the best performance. Moreover, even without the information about the final system performance, our methodology was able to correctly rank all alternative prediction models.

While our evaluation in this chapter was limited to one architecture, we offered a general methodology and evaluated it with real failure traces under different scalability scenarios. The evaluation in this chapter was based on our evaluation guidelines presented in Section 6.3. We conducted a sensitivity analysis of our results by investigating the methodology over a spectrum of utility function complexities, machine learning methods, and dataset sizes. Finally, this chapter provided supporting empirical evidence for the following contributions of the thesis mentioned in Section 1.4:

C3 Training utility-change prediction models for rule-based self-adaptive software

C8 Coverage of a wide spectrum of self-adaptation problem space beyond the state-of-the-art

8.3.1 *Fulfillment of Requirements*

Based on the overall discussion and the answer provided for Q8.1 in this chapter, we showed that it was possible to systematically learn a prediction model for the utility-changes in a utility-driven rule-based self-adaptive system, even when the detailed knowledge of the system was not available. Therefore, this chapter confirmed that our proposed methodology to train utility-change prediction models for architecture-based self-adaptive software fulfills requirement $\mathcal{R}6$ in Table 1.1.

9 EVALUATION OF HYPEZON

This chapter includes the quantitative evaluation of HypeZon, the hybrid solution for self-adaptation introduced in Chapter 5. We study the effectiveness of the two designs of HypeZon, i.e., HZ_i and HZ_e, in a comparative study with a deterministic hybrid planner using `Znn.com` and mRUBiS application examples. Based on the empirical evaluation, this chapter aims at answering the following questions:

Q 9.1 *how do internal and external designs for meta-self-awareness affect* HypeZon*?*

Q 9.2 *how does* HypeZon *perform in comparison to a deterministic hybrid planner?*

Q 9.3 *what are the effects of hybrid planning*[1] *on the reward and timeliness of the adaptation?*

In Section 9.1, we discuss the instantiation of HypeZon based on our implementation setup. In Section 9.2, we introduce the alternative solutions for hybrid adaptation. Section 9.3 presents the experiments and discusses the results by answering the three questions. In Section 9.4, we discuss the threats to the validity of the results and finally, Section 9.5 summarizes this chapter and discusses the fulfillment of the requirements.

9.1 IMPLEMENTATION

HypeZon builds on the Monitor, Analyze, and Execute activities of the incremental adaptation engine presented in Section 4.3 and offers an alternative solution for the Plan activity. This way, the incremental calculation of utility-changes as the impact of the adaptation rules are also available to HypeZon—see Figure 5.1 for a schematic overview. The implementation of HypeZon is based on the EMF. The deployment of HypeZon on an adaptable software conforms to the implementation setup presented in Section 7.1. The difference is that, in the context of HypeZon, the Adaptation Engine in Figure 7.1 is implemented with meta-self-aware properties—see Figure 5.1. Figure 9.1 shows the deployment of the hybrid planning solutions for evaluation in this chapter. The two variants of HypeZon, HZ_e and HZ_i, are implemented according to Figure 5.6a and Figure 5.6b respectively. The *Awareness-level* MAPE activities in both HypeZon variants that constitute the adaptation loop in Figure 5.6 are in charge of adapting the system and replicating the same MDE-based scheme as discussed in Section 7.1. Hence, the activities of the adaptation loop in HypeZon variants they use SDs to analyze, plan, and execute adaptations on an architectural RTM of the system. The SDs are defined according to the context, i.e., the adaptable software and the potential adaptation issues. Thus, apart from the meta-self-aware design of the loops in the HypeZon variants, the adaptation mechanism in the polices follows the same incremental event-based scheme as we used to implement Venus and the other incremental alternative self-adaptation approaches, i.e., *Solver* and *Static* in Section 7.2.

1 We use the terms hybrid (self-) adaptation and hybrid planning interchangeably.

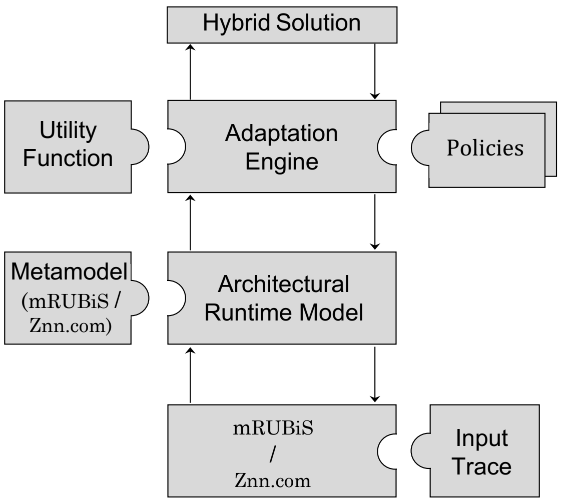


Figure 9.1: Implementation decomposition for evaluation of hybrid solutions.

In the context of HZ_e, the *Meta-awareness-level* loop in Figure 5.6a has a purely code-based implementation in Java, i.e., excluding the SDM tools. The code contains implementation of the hybrid planning algorithm shown in ALGORITHM 5.1–5.3. Upon each execution interval $\mathcal{I}$, the execution of the code is invoked by the Plan activity in the awareness-level loop. See Figure 5.8 for the sequence diagram for HZ_e execution. In the context of HZ_i, detailed in Figure 5.6b, the meta-self-aware self-loop and its corresponding activities that realize ALGORITHM 5.1–5.3 also have a pure code-based implementation. The meta-self-aware loop in HZ_i, in contrary to the one in HZ_e, is realized by code snippets that wrap around the MAPE loop activities at the awareness-level, consequently, the meta-self-aware loop is invoked for execution at the same frequency as the adaptation loop runs. See Figure 5.9 for the sequence diagram for HZ_i execution.

9.2 ALTERNATIVE SOLUTIONS FOR HYBRID ADAPTATION

We evaluate two different variants of HypeZon in a comparative study against each other as well as a home-developed *deterministic* hybrid planner. In the following, we briefly introduce the deterministic solution and summarize the characteristics of the three alternative solutions for hybrid planning.

Deterministic Solution. We implemented a deterministic, coordinating hybrid planner that uses predefined, fixed thresholds on quality attributes of interest, e.g., response time, as constraints. Violation of the constraints indicates a trigger for policy switch. The proposed hybrid planner, Chp_{dtr} henceforth, does not support runtime adjustments of its control parameters and considers look-ahead, planning, and execution horizons with fixed sizes and, as a result, has smaller planning overhead at runtime. ALGORITHM 9.1 shows a simplified, high-level description of Chp_{dtr}. Chp_{dtr} takes set of available policies Π and the current response time RT_{curr} as inputs. Similar to ALGORITHM 5.1, for a planning problem captured in the planning horizon Φ, Π also includes estimations of $\hat{U}$ and $\hat{C}$ for the available policies. Chp_{dtr} also maintains the *current* response time of the system, i.e., RT_{curr} (in contrast to the *average* response time $RT_{\mathcal{I}}$ in ALGORITHM 5.1). Based on the specific objectives of the system, Chp_{dtr} defines a threshold, namely *high*, for RT_{curr}. Reaching the threshold indicates that RT_{curr} is higher than the permitted

ALGORITHM 9.1 : $\textsc{Chp}_{dtr}$

```
1  Require: $\Pi$, $RT_{curr}$
2  $\Phi \leftarrow \mathcal{L}$          // Planning horizon gets all issues in look-ahead horizon
3  $\pi^* \leftarrow null$
4  if $RT_{curr}$ high then                          // Objective not satisfied
5      $\pi^* \leftarrow \pi_j \in \Pi : j = \operatorname{argmin}_j \hat{C}(\pi_j(\Phi))$     // Policy to minimize cost
6  else
7      $\pi^* \leftarrow \pi_i \in \Pi : i = \operatorname{argmax}_i \hat{U}(\pi_i(\Phi))$     // Policy to maximize utility
8  end
9  List of Actions $\leftarrow \pi^*(\Phi)$
10 Return List of Actions
```

upper bound. Otherwise, RT_{curr} is within the acceptable range. The threshold is defined at design time, thus is deterministic and not subject to change. If RT_{curr} is *high*, $\textsc{Chp}_{dtr}$ switches to a policy with minimum planning time captured by $\hat{C}(\pi(\Phi))$ (line 5 in ALGORITHM 9.1), otherwise, a more time-intensive policy that has the highest plan utility $\hat{U}(\pi(\Phi))$ is selected (line 7). $\textsc{Chp}_{dtr}$ is proposed only as an algorithm, thus does not require any specific architectural modifications to the conventional MAPE-K loop and is realized during the Plan activity of the adaptation loop.

HypeZon Solutions. We proposed two different designs to realize hybrid planning with HypeZon in Section 5.3.2, i.e., HZ_e and HZ_i. Both variants implement the hybrid planning with receding horizon control as described in ALGORITHM 5.1-5.3—see Section 5.3.3 for details of the inner working of the variants. Table 9.1 summarizes the characteristics of the alternative solutions for hybrid planning.

9.3 EVALUATION

In this section, we present a comparative study of the alternative solutions for hybrid self-adaptation and answer the questions Q9.1–Q9.3. First, we introduce the off–the-shelf adaptation policies used by the hybrid solutions—see Figure 9.1 for reference; next, we present the experiment design for the application examples followed by the quantitative evaluation of the solutions.

Table 9.1: Characteristics of hybrid solutions.

Property of Interest	**Solution**		
	$\textsc{Chp}_{dtr}$	HZ_e	HZ_i
Control parameter tuning	Design-time	Runtime	Runtime
$\mathcal{L}$, Φ, H_n	∞	Adjustable	Adjustable
Policy switch criteria	Deterministic	Dynamic	Dynamic
Implementation	Algorithmic	Architectural design	Architectural design
Self-awareness scope	NA	Global	Local
Execution intervals	per adaptation cycle	Adjustable	per adaptation cycle

9.3.1 *Policies*

The hybrid planners in our experiments coordinate two different off-the-shelf policies for self-adaptation. As individual policies, we used *Static* and *Solver*, introduced in Section 7.2. We equip the adaptable software, i.e., mRUBiS and `Znn.com`, with self-adaptation capabilities using either *Static* or *Solver* solutions. The policies conform to the principals of static and dynamic decision-making, as discussed in Section 2.3.2, respectively. *Static* uses design time estimations for the expected utility. Therefore, at each state s, for each applicable rule r, the expected effect on the utility is predetermined. We have shown in Chapter 7 that this policy is sub-optimal in terms of utility and reward but fast in terms of adaptation time. Regarding *Solver*, the results in Chapter 7 suggest that while this policy finds the optimal target state at each adaptation step, it can presents long planning times. Runtime information are used to switch between *Static* and *Solver*. For example, during surges of client traffic in `Znn.com` where more issues such as `latency` are likely to occur, employing *Static* that can provide timely rather than optimal adaptation plans is beneficial. However, once the traffic surge calms down, the utilization of the servers can be optimized by switching to *Solver*—see Section 7.2 for detailed descriptions of *Static* and *Solver*. Switching between policies has a cost as it requires deploying specific settings for the new policy, e.g., initializing a constraint solver or loading prediction models. The switch from policy π_i to π_j is charged with a cost c_{ij} that is subtracted from the system utility. The switch from policy *Solver* to *Static* is charged with a cost $c_{slv,stc} = 2$ and $c_{stc,slv} = 200$ applies to the switch in the opposite direction. The considered costs are independent of the runtime conditions and are defined relatively and based on measurements of the policy deployment time.

9.3.2 *Experiment Design*

mRUBiS. We discussed in Section 5.1 an exemplary scenario for mRUBiS application where a single adaptation solution is incommensurate to fulfill system objectives. In this section, we use mRUBiS with 1000 tenants as one of the application examples for hybrid planning. For each state s, the utility of the mRUBiS architecture, henceforth $U_{mRUBiS}(s)$, is the sum of the utility of all its tenants—see Equation 3.2. Similarly, we define the utility of a tenant as the sum of the utility of its 18 constituent components as Equation 5.1. The utility of a component is computed according to Saturating utility function in Table 3.2 with $s = (\bar{c}, \bar{o})$ and $\bar{c} =< connectivity, reliability, criticality >$ and $\bar{o} =< request, replica >$—see Section 3.2.3 for details of the Saturating utility function for mRUBiS. The employed utility function for mRUBiS considers the performance $\mathcal{P}$ of a component defined as Equation 3.9. Each component in mRUBiS has a unique $\mathcal{P}_{max}$ to which the performance of the component saturates. As input traces for mRUBiS, we use the realistic failure models from Table 6.3. We inject mRUBiS with issues of type CF and PI—see the metamodel of mRUBiS in Figure 2.8 for details of the supported issues, adaptation rules, and their attributes.

`Znn.com`. The second employed case study is `Znn.com`—see Section 6.2. We instantiated a version of `Znn.com` with 20 severs, where only 15 are considered within the budget—see Figure 6.4 for instance of an SD capturing the `overBudget` issue in `Znn.com`. The request arrival traces are generated based on the realistic failure traces, i.e., web traffic logs of FIFA 98 world cup site [17] introduced in Section 6.3.2.2, and are employed as the input traffic for `Znn.com`. Out of the nine realistic traces in Section 6.3.2.2, we consider three *long* traces, i.e., Day 10, Day 50, and Day 90—see Figure 6.9. We also consider *short* traces of Day 30 and Day 80—see Figure 6.8. The traces include clients web content requests from the web servers over the course of 24 hrs (*long* traces) and 60 min (*short* traces). For each state $s = (\bar{c}, \bar{o})$, the overall utility of `Znn.com`, i.e., $U_{znn}(s)$, is defined according to Equation 6.1.

The sampling interval $\mathcal{I}$ for HZ_e indicates that the meta-awareness loop in HZ_e is executed once for every $\mathcal{I}$ executions of the adaptation loop. In HZ_i, however, $\mathcal{I} = 1$ as the meta-awareness is embedded in the adaptation loop and is executed at the same frequency as the adaptation loop—see Figure 5.8 and Figure 5.9 for execution sequence diagrams of HZ_e and HZ_i, respectively. The experiments are repeated and averaged over 1000 simulation runs under controlled and reproducible conditions. The reported values are normalized accumulated utility over time, i.e., normalized reward with $Normalized\ Reward = U_{znn} - c_{ij}$ for `Znn.com` and $Normalized\ Reward = U_{mRUBiS} - c_{ij}$ for mRUBiS. For experiments with `Znn.com`, we consider the following initial values for the two zones on the request response time RT; $10\ sec < RT$ is associated to *high* and $0 < RT \leqslant 10\ sec$ corresponds to the *optimal* range—see ALGORITHM 5.1. CHP$_{dtr}$ uses the same threshold as *high*—see ALGORITHM 9.1.

In experiments with mRUBiS, instead of response time RT, we use performance $\mathcal{P}$ of the services as indicators for system objective satisfaction—see the decision node in the flowchart describing hybrid planning with HYPEZON in Figure 5.7. Thus, ALGORITHM 5.1 and ALGORITHM 5.2 replace $RT_{\mathcal{I}}$ and $\hat{RT}$ with $\mathcal{P}_{\mathcal{I}}$ and $\hat{\mathcal{P}}$ respectively; $\mathcal{P}_{\mathcal{I}}$ is the average performance of the system during the sampling interval $\mathcal{I}$ and $\hat{\mathcal{P}}$ is the estimated system performance for the current adaptation cycle that is approximated using linear regression based on $\mathcal{P}_{\mathcal{I}}$ and expected system load, i.e., $\hat{load}$. Similarly, ALGORITHM 9.1 replaces RT_{curr} with $\mathcal{P}_{curr}$, with $\mathcal{P}_{curr}$ the current performance of the system. Consequently, based on the specific business objectives of mRUBiS, we define two zones for performance: *low* range defined as $0.1 \times \mathcal{P}_{max}$ and the *optimal* range indicates that $\mathcal{P}_{\mathcal{I}}$ is within the acceptable range, i.e., $0.1 \times \mathcal{P}_{max} < \mathcal{P}_{\mathcal{I}}$. CHP$_{dtr}$ uses the same value for the *low* range—see ALGORITHM 9.1. Note the change of the term *high* to represent undesirable response time values to the term *low* in the context of performance to capture unsatisfactory system performance. Therefore, in ALGORITHM 5.1, ALGORITHM 5.2, and ALGORITHM 9.1, in experiments with mRUBiS, the range *high* for RT is replaced by *low* for $\mathcal{P}$.

Note that the threshold values in the HYPEZON variants, for both mRUBiS and `Znn.com`, are not deterministic and may change at runtime. The size of the look-ahead, planning, and execution horizons in CHP$_{dtr}$ is set to ∞, thus CHP$_{dtr}$ plans for *all* the existing issues and executes the *complete* plan—see Table 9.1. In the experiments with mRUBiS, the upper bound *high* in ALGORITHM 5.3 is initialized with 35 and 200 for $\hat{load}$ and $\hat{C}(\pi^*(\Phi))$; in the experiments with `Znn.com`, 500 and 80 are used respectively. The values are selected based on empirical profiling and are only the initializing values and thus subject to change during the experiments.

9.3.3 *Results: Answering Q9.1, Q9.2, and Q9.3*

Next, we evaluate HYPEZON variants in comparison to CHP$_{dtr}$ via a set of quantitative experiments. We structure the section by answering the three questions from the beginning of this chapter.

Q9.1: how do internal and external designs for meta-self-awareness affect HypeZon? Table 9.2 shows the normalized reward during 24 hrs for `Znn.com` (top) and mRUBiS (bottom). *Solver* and *Static* perform simple adaptations without any hybrid planning thus $c_{ij} = 0$. HZ_e is executed with $\mathcal{I} = 5$. Table 9.3 presents the sensitivity analysis of the results of HZ_e with different execution intervals $\mathcal{I}$. In our experiments, HZ_e with $\mathcal{I} = 1$ is identical to HZ_i. The results in Table 9.2 suggest that in majority of the experiments, HZ_e achieves higher reward compared to HZ_i. The reason is that HZ_e has relatively larger execution intervals $\mathcal{I}$ that provides an extended monitoring period. HZ_e makes decisions about setting its control parameters and policy switch based on the estimations over this extended monitoring period. Larger execution timescale of the meta-awareness loop in HZ_e results in more history, i.e., accumulated observations and experiences, consequently, HZ_e makes more informed decisions. Conversely, HZ_i holds a localized view of the system load ($\hat{\mathrm{load}}$) that is limited to its relatively small monitoring period, i.e., since the last execution of the adaptation loop, and sets its control parameters and switches the employed policies accordingly. Therefore, the relatively small execution timescale of the meta-awareness loop in HZ_i may lead to premature decisions due to insufficient and localized information and as a result, the hybrid planner is likely to demonstrate nervous and volatile behavior regarding policy switch decisions.

In the experiment with *Grid*5000 for mRUBiS, HZ_i outperforms HZ_e—see Table 9.2. The reason is that the IAT of the adaptation issues in this trace is significantly smaller compared to the other traces–see Table 6.3. Smaller IATs between issues demand more frequent planning. In this case, as discussed in Section 2.1.2, more frequent control calculations are beneficial since the system and its context change in high pace. Therefore, when the system experiences adaptation issues at high arrival rate, smaller sampling intervals $\mathcal{I}$ for the controller are more beneficial since HypeZon variants adjust their control parameters at each sampling interval. Results in Table 9.2 and Table 9.3 suggest that the reward of hybrid planners is affected by the characteristics of the input traces. Moreover, as confirmed by Table 9.3, larger values for execution intervals of HZ_e result in sub-optimal adaptations. HZ_e with $\mathcal{I} = 15$ achieves only 25% and 34% of the optimal utility for Day 10 and *LRI* respectively.

Figure 9.2 depicts the points in time that HZ_e and HZ_i make a policy switch over the course of 60 min with `Znn.com` and Day 80 trace. HZ_e is executed with $\mathcal{I} = 5$. The gray curve in Figure 9.2 represents client request distribution in Day 80 trace—see Figure 6.8.

Table 9.2: Normalized reward over 24 hrs for `Znn.com` (top) and mRUBiS (bottom).

Trace	HZ_i	HZ_e	CHP_{dtr}	*Solver*	*Static*
Day 10	0.79	1	0.81	0.65	0.53
Day 50	0.55	1	0.85	0.49	0.38
Day 90	0.76	0.83	1	0.7	0.52
*Grid*5000	1	0.88	0.65	0.47	0.54
LRI	0.72	1	0.79	0.51	0.32
DEUG	0.83	0.97	1	0.47	0.43

Table 9.3: Normalized reward of HZ_e with different $\mathcal{I}$ over 24 hours.

System-Input trace	$\mathcal{I} = 1$	$\mathcal{I} = 2$	$\mathcal{I} = 3$	$\mathcal{I} = 4$	$\mathcal{I} = 5$	$\mathcal{I} = 10$	$\mathcal{I} = 15$
Znn.com - Day 10	0.79	0.81	0.85	0.92	1	0.43	0.25
mRUBiS- *LRI*	0.65	0.73	1	0.93	0.9	0.6	0.34

HZ_i switches between its constituent policies more often compared to HZ_e that only switches three times (see the square markers) during 60 min. As suggested by the majority of the measurements in Table 9.2, compared to HZ_e, the relatively more frequent policy switches in HZ_i result in reward loss. As shown in Figure 9.2, between minute 0 and minute 20, HZ_e switches *once* in response to the relatively small turbulence in the number of the requests; HZ_i in contrast, due to its localized view of the system load, switches between the policies more frequently. The same pattern is observed for HZ_i between minute 38 and 50.

Answering Q9.1: The results in Table 9.2 and Figure 9.2 suggest that the relatively small execution timescale of the hybrid planner in HZ_i may lead to premature decisions due to insufficient and localized information, and, as a result, the system is likely to demonstrate nervous behavior. However, during volatile operation conditions when the system is experiencing rapid changes, e.g., when IAT between the failures is small, more frequent control calculations are beneficial since the system and its context change in high pace. Therefore, smaller sampling intervals $\mathcal{I}$ for the controller would be more effective.

Q9.2: how does HypeZon perform in comparison to a deterministic hybrid planner?

Table 9.2 shows that for Day 90 in Znn.com and *DEUG* in mRUBiS, CHP_{dtr} obtains higher reward compared to the HypeZon variants. Analysis of Day 90 characteristics revealed that the average request arrival rate during Day 90 is 69% of the rate of Day 10 and 42% of the rate of Day 50. Similarly, the IAT between the bursts in *DEUG* trace is significantly larger than the alternative traces used for mRUBiS—see distribution for *DEUG* in Table 6.3. The results together with the analysis of the input trace characteristics confirm that runtime conditions, i.e., the characteristics of the input traces, significantly affect the reward of the adaptation. The results in Table 9.2 show that in four traces the HypeZon variants outperformed CHP_{dtr}. For the two traces with less

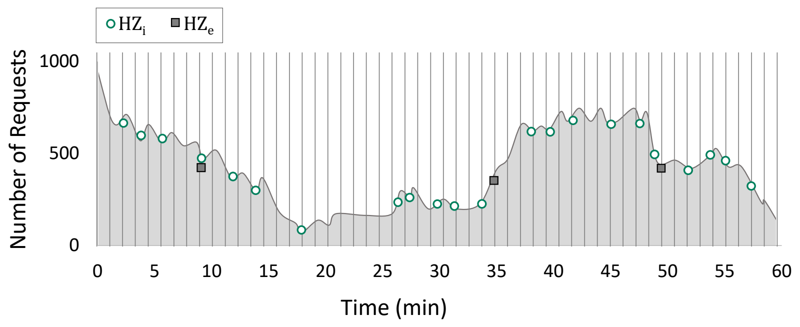


Figure 9.2: Policy switch decisions by HZ_i and HZ_e in Znn.com.

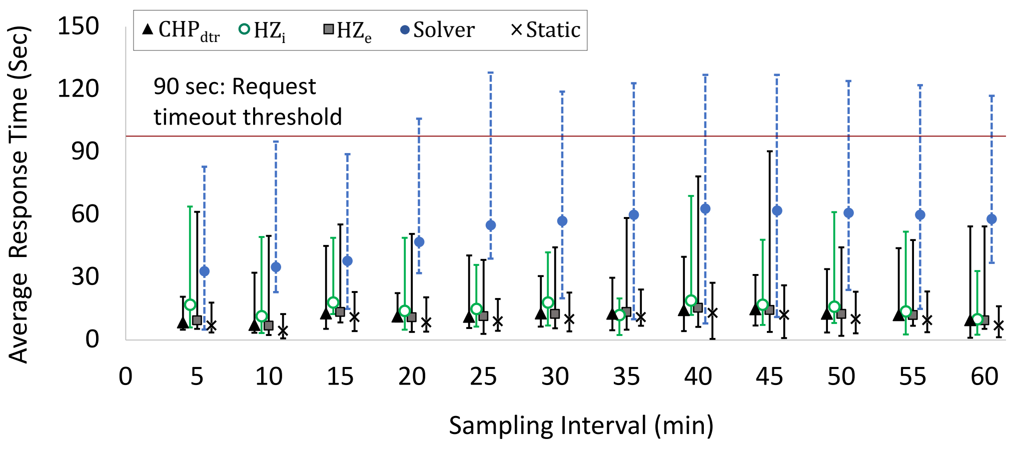


Figure 9.3: Average request response time (sec) in `Znn.com`.

extreme characteristics, i.e., Day 90 and *DEUG*, the pre-defined values of the control parameters and thresholds in CHP$_{dtr}$ are beneficial and outperform the HYPEZON variants. Overall, results suggest that, compared to CHP$_{dtr}$, HYPEZON variants are more effective in copping with the varying operation conditions, e.g., input traces with more extreme characteristics.

Figure 9.3 presents average request response times for clients in `Znn.com` during 60 min with Day 30 (*short*) trace. HZ$_e$ is executed with $\mathcal{I} = 5$. Each measurement at sampling interval $\mathcal{I}$ shows the average response times (bullets) as well as the maximum and minimum response times (vertical bars) during the last 5 min. Note that the hybrid planning overhead also affects the response time values. Table 9.4 shows the corresponding normalized reward of the planners for the same experiment. CHP$_{dtr}$ uses predefined and deterministic values for the control parameters, thus compared to the HYPEZON variants, has a smaller planning overhead and presents smaller response times. In addition to the three hybrid planners, Figure 9.3 includes the response times of their constituent adaptation policies, i.e., *Solver* and *Static*. *Static* has similar response times to CHP$_{dtr}$. This is due to the deterministic decision-making in CHP$_{dtr}$ that as soon as response time raises above 10 sec, CHP$_{dtr}$ switches to the *Static* solution for planning. Compared to CHP$_{dtr}$, HZ$_e$ has slightly higher average response times. However, as shown in Table 9.4, HZ$_e$ obtains 35% higher accumulated utility over 60 min in the same experiment. Response time values for HZ$_i$ are higher than HZ$_e$ and CHP$_{dtr}$. Despite its higher response times, Table 9.4 shows that HZ$_i$ obtains 14% higher accumulated utility compared to CHP$_{dtr}$. This suggests that the reward loss in CHP$_{dtr}$ cannot be compensated by its relatively short planning times.

Answering Q9.2: For the considered scenarios, in majority of the cases HYPEZON variants outperform CHP$_{dtr}$ in terms of system reward—see Table 9.2 and Table 9.4. Compared to CHP$_{dtr}$, HYPEZON variants are more beneficial in copping with the vary-

Table 9.4: Normalized reward over 60 min for `Znn.com`.

HZ$_i$	HZ$_e$	CHP$_{dtr}$	*Solver*	*Static*
0.79	1	0.65	0.59	0.51

ing operation conditions, e.g., input traces with more extreme characteristics. In two of the experiments with more uniformly distributed input traces CHP$_{dtr}$ outperformed HYPEZON, therefore, we can conclude that more stable operation conditions where traffic surges or large and rapid failure bursts are not expected can be handled by a simpler planner like CHP$_{dtr}$ with deterministic parameter setting. In the context of self-adaptive systems however such an assumption for the input space of the system is not realistic. Additionally, CHP$_{dtr}$ has only a negligible runtime overhead in terms of operation cost and can be executed more cost-efficiently compared to HYPEZON variants—see Figure 9.3.

Q9.3: what are the effects of hybrid planning on the reward and timeliness of adaptation? For both application examples, the normalized reward values in Table 9.2 suggest that hybrid planning approaches improve the reward of the system compared to their constituent policies. *Solver* in Figure 9.3 has the highest *average* response time values, it also presents significantly high *maximum* response times and in majority of the measurements, encounters timeouts, i.e., from minute 20 to 60. While results in Figure 9.3 show relatively low response times for *Static*, according to Table 9.4, it obtains only 51% of the maximum reward in HZ$_e$.

The non-hybrid solutions that employ a single policy for planning are likely to exhibit sub-optimal behavior at runtime caused by their varying operation conditions. We have shown in our work [see 177] that the choice of the adaptation policy in a self-adaptive system should be steered with respect to the characteristics of the input trace, otherwise, the employed policy may render sub-optimal at runtime. Thus, as also confirmed by our empirical evaluation in this chapter, hybrid planners, either with deterministic or adjustable parameters, results in improvements of the reward (Table 9.2 and Table 9.4) and the timeliness (Figure 9.3) of the adaptation.

Finally, to validate our hypothesis regarding the behavioral patterns of different self-adaptation solutions in the face of increasing adaptation complexity, illustrated in Figure 1.1 and discussed again in Figure 5.2, we ran a series of experiments on mRUBiS with 1000 tenants; we injected failures according to the deterministic failure models introduced in Section 6.3.1.1. The experiments include *four* subsequent MAPE-K runs; before each adaptation cycle, mRUBiS is injected with one input trace; the traces are ordered in increasing size simulating the growing complexity of the operation condition as depicted in Figure 1.1. Capturing the complexity of the multi-dimensional adaptation space as points across the x-axis is a an oversimplification of the phenomenon. Here, we perform a series of experiments during which we maintain all the dimensions of the adaptation space constant and emulate the increasing complexity of the adaptation by step-wise increasing the complexity of the adaptation setup, i.e., increasing system load. Through out the experiments, we use the same model of the mRUBiS architecture with the same characteristics and size; the only varying parameter is the input trace that defines the number and the type of the issues that are injected to the system. As the adaptation solutions we use *Static*, *Solver*, HYPEZON (HZ$_i$), and VENUS. HYPEZON uses *Static* and *Solver* as its constituent policies for runtime coordination.

We inject mRUBiS with one of the traces, execute the approaches for adapting the system and resolving the issues, measure the overall accumulated utility over 5 min of system execution, i.e., reward, then we inject the next trace. The reward values are normalized. In order to have a baseline, we also report system reward when there are no issues injected, i.e., before the first MAPE-K run. Figure 9.4 shows the normalized

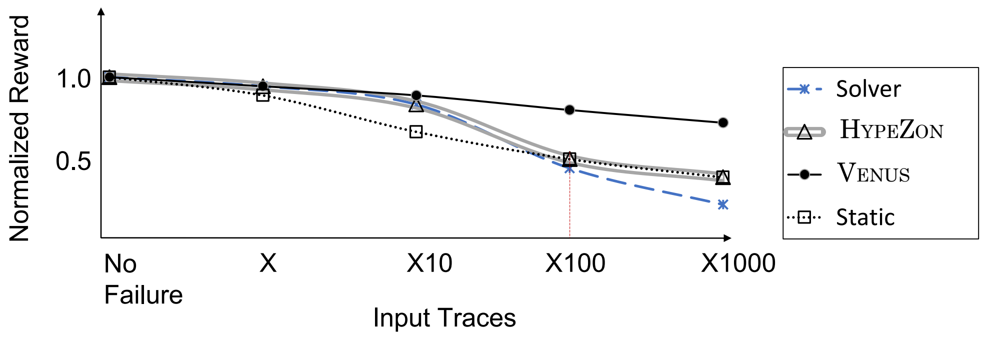


Figure 9.4: Normalized reward of self-adaptation solutions in mRUBiS with growing complexity of input traces.

reward of the approaches as the complexity of the input trace increases. We executed HZ_i varinat of HypeZon, therefore, the hybrid planner is executed before every MAPE-K loop run. The red line on the x-axis (next to X100) marks the point where HypeZon switches to *Static*. The measurements are discrete points captured by the markers in Figure 9.4, the points are only connected via the lines to visualize the change pattern and for comparison against the schematic chart presented in Figure 1.1 and Figure 5.2.

As the complexity of the adaptation input increases, i.e., there are more issues to resolve, the reward of all approaches is reduced. HypeZon selects the best-performing policy before each adaptation. For X100, HypeZon switches to *Static*—see the switch at X_2 in Figure 1.1. As discussed earlier, a coordination-based hybrid solution like HypeZon that composes off-the-shelf policies at runtime performs only as good as its best performing constituent. A customized combination-based solution, however, e.g., Venus, may combine policy formalism of the individual solutions in its design to construct a new policy. The resulting adaptation solution can leverage the strength of all the constituents, thus outperforming them in terms of cost minimization and quality maximization. The results for Venus in Figure 9.4 confirm this.

Answering Q9.3: employing a hybrid planner, either with deterministic or adjustable dynamic parameter setting, results in improvements of the reward and timeliness of the adaptation.

9.4 THREATS TO VALIDITY

Internal validity. The main threat to internal validity is the parameter setting and the explored input traces for the experiments. While our evaluation includes sensitivity analysis of the outcomes in some experiments, different control parameters and traces may lead to different results. The performance of the hybrid planners may depend on system parameters, e.g., server costs in `Znn.com`, or thresholds for the switching and adjustments in the algorithms. This threat is mitigated by carefully reproducing identical experiment setting for the comparative studies. To mitigate the risk of confounding the outcome of different planning policies, we established a set of controls. These controls were threefold: (i) same monitoring, analysis, and execution phases, (ii) same architectural RTM, and (iii) same utility functions. We further mitigated the threats to internal validity by considering two different application examples, three different hybrid planners, and a sizable set of input traces with substantial variation, which leads to a robust assessment of planner performance through cross-validation.

External validity. We present a hybrid solution for self-adaptation that coordinates arbitrary off-the-shelf adaptation policies. Because our evaluation considers coordinating specific policies, the generalization of the results beyond the scope of this study is the main threat to external validity. Although we present a generic solution for hybrid adaptation that leverages control-theoretical principles, our evaluation focuses on investigating the reward and response times in two case studies. Thus, we cannot generalize our results to other adaptation concerns, e.g., security, reliability, and robustness, and target systems. Experimenting with more diverse policies could mitigate this threat. The dependencies between constituent policies and the hybrid planners are affected by various factors, including the utility function and assumptions behind the approaches. Nonetheless, we expect these dependencies to hold for any utility function that is accrued over states of input traces and reflects that timely adaptations are vital to the system's goals. We further mitigate the threat to validity by evaluating on two published application examples that are commonly used exemplars for software engineering and self-adaptive system research community, and are publicly available. We have several past experiences with both from earlier work.

Construct validity. The main threat to construct validity is the correctness of the implementation of the application examples and the approaches. Another threat is the fact that the alternative solution in the comparative study, i.e., $\text{CHP}_{\mathtt{dtr}}$, is also home-developed. We mitigate this threat by using open source and established application examples, the correctness of our implementation of the approaches have been verified by co-authors from collaboration in previous publications.

9.5 SUMMARY

In this chapter, we evaluated the two designs of HypeZon in a comparative study with an alternative solution for hybrid adaptation. We answered the three questions via the experiments:

Q9.1: *how do internal and external designs for meta-self-awareness affect* HypeZon*?* We showed that meta-awareness capabilities that are realized either by the external or internal design are beneficial for hybrid planning. The reason is that they provide extended control flexibility at runtime. The relatively small execution timescale of the hybrid planner in $\text{HZ}_{\mathtt{i}}$ may lead to premature decisions due to insufficient and localized information, and, as a result, the system is likely to demonstrate nervous behavior. However, during volatile operation conditions when the system is experiencing rapid changes, more frequent control calculations are beneficial since the system and its context change in high pace. Therefore, smaller sampling intervals $\mathcal{I}$ for the controller are more effective. Our results suggested that the common practice of increasing the visibility of control loops in the architecture is also beneficial in designing meta-self-aware systems with hybrid planning.

Q9.2: *how does* HypeZon *perform in comparison to a deterministic hybrid planner?* HypeZon variants are more effective in copping with the varying operation conditions, e.g., input traces with more extreme characteristics. However, we also made some observations indicating that more stable operation conditions can be effectively handled by a simpler hybrid planner like $\text{CHP}_{\mathtt{dtr}}$ with deterministic parameter setting. Because

CHP_{dtr} has only a negligible runtime overhead in terms of operation cost, it can be executed more cost-effectively compared to the HypeZon variants.

Q9.3: *what are the effects of hybrid planning on the reward and timeliness of the adaptation?* we showed that employing a hybrid planner, either with deterministic or adjustable parameters, results in improvements of the reward and timeliness of the adaptation.

The findings of this chapter revealed that considering hybrid planning and hybrid self-adaptation as a case for meta-self-awareness has the following advantages and limitation; the external design in HZ_e provides a broader view on the target system and the adaptation process that allows to observe phenomena with a large scope that are not visible at the awareness level. HZ_e supports explicit separation of concerns at the architecture level and allows for reusability, easier maintenance, and independent adjustments of control loop parameters at each level. The internal design of HZ_i constrains the controller in the meta-awareness subject to a localized view of its object. In this design, the meta-awareness logic is dispersed throughout the awareness level. The intertwined realization of the awareness and meta-awareness together with the embedded and dispersed meta-awareness logic makes it challenging to reason about the outcome of the meta-awareness, causing composability and resusability requirements challenging to achieve.

Finally, this chapter provided supporting empirical evidence for the following contributions of the thesis as mentioned in Section 1.4:

C7 A generic scheme for hybrid self-adaptation

C8 Coverage of a wide spectrum of self-adaptation problem space beyond the state-of-the-art

9.5.1 *Fulfillment of Requirements*

The optimality and scalability of HypeZon depend on the characteristics of its constituent policies. In the context of a hybrid solution like HypeZon, such claims are not generally justified as they are inherent to the planning policies and not the hybrid planner. Being integrated in an adaptation engine that supports incremental execution of the adaptations reduces the execution cost and provides for a more scalable solution by definition. However, HypeZon cannot provide optimality or scalability guarantees and fulfillment of $\mathcal{R}1$ and $\mathcal{R}3$ by HypeZon is dependent of its constituent policies. Runtime coordination of multiple off-the-shelf policies in HypeZon allows the scheme to re-use the existing policies in contrast to developing a new solution from scratch, thus reducing the development cost. Therefore, HypeZon fulfills $\mathcal{R}2$. HypeZon relies on utility functions to evaluate individual adaptation policies for runtime coordination. Additionally, the scheme balances the quality-cost trade-off by coordinating the policies that individually can only partially fulfill quality (*Solver*) or cost (*Static*) requirements for different system inputs. We have shown in this chapter that HypeZon can balance the quality and cost of the adaptation by switching between different policies at runtime and tuning its control parameters, thus fulfilling $\mathcal{R}4$.

10 RELATED WORK

This chapter discusses the related work. In this thesis, leveraging MDE principles, graph-based formalism, and utility theory we proposed a solution for incremental architecture-based self-adaptation of software systems, comprising two alternative planning mechanisms. We presented Venus as the main contribution of this thesis whereby a design-time combination of rule- and optimization-based policy formalism is used enabling Venus to address the quality-cost trade-off in engineering self-adaptive software systems. The scheme leverages utility functions to explicitly capture the quality objectives and assign utility values to the matches for the adaptation rules. We complemented the solution by proposing a second planning mechanism, HypeZon, a hybrid solution for self-adaptation that coordinates, at runtime, multiple off-the-shelf adaptation policies that individually may operate either cost-effectively or deliver high-quality plans. Additionally, having defined utility functions for dynamic software architectures, we proposed two different solutions to analytically engineer utility functions and systematically train prediction models to capture utility functions for the adaptation engine.

In Section 10.1, we discuss the landscape of architecture-based self-adaptation regarding their *techniques* for development of self-adaptive software. Next, in Section 10.2, we investigate the planning mechanisms for architecture-based self-adaptation using the requirements for architecture-based self-adaptation introduced in Chapter 1 and discuss if and how these approaches address the quality-cost trade-off during adaptation. The requirements allow us to compare these approaches to Venus and HypeZon. Finally, in Section 10.3, we discuss the alternative solutions to our proposal for training prediction models.

10.1 LANDSCAPE OF TECHNIQUES FOR ARCHITECTURE-BASED SELF-ADAPTATION

In this section, we discuss some of the most practiced techniques for adaptation of software architectures. In an SLR conducted by Weyns et al. [425], next to feedback loops and application domains, software architecture is identified as the main focus of research in the field of software self-adaptation. In general, there exists a large body of work on architecture-based self-adaptation of software systems—see [53, 161, 213, 273, 296, 321, 323]. Architecture-based adaptation of software systems entails modifications to software architecture. Such changes cannot be directly realized by control theory and require additional means for engineering self-adaptive software. In general, approaches for developing self-adaptive software often provide a framework and use some forms of models [301, 367]. In the following section, we review the state-of-the-art in approaches for architecture-based self-adaptation that provide a complete adaptation engine with execution support, i.e., we exclude studies that only focus on the design, e.g., [13, 14, 199, 292].

10.1.1 *Runtime Models and MDE*

Bencomo et al. discovered in their systematic mapping study that a large body of work on models@time, i.e., 131 out of 237 papers, are associated with the field of architecture-based self-adaptation, i.e., the modeled artifacts is the software architecture. We discussed in Section 2.4 that, thanks to their high abstraction level with a global view, models@run.time serve as a natural choice to enhance architecture-based adaptation providing the necessary semantic basis for the systems to achieve self-adaptation. In the following, we discuss solutions based on RTMs that support a full-loop self-adaptation of software architectures.

Model Transformation. Major group of work on architecture-based adaptation with RTMs (see [83, 388, 416]) use model transformation techniques to implement the adaptation process via taking one or more source models and a set of transformation rules to produce one or more target models as outputs [377]. Thus, model transformations address, among others, the problem of keeping multiple models describing the software from different views and at different levels of abstraction consistent to each other. Explicit transformations promise to improve the software quality in MDE [58].

The MAPE-K model for feedback loops [252], the architectural reflection pattern [65], and the three-layer architecture for self-managed systems [273] constitute the most common reference models and architectures for the initial design of self-adaptive software in general. EUREMA [413, 416] by Vogel et al. proposes an MDE approach for engineering self-adaptive software via a *Domain-specific Language* (DSL) to realize both the MAPE-K model and the three-layer reference architecture. The adaptation activities of the feedback loops are specified by RTMs. The framework also supports user-defined implementations for individual adaptation activities. The interplay between the activities and the RTMs are harmonized by certain types of RTMs titled runtime megamodels. A megamodel is a model containing other models and relationships between them while the relationships constitute operations such as model transformations [43]. The framework also offers a modeling language to support the explicit design, execution, and adaptation of feedback loops providing for flexibility and extensibility. An adaptation feedback loop in EUREMA can be executed based on change events as well as state-based information. EUREMA supports runtime evolution of RTMs as well as the adaptation activities. The scope of the work by Vogel et al. reaches beyond the direction of this thesis, e.g., offering a framework that is open for user-defined languages to express RTMs, supporting multiple feedback loops and layered architectures, and evolving models at runtime. However, similar to EUREMA, in this thesis, we adopt the generic concept for executing feedback loops by considering the feedback loop's knowledge as RTMs and the individual adaptation activities as model operations, carried out by model transformation rules via executable Story Diagrams (SDs), that work with the RTM. EUREMA supports incremental principles for execution of the adaptation loop, thus achieving runtime efficiency and scalability. As discussed in Section 4.3, the incremental monitoring in our proposed adaptation engine derives from Vogel et al. [417, 418] that is developed in EUREMA.

Chen et al. [83] propose a combination of requirement-driven self-adaptation and architecture-based self-adaptation to reconfigure component-based architecture models using incremental and generative model transformations for complex architectural adaptations. Three different RTMs, including a goal model, an architectural model, and a design decisions model, relating the requirements and the architectural design, are used. The requirements are captured as goal models where architectural design decisions construct the design decisions models. The planning is driven by a requirement-driven policy. Adaptations are carried out periodically, via execution of scripts for model transformations that change the architectural model to the target configuration. RTMs capture the state of the adaptable software and support the state-based execution of the adaptation feedback loop. However, despite precluding change-based adaptation, the framework achieves a runtime-efficient and scalable execution for the planning but performance of the whole feedback loop is not evaluated. Similar to our solution for architecture-based self-adaptation, Chen et al. automate the detection and execution of architecture-based adaptation via model transformation, however, while we rely on an existing set of adaptation rules whose LHS characterizes the specific issuers that we are searching for in the software architecture, the solution by Chen et al. aims to go beyond the assumption that requirements of self-adaptive systems are well-understood at design time and unchanged at runtime, thus supports architectural adaptations resulting from requirements changes. Although, unanticipated changes are not supported by their solution, the approach offers flexibility in combining primitive architectural changes and restructuring adaptations.

Song et al. [388] propose a model-driven runtime engine for self-adaptation. The solution provides a language for modeling declarative adaptation polices in the form of domain-specific constraints. RTMs are used to maintain the monitored system state through causal connection. Adaptations are carries out following a rule-based, goal-based policy. The adaptation models that are constructed manually and by the domain experts are expressed in the OCL as a general purpose language with higher expressive power that tolerates the conflicts in constraints and captures system policies via a set of constrains. Possible system modifications are automatically extracted from the combination of the RTM and the adaptation model. The solution directly captures system constraints as adaptation policies, using concepts specific to the application domain and applying OCL. The adaptations, implemented as model transformations, are captured as ECA rules via the policy language. A transformation engine takes the system model and OCL constraints from the adaptation model and maps them to first order logic *First Order Logic* (FOL) predicates, interpreting what the constraints imply on the current system. The FOL-based predicates are then fed to a constraint solver to dynamically compute the required modifications on the current system that satisfy the adaptation model constraints. The solver takes into account the constraints, their priorities, and the cost of system modifications. Finally, the models@runtime engine propagates the changes to the real system. The proposed solution does not have a control loop-based realization in its design. Adaptation rules, in contrary to our work, are automatically generated from descriptions of adaptation options and domain-specific constraints. Search-based capabilities are used to derive adaptation actions that generate suitable target configurations. The approach does not explicitly express the feedback loop and as confirmed by the authors, does not exhibit runtime-efficiency and scalability for large, complex systems. In general, the runtime efficiency is only shown for small systems/models without investigating the scalability.

Variability Modeling. Besides model transformation, techniques such as variability modeling have been used in combination with RTMs to perform model-driven self-adaptation where the planning of adaptations is determined by change models, henceforth variability models, that define the variability of the software.

DiVA [311–314] by Morin et al. employs RTMs with variability modeling technique to support architecture-based self-adaptation. DiVA uses an architectural RTM of the adaptable software, a model of the context, a feature model for the variability, and ECA rules to perform adaptations. Given a context model, these rules describe which feature should be used on the architectural model. The four models required by DiVA are expressed in a DSL. The framework implicitly captures the feedback loop in its implementation. An adaptation in DiVA is triggered via querying the sensor inputs which support complex event processing. Consequently, DiVA supports various types of triggers for adaptation. DiVA partially supports event-based adaptation, i.e., change events are processed to update the architectural model while the analysis, planning, and execution activities work in a state-based manner. During an adaptation, the analysis and planning are reported to be efficient for small models; the execute activity performs an expensive state-based comparison of two architectural models for modification which will likely not scale. In general, the runtime efficiency is only shown for such small systems/models without investigating the scalability. Thus, while supporting the event- and state-based principles, DiVA does not combine both to achieve an overall efficient solution.

10.1.2 *Architecture Models Defined with ADLs*

Apart from the causally connected RTMs of software architecture that require MDE techniques for implementation, many architecture-based approaches represent the architecture in other form of models using ADLs to describe architectural models. These models capture the architecture in different abstract representations—see [37, 163, 169, 273, 321, 322].

Driven by the external control for self-adaptation, the Rainbow framework [90, 91, 93, 163] divides the adaptable software into an architecture layer and a system layer with the managed resources. Rainbow sets its main goal on cost-effective development of the adaptable software. The framework is based on the Acme ADL to express the architecture and the context. Rules and constraints, as part of the architecture model, are specified in a first-order predicate logic provided by Acme. The architecture layer in Rainbow has different components, defining adaptation strategies and utility preferences in Stitch [91]. A translation infrastructure translates and controls the deployment of the adaptation plans at the system layer. The framework captures the RTMs, i.e., the architectural model, in Acme. Hence, Rainbow does not employ RTMs that follow MDE principles. For supporting runtime adaptation with Rainbow, an architectural style notion of the system is used and extended with adaptation operators and strategies. Rainbow supports structural and parameter adaptation. The MAPE-K loop is implicit in the design. Rainbow is a purely rule-based approach where rules define constraints for the architectural model, as well as strategies, tactics, and operators that realize the adaptation. The framework offers customization points in the adaptation engine whereby engineers can reuse the framework and customize it for a specific system. The adaptation triggers are not explicitly defined, rather are left as customization points which have to be realized within the implementation of the sensors and adapters. The feedback loop is executed in a state-based manner. Before an adaptation, all the constraints are evaluated on the architectural model when the model changes but change events are not processed to drive the constraint checking and model evaluation. Hence, Rainbow does not support an event-based execution of the adaptation loop. The Framework does not comprehensively evaluate the performance and scalability and reports efficient execution for rather small adaptable systems.

Genie [35, 37] is a middleware that uses DSLs for the construction of the models associated with both the structural and the environment variability. The structural variability models are used to generate components and component configurations. Genie promotes the model-driven development of middleware-based self-adaptive software where software artifacts are generated using models and generative techniques. Developers specify configurations of the adaptable software and adaptations in terms of conditional transitions between these configurations. Using these models, Genie generates configuration files and ECA rules for the software running on the middleware. Genie implicitly realizes the adaptation loop. Context changes are captured by change events that trigger the ECA rules. The rules are captured as transition models (expressed in a language specific to the middleware and without using any RTM). Genie supports event-based adaptation but does not offer a platform, e.g., a reflection model or an RTM, to capture state-based information. The approach is not concerned with delivering adaptations efficiently or at a certain quality, rather, offering an ADL with the focus on providing the architectural principles to support the generation and operation of component-based adaptive systems. The authors of Genie do not report about the runtime efficiency of the feedback loop.

MADAM [146, 168] is a middleware following an architecture-centric approach where employing architecture models at runtime allow the generic middleware components to reason about and control the adaptation. The framework realizes parameter and structural adaptation of mobile applications. The architectural model of the adaptable software is used as a code-based and therefore, as an implicit RTM by the middleware. It offers a UML-based architectural language to characterize system components and their variability regarding quality of service properties, system context, and resources. At runtime, MADAM monitors the system context and a utility function steers the adaptation engine towards finding the most promising configuration for the current context—we discuss the planning scheme of MADAM in more details later in this chapter. MADAM does not explicitly capture a feedback loop for adaptation in its implementation. Change events that notify about the context changes trigger adaptations in MADAM. The state of the adaptable software is not used for reasoning except for obtaining a reconfiguration script by comparing the current with the target architecture of the adaptable software. MADAM achieves a runtime-efficient execution for small-sized applications with a limited variability. The authors acknowledge scalability problems.

The MUSIC framework [359, 360] builds on MADAM approach for the implementation of context-awareness and self-adaptation, where context management and adaptation logic are delegated to generic middleware. MUSIC is an extension of MADAM framework inheriting its development methodology and realization of self-adaptation. The major extensions in MUSIC have been done to the middleware to cope with ubiquitous applications. Therefore, we refer to the discussion of MADAM.

The Framework by Alférez et al. [3] is another solution for runtime architecture-based adaptation where RTMs are defined with ADLs. The framework uses forward chaining from AI—a bottom-up computational model that starting from a set of know facts, applies a logical process to infer unknown facts whose premises match the known facts and moves forward using determined conditions and rules to find a solution [see 361]—together with RTMs to plan the adaptation. The framework requires several RTMs: a requirement model that captures the system goals, an architectural model, a variability model, a context model, and a set of tactic models each capturing an adaptation strategy as features for the variability model. The context model includes conditions to be analyzed during monitoring to detect the need for adaptation. The framework implicitly realizes a feedback loop that monitors the system and its context, evaluates the context conditions, and using the variability model, reconfigures the architecture accordingly. Forward chaining is employed to find suitable adaptation tactics, woven to the variability models, that address unknown contextual situations. The adapted variability model then is used to steer the adaptation. The adaptations are performed periodically and only based on state information excluding event-based processing of the changes. The framework is not evaluated for the runtime efficiency of the feedback loop and scalability.

10.1.3 *Discussion*

As revealed by our review of some of the most-practiced techniques to develop an architecture-based solution for software self-adaptation, the two major groups of techniques, regarding development and adaptation of software architecture models, entail MDE-based principles and ADLs in combination with some form of RTMs. Our work is well in line with the state-of-the-art as we employ an architecture-based approach to self-adaptive software that, via the transformation of architectural RTMs, consistently evolve the adaptable software and the architectural model of the software using MDE Principles. Our proposed solution supports automated parameter and structural adaptation of software systems controlled by feedback loops via explicitly capturing the control loop in its design and implementation, conforming to the MAPE-K reference model. While the technical contributions of this thesis concern the Analyze and Plan activities of the MAPE-K feedback loop, leveraging the incremental Monitor module from Vogel et al. [418], the contributions are embodied in a MAPE-based adaptation engine that is executed incrementally. This way, the solution offers supporting mechanisms for efficient execution of the adaptation loop, thus advancing the baseline for architecture-based self-adaptation.

The survey on self-adaptive software by Salehie and Tahvildari [367] suggest that development of self-adaptive software systems is often supported by using some form of models. In this thesis, we use an architectural RTM of the adaptable software as the knowledge element of the MAPE-K loop. Additionally, we use a goal model of the adaptable software to extract an objective function that steers the adaptation. In Section 3.3, we also provide a solution where the goal model of the system is not available, hence, we learn system preferences via observations. The results of our investigation in this section confirmed the observation of Salehie and Tahvildari [367]: state-of-the-art employ models to represent the knowledge part of a feedback loop, i.e., they use models to describe the adaptable software as well as working data for the feedback loop such as adaptation rules, e.g., Rainbow and Genie, and variability models, e.g., DivA and Alférez et al.

In line with the discovery made in a 2019 survey on models@run.time by Bencomo et al. [36], we observe that among the MDE development approaches to self-adaptive software, including this thesis, model transformations are heavily explored as a natural prerequisite for models@run.time in order to connect the RTMs and the system. Examples are Chen et al., Song et al., and EUREMA. Another observation was that the state-of-the-art, similar to this thesis, often support both structural and parameter adaptation.

The execution of a feedback loops is either event-based or state-based. The former is based on the change events that notify about changes in the adaptable software or context, while the latter requires periodical evaluation of the (reflection) model of the adaptable software that captures the state to detect adaptation triggers. As discussed earlier in this section, solutions that support event-based processing of software and context change, i.e., EUREMA, Genie, and MADAM/MUSIC, are more likely to show runtime efficiency. In case of MADAM/MUSIC, scalability issues have been reported by the authors. DiVA only partially supports change events of the RTMs while the analysis, planning, and execution activities work in a state-based manner, thus suffer from scalability challenges—see [311]. State-based execution of adaptation activities entails non-incremental processing of the architecture models, i.e., with every execution the whole state captured in the architectural model should be processed. This renders the execution of the adaptation loop time-intensive and inefficient. Examples are the frameworks by Song et al., Chen et al., Alférez et al., and Rainbow. Only a few approaches such as EUREMA and DiVA combine both event- and state-based principles. In the context of DiVA, as discussed above, while both event-based and state-based principles are supported, the framework does not use change models for incremental adaptation, thus the combination does not improve the runtime efficiency.

In this thesis, similar to EUREMA, we exploit change events, leveraging event-based principles for an incremental execution of the adaptation loop. We also support state-based execution of the adaptation loop as it holds a global view on the adaptable software through maintaining RTMs that represent the state of the adaptable software. While state-based realization of the adaptable software supports decisions with global impacts, event-based processing of the changes enables incremental execution of the feedback loop. We have investigated the runtime efficiency and scalability of both adaptation solutions, i.e., Venus and HypeZon via qualitative discussions and quantitative evaluations—see Section 7.3.1, Section 7.4.1, Section 7.5.2, and Section 9.3.3.

Finally, as discussed above, the runtime efficiency of executing the adaptation loop is often either not investigated and, thus remains unknown, e.g., Genie and Alférez et al., or efficiency claims are only partially justified, i.e., either investigating only the performance of an individual adaptation activity but not of the whole feedback loop, e.g., Chen et al, or for a subset of the activities, e.g., DiVA. Song et al., Rainbow, and MADAM/MUSIC report efficient executions although they often do not investigate the scalability. EUREMA reports on both runtime efficiency of the adaptation loop and scalability.

10.2 PLANNING MECHANISMS FOR ARCHITECTURE-BASED SELF-ADAPTATION

In Section 10.1, we discussed the landscape of solutions for architecture-based self-adaptation regarding their *techniques* for development of the self-adaptive software. This section discusses the related work in *planning* mechanisms for architecture-based self-adaptation by using the requirements introduced in Chapter 1 and summarized in Table 1.1. The requirements allow us to compare these approaches to Venus and HypeZon. Among the large body of existing work, we chose approaches that propose a *complete* solution for self-adaptation, i.e., addressing the whole feedback loop and not only a planning mechanism, and that are concerned with quality, cost, or their trade-offs during planning. We organize the section as the following: we discuss search and optimization-based planning solutions in Section 10.2.1. Learning-based solutions are pretested in Section 10.2.2. In Section 10.2.3 we discuss rule-based solutions for architecture-based self-adaptation and finally, Section 10.2.4 discuses hybrid solutions.

10.2.1 *Search and Optimization-based Planning*

On one end of the spectrum there are search and optimization-based approaches for runtime reasoning. Potential adaptation decisions are discovered and evaluated, at runtime or at design time, often by means of exhaustive search in software adaptation space or by optimizing a form of an objective function. In this context, employing utility functions and utility-driven decision-making schemes have been extensively investigated.
Greedy Algorithm MADAM/MUSIC [146, 168, 359, 360] are adaptive middleware for component-based applications that plan architectural adaptation by exploiting quality properties of alternative implementations of components. MUSIC [359, 360] is the successor of MADAM that extends it to ubiquitous environments and uses the same self-adaptation mechanism as MADAM—see 10.1. MADAM is focused on adaptation of one node while MUSIC takes the use of service into account that are deployed on other nodes. A *Quality of Service* (QoS)-aware RTM provides the knowledge for planning activity that, pursuing a greedy algorithm, maximizes the utility of the architecture. A greedy algorithm follows the problem-solving heuristic of making the locally optimal choice at each step. Using properties and property predictor functions of alternative components, each reconfiguration is planned and then evaluated for the current execution context by a utility function. MADAM/MUSIC represent different configurations via different component architectures which are discriminated based on the properties related to the context elements. However, the properties of each configuration regarding the context are constant, predefined values. A utility function is applied to map the provided properties of the context to scholar values representing user needs and static priorities. Strategy selection in MADAM/MUSIC is conducted dynamically regarding the given context, but the expected utility is based on design time estimates. It is only the different user needs that assign different weights to utilities and may result in different solutions for the same context. Adapting a system via the MADAM/MUSIC framework peruses a greed optimization to maximize a utility function that captures system quality attributes ($\mathcal{R}1$), however, the solution searches the possible adaption space at runtime to discover the plan yielding the highest utility, hence, is not cost-effective ($\mathcal{R}2$). The scalability of the framework is not evaluated but authors point out that for large adaptation spaces they expect scalability issues ($\mathcal{R}3$). Quality-cost trade-offs are captured by utility functions ($\mathcal{R}4$). The framework support dynamic architectures ($\mathcal{R}5$) but requires developers to provide utility functions, typically expressed as a weighted sum of dimensional utility functions where the weights express user preferences ($\mathcal{R}6$).
Model-checking *Markov Decision Process* (MDP) [345] and *Partially Observable Markov Decision Processess* (POMDPs) [239] describe discrete-time stochastic control processes and are commonly used to model and optimize decision-making in stochastic environments. Probabilistic model-checking has been used to solve complex MDP and POMDP optimization problems at runtime—see [70, 71, 394, 395]. Probabilistic model-checking

refers to a set of techniques that enable quantitative analysis and policy synthesis in systems with probabilistic behavior. The technique guarantees to achieve optimal expected probabilities and rewards [see 277]), which is mapped to maximizing utility. It takes as input a formal specification of the stochastic system, which is internally translated into an MDP, and solves it. The time complexity of model-checking typically results in solutions that do not scale for large configuration spaces as the MDP/POMDP model has to be constructed every time for decision-making to incorporate the latest predictions of the environment behavior. Therefore, rendering these solutions infeasible for application in adaptive systems with large and complex state-space or requiring instantaneous adaptation decisions. Although various optimization algorithms have been suggested to improve the planning time for MDP [266, 308] and POMDP [339] planning, planning delays in probabilistic domains remain a serious challenge.

Approaches such as work by Moreno et al. [308] offer improvements to runtime probabilistic model-checking whereby most of the computation is performed off-line, thus reducing execution time. In the solution proposed by Moreno et al. an MDP model of the adaptable system is constructed off-line where feasibility of the adaptation decisions for possible system states are assessed. During system execution, the planning activity uses stochastic dynamic programming to solve the MDP and select the adaptation action that maximizes the system utility ($\mathcal{R}1$). Computations are performed as much as possible off-line to reduce the planning efforts online ($\mathcal{R}2$). The efficiency of the planning is shown to be improved in comparison to using probabilistic model-checking at runtime, while providing comparable results in terms of adaptation utility. However, the efficiency of the whole feedabck loop and scalability of the adaptation is not evaluated ($\mathcal{R}3$). The solution explicitly considers quality-cost trade-offs ($\mathcal{R}4$). Assuming a static system model, at runtime, only the behavior of the environment is dynamic, thus dynamic architectures are not supported ($\mathcal{R}5$). The decision-maker depends on an MDP model of the system and a user-defined goal model to operate ($\mathcal{R}6$).

Franco et al. [149] address the runtime disruption of non-functional goals by predicting their expected values for each adaptation strategy. The quantitative prediction is based on a stochastic model translated from a model of the software architecture. At runtime, before each adaptation, the Analyze activity, using a model-checking tool, predicts the impact of each possible adaptation action (strategy) on a copy of the snapshot model of the system architecture. During planning, the adaptation action with highest predicted impact on system quality objectives is chosen. The solution aims to maximize system quality objectives ($\mathcal{R}1$) but executes a time-intensive model-checking process before each adaptation at runtime to discover the optimal adaptation plan, thus is not cost-effective ($\mathcal{R}2$). The efficiency of the adaptation is evaluated for one adaptation scenario but scalability claims are only supported by enforcing time constraints on the model-checking process during the analysis and potentially affecting the quality of the adaptation plans ($\mathcal{R}3$). The quality-cost trade-off is explicitly captured via utility functions ($\mathcal{R}4$). Dynamic architectures are not supported ($\mathcal{R}5$) and the approach relies on prior knowledege to construct a utility function that is employed by the model-checker at runtime ($\mathcal{R}6$).

The PLASMA approach [398] addresses the planning problem via model-checking. It proposes a three-layer architecture for plan-based adaptation, where the goal is the generation of plans for adapting and executing software applications. The adaptation loop and plan generation loop are executed separately. During system execution, the planning layer, using a developer-defined domain model and the initial and goal states of the application, performs model-checking to device a plan for reaching the goal state. PLASMA does not rely on design-time plans for adaptation, thus supports dynamic and unforeseen runtime conditions and can generate effective plans at runtime. However, the solution does not offer any guarantees for system quality objective satisfaction ($\mathcal{R}1$) and is not cost-effective due its time-intensive planning-as-model-checking mechanism ($\mathcal{R}2$). The performance and scalability of the approach is not evaluated ($\mathcal{R}3$) and quality-cost trade-offs are not considered ($\mathcal{R}4$). PLASMA tends to offer a solution for online planning and supports dynamic software architectures ($\mathcal{R}5$). The approach requires user-defined target states, captured by the goal model, thus cannot operate without prior knowledge about the application domain and user preferences ($\mathcal{R}6$).

While model-checking and probabilistic planning enable exploration of rich adaptation spaces, they render attainments of optimal solutions and other guarantees, e.g., constraint satisfaction, computationally costly, resulting in long planning times. MOSAICO [72] improves the scalability of such approaches by using model-checking mechanism in an off-line manner for plan synthesis. The solution targets the trade-off between quality and computation cost by discretizing the system and environment states and off-line synthesis of adaptation plans for the different discretization points [72]. MOSAICO requires user-defined utility models to estimate the impact of the adaptation decisions on system quality objectives. A parametric model of the system that captures properties of interest, e.g., quality attributes, is used to synthesis a plan with highest expected utility at design time. The parametric model is specified based on the architecture model, utility model, and estimated impact of the tactics. At runtime, to generate the best decision for current state, the set of states in the discrete abstraction that have been used to synthesize the repertoire of adaptation plans is searched. The current state of the system may or may not belong to the set of states in the discrete abstraction. In the latter case, the plan selected for execution will be the one synthesized for the closest state in the abstraction. Analogously to off-line planning solutions, MOSAICO cannot provide optimality guarantees in terms of system quality objective satisfaction as plans are generated at design time ($\mathcal{R}1$). The solution is cost-effective ($\mathcal{R}2$). The performance and scalability of the whole adaptation loop and the runtime planning is not evaluated ($\mathcal{R}3$). Quality-cost trade-offs are explicitly captured ($\mathcal{R}4$). Dynamic architectures are not supported as the off-line planner uses the design-time system architecture to generate the state space and runtime evolution of the architecture is not considered by the synthesized plans ($\mathcal{R}5$). MOSAICO requires user-defined utility profiles that capture runtime preferences to steer the synthesis of plans ($\mathcal{R}6$).

Heuristics and Meta-Heuristics Search-based meta-heuristics such as hill-climbing [304] conduct a local neighborhood search in the solution space for self-adaptation whereby relying on a given starting point, a better solution in the neighborhood is searched. SASSY [304] uses a utility function to capture the quality-cost trade-off. An adaptation loop is triggered upon a drop in the overall system utility. For each adaptation, starting from the current architecture, SASSY performs a hill-climbing combinatorial search in the space of possible architectures. The architecture with the highest utility becomes the new visited point in the search space. The search stops if no architecture in the neighborhood increases the utility. The solution cannot provide optimality guarantees regarding system objectives due to the inherent local-optima problem in search-based solutions ($\mathcal{R}1$). Starting from the current state, SASSY searches the complete neighbor configurations in the fragment of the adaptation space that are reachable from the current configuration, thus is not cost-effective in terms of planning time ($\mathcal{R}2$).

The approach is evaluated for scalability and adaptation performance ($\mathcal{R}3$). Moreover, an evaluation of the overall utility of the system utility is required to detect the need for adaptation, thus event-based processing of the adaptation triggers are precluded rendering scalability challenging. The quality-cost trade-offs are explicitly captured via utility functions ($\mathcal{R}4$). Dynamic architectures are supported ($\mathcal{R}5$). SASSY demands user defined objective function and builds on prior domain knowledge for operation ($\mathcal{R}6$).

Genetic Algorithms *Genetic Algorithms* (GAs) [210] are shown to be effective in discovering near-optimal adaptation plans for large and complex adaptation spaces [154, 209, 331, 352, 353]. Hermes [352] is an evolutionary computation-based approach that searches the adaptation space to find the best path to a desired configuration. Hermes uses genetic programming to generate executable reconfiguration instructions specifying structural and behavioral changes to the adaptable system. The approach requires a set of instructions characterizing the adaptation paths that are extracted from a user-defined component-dependency map between different system configurations. At runtime, Hermes uses evolutionary techniques to gradually transform and possibly improve an adaptation path by adding, removing, replacing, and reordering reconfiguration instructions to better balance the competing objectives, while ensuring a safe transition to a desired target configuration. The approach does not implement a complete adaptation loop. Assuming monitoring data, starting configurations, and target configuration as given, Hermes is assigned with the task of finding the safest path to the target configuration with the primary objective of maximizing reconfiguration performance and reliability ($\mathcal{R}1$) while minimizing reconfiguration costs ($\mathcal{R}2$). However, the optimality guarantees cannot be provided. The performance of the path-planning by Hermes is evaluated for a industry-size case study but the efficiency and scalability of an adaptation loop is not evaluated ($\mathcal{R}3$). Hermes explicitly captures the quality-cost trade-off via defining fitness sub-functions ($\mathcal{R}4$). The solution requires design-time knowledge of possible system configurations prior to execution, thus does not supports dynamic architectures ($\mathcal{R}5$) while heavily relying on user-defined component-dependency maps and fitness functions to generate reconfiguration paths ($\mathcal{R}6$).

In general, GA-based solutions are not concerned with generating adaptation plans at runtime, but with selecting from an existing set of plans and computing the most feasible paths to the plan. However, these solutions lack flexibility to cater to the needs of dynamic architectures and dynamic environments, thus cannot offer optimality guarantees regarding the final configuration as the plans are determined based on design-time knowledge.

To improve runtime efficiency, techniques such as caching, pre-computation, and near-optimality have been applied [171] and computations are performed as much as possible off-line to reduce the planning efforts online [308]. Moreover, Moreno et al. [309] propose a method for combinatorial optimization based on cross-entropy and an any-time algorithm with random sampling from the solution space. Anytime planning algorithms, based on the idea of incremental planning, are optimizing, e.g., value iteration algorithm for MDP planning whereby the planning process can be interrupted at any time to get a sub-optimal plan [440]. Longer planning times lead to better plans. Such solutions considerably reduce the computation time, however, they are not guaranteed to find an optimal adaptation plan.

Search and optimization-based approaches pursue a search-based optimization in the potentially large adaptation space, which typically do not scale well for complex systems with large configuration spaces. Such approaches can manage to find the optimal configuration but there is no guarantee to reach the result within a reasonable time. Executing an optimization algorithm for each adaptation decision at runtime causes a large overhead degrading the performance. Tichy et al. [403] suggest reducing the search space to speed up adaptation and avoid long delays. In contrast, the adaptation solutions proposed in this thesis compute the utility for each possible adaptation option, incrementally, and at runtime, taking into account the actual issues and their contexts, i.e., runtime knowledge influencing the utility.

10.2.2 *Learning-based Planning*

Another technique for planning and decision-making with growing popularity is to machine learning. Gheibi et al. [182] in their 2021 SLR on application of machine learning in self-adaptive systems identified a total of 15 studies on architecture-based self-adaptation that address the quality-cost trade-off during adaptation. Among the 15 papers, we identified four studies that use machine learning during analysis and planning and are concerned with learning policies/rules which we consider relevant to this work that we briefly discuss in the following.

Kim et al. [255] use RL for online planning in architecture-based self-management that enables a software system to change and improve its plan. State and actions are defined through goal- and scenario-based discovery processes where goals capture the system objectives and scenarios, realized in form of condition-action, represent a sequence of events to achieve a specific goal. The state space is defined based on the condition of the scenarios and a symbolic notation of the current architecture that is denoted as a vector of components and connectors. Adaptations are defined based on the action part of the scenarios and describe architectural changes such as adding, removing, replacing components, and reconfiguring the architectural topology. A fitness function, derived from system goal model, represents the reward. Kim et al. apply Q-learning method [274] to discover the action that maximizes system reward. Using an $\epsilon-$greedy selection strategy, Q-learning avoids local optima, thus is able to find the optimal action, i.e., the most profiting action regarding system quality objectives. However, the online planner my operate sub-optimally until the learner converges ($\mathcal{R}1$). The approach employs online planning where actions are chosen for execution through the process of exploitation and exploration, which, for large state spaces, has a time-intensive convergence. Hence, in the context of highly configurable and dynamic software architectures, the online learning-based planning is not cost-effective ($\mathcal{R}2$). The runtime efficiency and scalability of the approach is not evaluated ($\mathcal{R}3$), nonetheless, for large state-spaces, online learning solutions by definition suffer from scalability issues [see 357]. Quality-cost trade-off policies are not captured in the fitness function ($\mathcal{R}4$). The online planner supports adaptation of dynamic architectures ($\mathcal{R}5$). Finally, the approach uses Q-learning that is a modeless technique and can learn directly from raw experience without a model. Therefore, the approach may operate without detailed knowledge of user preferences and observe system reward as the ground truth ($\mathcal{R}6$).

FUSION [126, 129] uses supervised learning to continuously learn the impact of software features on system goals, thus improves the decision-making. System goals are defined as utility functions by developers. Additionally, the framework requires a feature model defining the variability and an architectural model of the adaptable software. Fusion solves the optimization problem of finding the optimal set of features that maximizes the utility. The approach aims to keep the system response time low and the reliability high under changing workload and the occurrence of unexpected events. The feedback loop consists of two cycles, one for learning and one for adaptation. The latter uses the learned knowledge to decide which features should be (de)activated if a goal is violated. Changes of the feature selection are propagated to the architectural model and to the system via the user-defined transformations. When runtime conditions render the learned knowledege in FUSION out of context, the framework performs sub-optimally until the learning converges ($\mathcal{R}1$). The framework is not cost-effective ($\mathcal{R}2$). FUSION uses events and the state captured in RTMs to drive the execution. The framework then uses features and inter-feature relationships to reduce the configuration space of a sizable system rendering the runtime analysis and learning feasible, thus achieves a runtime-efficient execution for the planning activity and for the learning. The performance of the whole feedback loop is not evaluated. The framework achieves a partially efficient execution and investigate only the performance of the planning activity but not of the whole feedback loop ($\mathcal{R}3$). Quality-cost trade-off policies are explicitly captured ($\mathcal{R}4$). Owing to online learning in FUSION, dynamic architectures are supported ($\mathcal{R}5$), Analogously to any learning-based approach, provided sufficient unbiased and representative data about the system execution under different feature configuration, FUSION infers an accurate model of the system behavior from data and does does not required manually constructed utility functions ($\mathcal{R}6$).

IDES [194] extends the rule-based adaptation scheme in Rainbow [91] whereby cost-benefit assessment of adaptation strategies evolve at runtime. The feedback loop realized by IDES consists of two cycles, one for evolving utility preferences and one for adaptation. The adaptation loop is supported by a reinforcement learner that dynamically updates adaptation policies based on the instant reward. The adaptation loop uses the updated knowledge to adapt the system. IDES improves the rule-based adaptation scheme of Rainbow by updating policy preferences at runtime and based on the received reward, therefore, adjusts design-time utility preferences based on runtime conditions. However, the policy evolution does not consider any exploration, thus has high probability to fall in a local-optima ($\mathcal{R}1$). The framework is cost-effective ($\mathcal{R}2$) and explicitly captures quality-cost trade-offs ($\mathcal{R}4$). The efficiency of the adaptation has not been evaluated in a conclusive study and scalability is not reported ($\mathcal{R}3$). IDES does not offer any improvements in the design-time policy definitions in terms of dynamic system attributes, thus does not support dynamic system architectures ($\mathcal{R}5$). The framework operates based on user-defined preferences and utility functions ($\mathcal{R}6$).

Zhao et al. [437] use an off-line RL-based phase to learn best performing rules and optimal values for system quality attributes from a set of given goals. The results constitute a case-base that is then employed at runtime. The plans are generated via *Case-based Reasoning* (CBR), i.e., utilizing the knowledge of past cases to solve new cases—see [1, 283]. Before each adaptation, following the general CBR cycle and an ϵ-greedy scheme, the best performing rules that maximize system reward in terms of quality attribute satisfaction are chosen ($\mathcal{R}1$). Leveraging an off-line training phase, the approach precludes a time-intensive training at runtime ($\mathcal{R}2$). The performance of the whole adaptation loop and the planning activity is not evaluated and scalability is not discussed ($\mathcal{R}3$). The quality-cost trade-offs are not explicitly captured ($\mathcal{R}4$). The approach supports new system goals while only amenable to static quality attributes defining system states, thus dynamic architectures are not supported ($\mathcal{R}5$). The solution supports black-box systems where system and user preferences can be learned during training ($\mathcal{R}6$).

In general, online learning-based approaches suffer from a slow learning curve causing them to perform sub-optimally in terms of adaptation decision qualities until the learning converges. In solutions where an off-line training mitigates the cold start at runtime, changes that render the current knowledge insufficient will be addressed only, at best sub-optimally, until the learners converge. Therefore, learning-based solutions are often challenged by scalability issues and sub-optimal adaptations.

10.2.3 *Rule-based Planning*

Rule-based approaches are recognized to be efficient and stable in predictable domains and support early validation [144]. They provide a quick recovery from a goal violation. However, they often result in sub-optimal adaptation decisions as they do not handle situations that have not been foreseen at design time.

Rainbow framework [90, 91, 93, 163] applies utility theory in combination with a stochastic model for the reasoning process. Rainbow addresses the adaptation issues based on solutions to similar problems from the past. When adaptation is needed, Rainbow chooses an adaptation strategy from a predefined repertoire, created at design time by domain experts based on their past troubleshooting experience. Rainbow considers the success rate of rules in the past to rank them and to eventually make a decision. For each observed configuration, there is an specific adaptation strategy (rule) assigned at design time. Therefore, the decision-making in Rainbow is insensitive to the runtime context as the framework maps rules (strategies) to runtime states solely based on design-time estimates of the applicability and desirability of the strategy for the given context. In this thesis, additionally to the dynamic properties of the rules in terms of utility impact and execution costs, the actual failures and their contexts, i.e., the affected components, are considered to make an adaptation decision. We have discussed the Rainbow framework in full length in Section 10.1, therefore, we briefly discuss how the framework satisfied the requirements in the following. Rainbow is limited by design time estimates of the potential impacts of the rules on utility of the system, thus does not guarantee to satisfy system quality objectives at runtime ($\mathcal{R}1$). The framework is developed and executed in a cost-effective manner ($\mathcal{R}2$), however, has not been investigated for scalability ($\mathcal{R}3$). Rainbow explicitly captures quality-cost trade-off policies ($\mathcal{R}4$). Dynamic architectures are not supported in Rainbow ($\mathcal{R}5$) and a user-defined utility profile, capturing system preferences, is required to steer the adaptation ($\mathcal{R}6$).

10.2.4 *Hybrid Planning*

Hybrid adaptation mechanisms combine, at various levels, concepts from two or more different adaptation policies, e.g., rule-based, goal-based, or utility based to achieve hybrid policies that collectively exhibit the features of the constituents. Trollmann et al. [405] provide a framework for classification of different combination types of adaptation policies with examples from the literature. In the following, we discuss three of the most relevant hybrid solutions for architecture-based self-adaptation. In the context of hybrid solutions, we introduce $\mathcal{R}7$ as an additional desired feature for the hybrid planner to be added to the set of six requirements for architecture-based self-adaptation in Table 1.1; $\mathcal{R}7$ indicates whether a hybrid planning solution is generic, thus can combine arbitrary adaptation polices for planning. A generic hybrid planner is not custom-

Table 10.1: Architecture-based self-adaptation requirements coverage by relate work. (●) indicates full, (◐) indicates partial, (○) indicates no, and (?) indicates unknown coverage.

Category	**Approach**	Desirable quality objectives satisfaction	Cost-effective	Scalable	Explicit quality-cost trade-off policies	Dynamic architectures support	Functional without design-time knowledge on preferences	Generic (Hybrid)
		$\mathcal{R}1$	$\mathcal{R}2$	$\mathcal{R}3$	$\mathcal{R}4$	$\mathcal{R}5$	$\mathcal{R}6$	$\mathcal{R}7$
Search & optimization	MADAM[146, 168]	●	○	○	●	●	○	
	MUSIC[359, 360]	●	○	○	●	●	○	
	Moreno et al. [308]	●	●	?	●	○	○	
	Franco et al. [149]	●	○	○	●	○	○	
	PLASMA[398]	○	○	?	○	●	○	
	MOSAICO[72]	○	●	?	●	○	○	
	SASSY[304]	○	○	?	●	●	○	NA
	Hermes[352]	◐	●	?	●	○	○	
Learning-based	Kim et al. [255]	◐	○	?	○	●	●	
	FUSION[126, 129]	◐	○	◐	●	●	●	
	IDES[194]	○	●	?	●	○	○	
	Zhao et al. [437]	●	●	?	○	○	●	
Rule-based	Rainbow[90, 91, 93, 163]	○	●	?	●	○	○	
Hybrid	Pandey[326, 327, 329]	◐	●	?	●	○	◐	◐
	Caldas et al. [67, 132]	●	○	○	○	○	○	○
	Qian et al. [348]	●	○	?	○	○	○	○
	VENUS	●	●	●	●	●	●	NA
	HYPEZON	◐	●	◐	●	●	●	●

designed to combine specific policies and considers the policies as black-box. Therefore, a generic hybrid planner is applicable for combining (coordinating) a variety of adaptation policies.

A coordinating hybrid planning framework is proposed by Pandey et al. [326, 327, 329]. Among the hybrid solutions for self-adaptation investigated in this section, the approach by Pandey et al. is closest to HypeZon, thus we discuss in more details how we distinguish HypeZon from Pandey et al. The approach operates an MDP-based (learning-based in [326]) planner in the background while a deterministic policy adapts the system. The choice between different policies is based on time and quality constraints. Adaptation triggers are detected via periodical evaluation of the system utility. The approach supports concurrent executions of adaptation policies where an optimization-based planner is continuously operating in the background. During adaptation, the hybrid planner checks in fixed intervals if the optimal policy has a plan. If the optimal plan is not ready, the deterministic policy adapts the system. The employed off-the-shelf policies concurrently generate plans based on the current state. The reactive, deterministic policy delivers timely plans based on a set of pre-defined tactics while the optimization-based policy searches for a better quality plan followed by a time-intensive optimization process. Upon delivery, the hybrid planner executes the optimal plan and improves the quality of the adaptation.

While the idea of making a trade-off between timeliness and optimality is explored by both Pandey et al. and HypeZon, we distinguish our work based on the following grounds: in the approach by Pandey et al., due to concurrent execution of multiple policies, the compatibility of the newly generated plans with the planning problem is formally verified before execution. In order to guarantee the compatibility between a plan and the planning problem, HypeZon only executes one policy at a time and avoids concurrent executions of multiple policies. As a result, a planning problem that is assigned to a policy remains unchanged during the planning time. This way, once the plan is ready, HypeZon does not have to check if the plan is still applicable to the current planning problem. This feature in HypeZon avoids the runtime overhead that is caused by compatibility analysis between the planners. However, concurrent executions of planners may reduce the time that hybrid planner has to wait until a plan is ready as it allows for executing proactive policies in the background.

Pandey et al. use static thresholds on quality attributes of interest, e.g., response time or performance, as triggers to switch between the adaptation polices. The hybrid planner by Pandey et al. is proposed as an algorithm, thus as opposed to HypeZon, does not require any specific architectural modifications to the conventional MAPE-K loop and is realized in the planning phase of the adaptation loop—see Section 5.3.3. Moreover, Pandey et al. do not support runtime adjustments of the control parameters and consider look-ahead horizon and planning horizon with predefined and fixed sizes. Executions horizon is not considered.

HypeZon is event-driven by reacting to the change events so that it avoids a continuous utility evaluation to detect adaptation needs. A complete evaluation of the system utility can be costly if the architecture is large. The adaptation loop realizing HypeZon seeks timeliness by using techniques for the incremental detection of adaptation issues and pattern-based computation of utility changes driven by change events. We compute the impact of different adaptation plans (rules) at runtime regarding the change events and plan the adaptation accordingly. Due to these characteristics of our scheme, HypeZon supports scalability at runtime independent of the size of the architecture.

The solution by Pandey et al. continuously operates an optimization-based planner in the background, thus delivers high-quality adaptation plans in terms of system quality objectives as soon as the plan is ready. However, in the mean time, the hybrid planner executes a deterministic, sub-optimal planner ($\mathcal{R}1$). In essence, compared to developing a hand-crafted planning solution, instantiating a hybrid solution that uses off-the-shelf planners imposes lower cost in terms of development effort [see 63]. Therefore the solution by Pandey et al. is cost-effective ($\mathcal{R}2$). The solution is runtime efficient and the hybrid planner is shown to outperform its constituent planners; the scalability of the approach is not investigated and remains unknown ($\mathcal{R}3$). Quality-cost trade-offs are explicitly captured ($\mathcal{R}4$). The solution does not support dynamic architectures ($\mathcal{R}5$) and partially, for the reactive planner, requires detailed knowledge of user and system preferences to construct a utility function while the learning-based planner may operate based on the knowledge acquired during the off-line training ($\mathcal{R}6$). Finally, while the solution by Pandey et al. is designed to particularly coordinate a deliberative optimization-based and a reactive policy, it remains generic in terms of the policy specifications and treats them as black-box, therefore, partially fulfills ($\mathcal{R}7$).

Caldas et al. [67, 132] propose a hybrid solution for self-adaptation where they combine control-theoretic principles to carry out an adaptation and evolutionary optimization to find the best adaptation plan. The approach requires a goal model, defined by the stakeholders, that captures system objectives regarding non-functional requirements. The adaptation process is realized in a two-layer architecture where the lower-layer is a closed feedback control loop, implemented as a PI controller, that provides for Monitor, Analyze, and Execution functionalities of an adaptation loop; the higher-layer is responsible for decision-making and synthesizing adaptation plans to be executed by the adaptation loop. The control parameters of the PI controller are tuned at runtime and during an evolutionary optimization to guarantee optimal performance with respect to runtime conditions. The adaptation loop processes change events to detect the need for adaptation. The higher layer searches for the best plan by performing a search-based heuristic to find best fitting values for a parametric formula, i.e., an algebraic equation evaluating a QoS property of the system at runtime. The correct values for tunable parameters, i.e., granularity and offset (scope) of the search that affect the success as well as the required time for the search algorithm, are discovered during an evolutionary optimization process. Upon convergence, the selected plan is handed to the lower layer for execution. The solution by Caldas et al. captures system quality objectives as an algebraic formula that is optimized at runtime ($\mathcal{R}1$). However, the optimality has the burden of executing two evolutionary optimizers at runtime, thus the approach is not cost-effective ($\mathcal{R}2$). The performance and scalability of the adaptation and the planner is not evaluated. The authors report that while using the parametric formula reduces the size of the adaptation space in terms of complexity and time, the proposed solution is still not free from problem of combinatorial state-space explosion ($\mathcal{R}3$). The solution does not explicitly capture quality-cost trade-off ($\mathcal{R}4$). Dynamic architectures are not supported ($\mathcal{R}5$) and the approach requires user-defined preferences formulated as the parametric formula that is optimized at runtime ($\mathcal{R}6$). Finally, the approach is designed to specifically combine control-theoretic formalism in the PI controller with evolutionary algorithms for optimization, thus is not generic ($\mathcal{R}7$).

Qian et al. [348] propose to combine goal-based reasoning and CBR for planning the adaptations. Past experiences of successful adaptations are retained as adaptation cases. When no similar past cases are available, goal reasoning is employed to provide adaptation solutions requiring a goal model of the system. Violation of the system quality attributes, e.g., response time, trigger an adaptation. Monitoring and analysis details are not discussed. To plan an adaptation, the approach searches for similar adaptation cases from the case base using CBR. When the observed conditions do not comply with any of the existing recorded cases, an automated goal-based reasoning steers the planning by identifying plans that adapt the system towards satisfying the violated goals. If the adaptation is successful, i.e., quality requirements are now satisfied, the adaptation solution is added to the case base as a new case. The automated goal-based reasoning conducts a search in the space of possible configuration changes, i.e., plans, and evaluates the satisfaction level of system goals by each plan. The plan yielding the highest goal satisfaction score is selected for execution. The approach by Qian et al. satisfies system quality objectives ($\mathcal{R}1$) but conducts time-intensive CBS, thus is not cost-effective ($\mathcal{R}2$). The performance of the whole adaptation loop and scalability is not investigated ($\mathcal{R}3$). Quality-cost trade-offs are not captured explicitly ($\mathcal{R}4$). Dynamic architectures are not supported ($\mathcal{R}5$). The approach requires design-time knowledge of system objectives in the form of a goal model ($\mathcal{R}6$) and is designed to combine specific policies for planning, therefore, is not generic ($\mathcal{R}7$).

10.2.5 *Discussion*

In this section, we investigated how state-of-the-art in planning mechanisms satisfy the requirements for architecture-based self-adaptation introduced in Table 1.1. More specifically, among the large body of work on mechanisms for self-adaptive systems, we targeted works that are concerned with quality-cost trade-offs during planning. In the following, we summarize the findings of this section.

$\mathcal{R}1$**: Solution satisfies system quality objectives at a desirable level**
Majority of the approaches are concerned with providing adaptation plans that improve system quality objective, e.g., MADAM/MUSIC, Moreno et al., Franco et al., Hermes, Kim et al., FUSION, Zhao et al., Pandey et al., Caldas et al., and Qian et al. Among these approaches, Hermes, Kim et al., FUSION, and Pandey et al. only partially offer optimality guarantees while the rest guarantee to provide optimal adaptation plans in terms of system quality objectives. PLASMA, MOSAICO, SASSY, IDES, and Rainbow framework do not have any optimality gauntness. Three out of the four approaches that employ machine learning during their planning satisfy the quality requirement (either fully or partially).The solution by Kim et al. and FUSION may perform sub-optimally in terms of system objective satisfaction until the RL learners converge. In general, guarantees of convergence in learning-based methods are often not possible. The investigated studies employ different forms of tabular RL, e.g., Q-learning; for this class of learning methods convergence can be guaranteed. However, the learner is bound to sub-optimal performance until the learning is complete. In case of Zhao et al., the approach uses an off-line trained model for planning until the online learner converges. Learning-based approaches, however, often suffer from scalability and high adaptation cost in terms of resources and time.

Our investigation revealed that the search and optimization-based solutions, discussed in Section 10.2.1, while primarily aim at maximizing quality objective satisfaction, often cannot provide optimality guarantees, e.g., PLASMA, MOSAICO, and

SASSY, or only have partial claims, e.g., Hermes. This is partly attributed to the complexity of the large adaptation spaces for highly configurable systems where, for certain search and optimization techniques, local optima and state-space explosion is inevitable. Given the assumptions listed in Table 4.1, VENUS provides optimality guarantees by pursuing a greedy optimization before each adaptation, where runtime context of the change events are used to estimate the impact of the applicable adaptation rules on system utility and cost. VENUS employs a utility functions (utility-change prediction models) to compute (estimate) the impact of the adaptation rules at runtime. While the values for the parameters of the utility function are captured and updated at runtime, the function itself is defined (trained) at design time (off-line).

Rainbow, as the only static, rule-based solution investigated in this chapter, selects the adaptation plans that maximize the system utility, however, due to employing design-time estimates of adaptation rule (tactic) impacts on system utility, optimality claims are not supported by the framework. VENUS is distinguished from deterministic solutions such as Rainbow as the runtime context of the adaptation rules in VENUS is considered in estimating their impact on system utility. Additionally, the execution of adaptation rules in VENUS is event-based (ECA rules) rather than state-based. Hence the adaptation rules capture the change events instead of being pre-assigned to certain system configurations. In our approach, each adaptation rule has an initial condition, enabled by a change (event) in the system. This condition needs to be satisfied, so that the rule becomes applicable. However, the condition does not provide enough information to drive the adaptation process. Thus, in contrary to rule-based approaches such as Rainbow, VENUS dynamically assigns applicable rules to issues based on the runtime estimations of their costs and impacts on the utility using relevant fragments of the RTM that provides the system state.

Regarding hybrid approaches, Pandey et al., similar to HYPEZON, employ a coordinating planner that coordinates off-the-shelf policies, thus optimality claims are only applicable to intervals where the optimization-based solution is in charge of the planning. The remaining hybrid solutions, i.e., Caldas et al. and Qian et al., deliver adaptation plans that are the outcomes of runtime search and optimization processes. In the context of HYPEZON, optimality claims are only partially supported, i.e., when runtime circumstances allow HYPEZON to execute the optimization-based policy.

$\mathcal{R}2$**: Solution is cost-effective**

Reducing the adaptation cost is a first-class entity in several of the approaches; examples are Moreno et al., MOSAICO, HERMES, IDES, Zhao et al., Rainbow, and Pandey et al. An important observation is that, apart from VENUS, only two approaches, i.e., Moreno et al. and Zhao et al., simultaneously satisfy the quality satisfaction requirement $\mathcal{R}1$ and tend to operate cost-effectively—see Table 10.1. This is of course to be expected as the quality and cost of the adaptation are contradiction objectives—see [137, 214]. Achieving high quality adaptation plans often requires an exhaustive search in the possible adaptation space which renders attainment of optimal adaptation plans time-intensive [85]. While reactive condition-based solutions for adaptation deliver adaptation plans timely, e.g., MOSAICO and Rainbow, they often cannot guarantee to provide the optimal solutions. Additionally, solutions that execute expensive optimization at runtime, e.g., MADAM/MUSIC, Franco et al., Kim et al., FUSION, Caldas et al., and Qian et al., can manage to find the optimal configuration but there is no guarantee to reach the result within a reasonable time. Among the learning-based solutions discussed in Section 10.2.2, the IDES framework extends Rainbow, thus inherits its cost-effective design and execution. Zhao et al. leverage an off-line training step, thus avoids the cold start issue affecting the performance of online learning solutions.

Among the hybrid solutions discussed in Section 10.2.4, only Pandey et al. and HypeZon have a cost-effective approach for self-adaptation while executing time-intensive optimization algorithms at runtime prevents Caldas et al. and Qian et al. to exhibit cost-effective adaptation. In this context, as marked in Table 10.1, we consider HypeZon, similar to the coordinating hybrid planner by Pandey et al., a cost-effective solution for adaptation because both approaches employ off-the-shelf polices for planning, thus compared to alternative solutions that design a planner from scratch, impose minimal design and development effort. HypeZon is a generic hybrid planner that can coordinated arbitrary adaptation policies and its application is not limited to the two showcase, rule- and optimization-based policies demonstrated in this thesis.

$\mathcal{R}3$**: Solution is scalable**

Scalability of the adaptation solutions is the most neglected requirement and has only been partially evaluated in FUSION. Remaining of the studies, while in part reporting on the performance of the planning or the whole adaptation loop, either do not report about the scalability, e.g., Moreno et al., PLASMA, MOSAICO, SASSY, Hermes, Kim et al., IDES, Zhao et al., Rainbow, Pandey et al., and Qian et al., or explicitly report that their approach cannot be executed in a scalable manner, e.g., MADAM/-MUSIC, Franco et al., and Caldas et al. Venus has been extensively stress-tested and evaluated for scalability across different architecture sizes, input trace complexities, and different application examples in Section 7.3.1, Section 7.4.1, and Section 7.5.2. In Section 9.3.3 we have investigated HypeZon for scalability. In the context of HypeZon, while being integrated in an incremental adaptation loop where each activity is executed in an event-based manner, unlike Venus, making scalability claims is not straight forward, as the performance of the hybrid planner depends on the one of its constituents. In this thesis, enabled by model-driven adaptation, state- and event-based principles are explored for incremental adaptation, thus the adaptation mechanism demonstrates scalability and runtime efficiency for large and dynamic software architectures with growing size and complexity. Scalability in our solutions is achieved through incremental processing of adaptation activities and leveraging the locality information of the changes affecting the self-adaptive software.

In our earlier work [176, 177], by means of an SLR, we identified a set of required improvements in evaluation of self-healing systems. Our findings revealed that while majority of the investigated work with performance claims use simulation-based evaluations, only a few studies properly evaluate the self-adaptive software based on input representing a realistic operation context. This entails evaluation of the proposed solution with multiple input traces with volatile characteristics in their simulated evaluations—see Section 6.3 for a detailed discussion. Our findings in [176, 177] suggested that proper design and evaluation of self-adaptive software still remains an open issue, since critical elements for the design space of a self-adaptive system under evaluation, as identified by [270], are often missing. The result of investigating $\mathcal{R}3$ in the state-of-the-art on solutions for architecture-based self-adaptation in this section is inline with our previous findings in [176, 177]. This phenomenon is captured and addressed in this thesis via implementing an evaluation methodology based on the results of our SLR—see **C8** in Section 1.4.

$\mathcal{R}4$: Solution explicitly captures quality-cost trade-off policies
The trade-off between the quality and cost of the adaptation is often explicitly captured via a form of an objective (utility function) or a goal model, e.g., MADAM/MUSIC, Moreno et al., Franco et al., MOSAICO, SASSY, Hermes, FUSION, IDES, Rainbow, and Pandey et al. Approaches that achieve a cost-effective solution, captured by $\mathcal{R}2$, consider an explicit encoding of cost in the objective function, e.g., Moreno et al., MOSAICO, Hermes, IDES, Rainbow, and Pandey et al. The only exception is Zhao et al., where the authors use off-line training for the planner to avoid a time-intensive learning process at runtime, thus achieving cost-effective execution while the adaptation cost is not considered as a defining factor during planning and therefore is not explicitly captured by the utility function that defines system reward for learning.

In VENUS and HYPEZON, we employ utility functions to steer the adaptation. Our notion of pattern-based utility definition, introduced in Section 3.1, allows us to not only use utility values to capture system quality attributes, e.g., performance and response time, but also capture the cost of each adaptation rule that yields the envisioned improvements. Additionally, we consider the estimated execution time of individual adaptation rules, i.e., adapting the software, during planning—see Section 4.3.4 and Section 5.3.3. In this thesis, we do not distinguish between the execution time and latency of rules, i.e., the time until an adaptation affects the system after its execution, as we assume immediate adaptation effects (repairs). To explicitly consider latency, similarly to Moreno et al., the latency of each adaptation rule needs to be estimated and then added to the execution cost of the rule.

$\mathcal{R}5$: Solution supports dynamic architectures
Adaptable software with dynamic architectures are prone to evolution during system execution, thus adaptation complexity is subject to change [321]. MADAM/MUSIC, PLASMA, SASSY, Kim et al., and FUSION support adaptation of dynamic architectures. However, all five approaches provide flexible planning accommodating the runtime requirements of dynamically evolving architectures at the price of time-intensive planning. Our investigation on the treatment of the adaptation cost ($\mathcal{R}2$) revealed that the five solutions that support adaptation of dynamic architectures are not cost-effective—see satisfaction of cost ($\mathcal{R}2$) and support for dynamic architectures ($\mathcal{R}5$) in Table 10.1. We attribute this observation to the employed mechanisms by the approaches to provide for runtime flexibility and architecture evolution. PLASMA employs a varinat of online planning based on model-checking where architecture evolution is considered before generating plans. SASSY executes a hill-climbing heuristic during software execution and allows for architecture evolution. MADAM/MUSIC use the runtime context to update weights of a utility function. Kim et al. and FUSION employ online learning for planning where dynamic architecture models are catered. In this thesis, we primarily target dynamic architectures—see **C2** in Section 1.4—that may change in size and complexity by considering the architectural RTM of the adaptable software as a graph, consequently, defining utility functions, adaptation rules, and adaptation issues as graph patterns that may easily be modified or extended to support architecture evolution.

$\mathcal{R}6$: Solution addresses problem of initially unknown runtime knowledge of user and system preferences

Similar to this thesis, $\mathcal{R}6$ is satisfied by approaches that support systematic acquisition of utility prediction models for black-box systems where prior knowledge is missing or becomes invalid at runtime. Such approaches use machine learning to either train offline, e.g., Zhao et al. or learn online, e.g., Kim et al. and FUSION, predictors for the effect of the adaptation decisions on the system utility and reward. These solutions address the problem of initially unknown runtime knowledge to support adaptation. The remaining solutions require the developer to provide a form of utility function based on detailed knowledge of the user or system preferences for every specific adaptation decisions necessary to restore the system to the desired states. As a result, these solution cannot be deployed on black-box systems. In Section 3.3, we presented a methodology to train utility-change prediction models for complex, highly configurable systems or systems with a black-box model where it is often not trivial to obtain the relevant values to manually construct a utility function. The adaptation loop then, executing either Venus or HypeZon, may use the prediction models to estimate the impact of the adaptation rule applications on system utility when a manually constructed utility function is not available.

$\mathcal{R}7$**: Hybrid solution is generic**

Similarly to HypeZon, Pandey et al. propose a coordinating hybrid planner for off-the-shelf policies. However, while the design of HypeZon is independent of its constituent policies, the solution by Pandey et al. requires the policies to conform to two specific categories of formalisms, i.e., reactive policies with cost-efficiency guaranties and optimization-based policies with quality guarantees. Therefore, their approach can be considered generic in a sense that despite the constraint on the policy type, the hybrid planner is designed to coordinate arbitrary policies of the expected type and considers them as black-box planners. However, the said constraint limits the applicability of the proposed solution by Pandey et al. to the specified categories. The other two hybrid solutions, i.e., Caldas et al. and Qian et al., are designed around specific policies and cannot be generalized for arbitrary policies. We distinguish HypeZon from Caldas et al. and Qian et al. on the ground that the main focus in HypeZon is on the orchestrating entity, i.e., the realization of the decision-making mechanism within the hybrid planner. HypeZon has the characteristics of a generic hybrid planner since it considers the employed adaptation policies as a black-box, thus can coordinate arbitrary adaptation policies.

10.3 PREDICTION MODEL ACQUISITION MECHANISMS

In this section, we discuss the related work for our approach to train prediction models for utility-change introduced in Section 3.3. The scope of the related work in this section is not limited to the research field of self-adaptation and autonomic computing. The reason is that, as also confirmed by Gheibi et al. in their 2021 SLR [182], only four studies, including our work [175], are concerned with addressing the problem of initially unknown runtime knowledge to support adaptation [348, 384, 402]. We deem the work by Qian et al. [348] slightly irrelevant to this list since, as discussed earlier in Section 10.2.4, the approach requires a user-defined goal model that captures system objectives and desired satisfaction levels, thus does not employ black-box performance models as the other three studies.

Skałkowski et al. [384] and Tesauro et al. [402] use model-free RL to train prediction models for action selection at runtime. Skałkowski et al. use RL during system execution to discover adaptation actions that best satisfy system objectives, however, they only provide early state research and offer proof-of-concept implementation and do not report on runtime performance and quality of the adaptable software as well as the learning mechanism. Tesauro et al. propose an RL-based solution where an off-line training supports the runtime decision-making to identify management policies.

Runtime planning heavily depends on accurate performance reasoning. Black-box performance models have been proven to ease understanding, debugging, and optimization of configurable systems. Additionally, growing complexity of software systems makes the software reconfiguration challenging as complex software architectures have considerably large number of configuration attributes that can be modified, making the system highly configurable [383]. The exponential configuration space, complex interactions, and unknown constraints make it difficult to attribute system performance to different parameter configurations via a white-box performance model.

In the context of performance reasoning, model-based performance prediction techniques have been vastly practiced [23, 33]. Time series techniques are applied for predictive modeling of response time [29]. Machine learning methods have been used to model online QoS for cloud-based software services [85] and virtualized applications [23]. Esfahani et al. propose a generic feature-oriented methodology to reduce the dimensions of the configuration space [129]. In this thesis, although we do not propose a solution for feature selection, we avoided this problem by allowing the decision trees to perform it automatically. Performance prediction and reasoning have also been used extensively in control theory [see 138]. Kim et al. [255] employ RL for online planning. Filho et al. [136] propose an online, unsupervised learning as a model-free solution to explore alternative system assemblies and Porter et al. [341] use online learning to steer adaptation at runtime. Online learning solutions suffer from challenges such as scalability and cold starts. In this thesis, we opted for off-line learning which allowed us to exclude problems of scalability and cold start.

Angelopoulos et al. [10] use machine learning to cope with dynamics that cannot be captured by the analytical model, as a result of migrating from the simulator to the real system. Tesauro et al. [401, 402], briefly discussed above, propose a hybrid approach that combines queuing network with RL to make resource allocation decisions. In multi-agent domains, the state space or configuration space is assumed to be fully discoverable, while in the field of self-adaptive systems this space can be extremely large and as pointed by Elkhodary et al. [126], the computational cost of the search for an optimal architectural configuration increases exponentially. We tackled this problem by systematically covering the configuration space via running a random simulator to generate data and using complex functions with high variability as ground truth.

Standard machine learning techniques, such as support vector machines, decision trees, and evolutionary algorithms have been tried to acquire performance models [61, 125, 219, 433]. These approaches trade simplicity and understandability of the learned models for predictive power [182]. Other automated planning techniques, e.g., transfer learning [325] and genetic algorithms [97], have been explored to generate adaptation plans at runtime. Neural networks and deep learning have also been successfully applied in many settings, including dynamic programming for expectation maximization [431] and learning hierarchical representations [361]. Nevertheless, they suffer from many problems, e.g., hyperparameter optimization, overfitting, scalability issues, and are not well-controlled learning machines [407].

In transfer learning, system learns a prediction model from previous experiences on cheaper sources, e.g., a simulator of the real system, with lower costs to accelerate model learning and the experience is then transferred to the real system at a later point in time. This technique has been explored for configuration optimizations by exploiting the dependencies between configurations attributes. Jamshidi et al. [232] employ transfer learning where a simulator of the real system is used to generate cheap samples and a regression model is used to learn the relationship between simulator and real system. A cost model is defined to transform the traditional view of model learning into a multi-objective problem that considers model accuracy and measurement effort. Similarly, this thesis also aims at learning the prediction models for self-adaptive systems, but in addition, we also propose a methodology that generates and selects models without the need to run a real system. Ultimately, in preference learning [see 318], the goal is to learn a utility function when the user preference information is uncertain. Conversely, we do not require knowledge of user preferences or system internal performance model to learn the system utility.

This thesis is concerned with model learning where sampling is a defining factor. Random sampling is known to be the proper way to collect unbiased data but it requires large sample set—see [383]. When sufficient data is not available, sparse sampling methods such as Box-Behnken and Plackett-Burman have been practiced in context of experimental design to ensure certain statistical properties [195]. Sarkar et al. [368] determine optimal sampling via considering cost as an explicit factor. Ye et al. [432] employ *Recursive Random Sampling* (RRS) which integrates a restarting mechanism into the random sampling to achieve high search efficiency. Employing a simulator makes it affordable to have large volumes of training data for machine learning. In this thesis, we employ a random adaptation policy to systematically generate unbiased data and broadly explore the configuration space of the adaptable software.

10.4 SUMMARY

In this chapter, we investigated the related work in three categories; Section 10.1 discussed our approach to developing a solution for architecture-based self-adaptation in comparison with the landscape of techniques for development of self-adaptive software, where the artifact of concern is the software architecture. Our work was shown to be well inline with the state-of-the-art regarding employing some form of software model during development of the solution and at execution time. Our realization of an adaptation engine uses causally connected RTMs leveraging MDE techniques to support consistent co-evolution of the software architecture and its RTM. While the technical contributions of this thesis, i.e., HypeZon and Venus, mostly concern the Analyze and Plan activities of the MAPE-K feedback loop, leveraging the incremental Monitor module from Vogel et al. [418], the contributions are embodied in a MAPE-based adaptation engine that is executed incrementally. Enabled by model-driven adaptation, this thesis explores both state- and event-based principles for incremental adaptation, thus the adaptation mechanism demonstrates scalability and runtime efficiency for large and dynamic software architectures with growing size and complexity. This way, this thesis offers supporting mechanisms for efficient execution of the adaptation loop.

Section 10.2 discussed the related work regarding their treatment of quality-cost trade-off in architecture-based adaptation of software systems. The state-of-the-art was investigated against a list of six requirements, driven from literature, for the problem at hand. An additional requirement $\mathcal{R}7$ was added in this chapter for hybrid solutions. Venus covers all of the requirements in contrast to the state-of-the-art. Embodied in an incremental framework for execution of the MAPE-K loop activities, Venus brings together the timeliness and cost-effectiveness of the adaptation rules and the high quality of utility-based optimization, thus satisfies $\mathcal{R}1$ and $\mathcal{R}2$. Furthermore, owing to the event-based detection of the adaptation triggers followed by incremental execution of the Analyze and Plan activities, Venus provides for a scalable and runtime efficient solution, thus satisfies $\mathcal{R}3$. Scalability of Venus is its most notable feature that has been extensively investigated in this thesis. One of the major contributions in this thesis, leveraged by both Venus and HypeZon, is the unique way of defining utility values for architectural patterns. Analogously, defining ECA rules as graph transformation rules where the LHS captures matches for issues in the RTMs, utility values are mapped to the matches of the rules. Thus, runtime context, including cost of the adaptation rules, is considered in calculation of utility values satisfying $\mathcal{R}4$. Venus stands out among the state-of-the-art regarding its efficient execution and support for large, dynamic architectures with growing size and complexity $\mathcal{R}5$. Finally, the adaptation engine proposed in this thesis my leverage utility-change prediction models that are constructed during an off-line training phase and executed at runtime. Consequently, the non-trivial process of utility elicitation for complex and black-box systems can be addressed $\mathcal{R}6$.

HypeZon offers only partial coverage to some of the requirements, i.e., $\mathcal{R}1$ and $\mathcal{R}3$, but still performs relatively better in comparison to the state-of the-art. As a coordinating hybrid planner, the quality and scalability features of HypeZon are determined by its constituting polices. Therefore, while the incremental adaptation loop embodying HypeZon supports scalable execution, we cannot guarantee this feature for HypeZon in general as it is dependent on the execution complexity of the off-the-shelf policies that are coordinated by HypeZon. The same arguments apply for the quality of the adaptations performed by HypeZon. Similar to Venus, HypeZon builds on the unique features of the general MAPE-K loop realization in this thesis, thus satisfies $\mathcal{R}4$., $\mathcal{R}5$., and $\mathcal{R}6$ on the same grounds as Venus that we discussed above. Finally, HypeZon is designed as a generic hybrid planner to coordinate arbitrary policies ($\mathcal{R}7$); the scheme focuses on the coordinating mechanism, rather than the inner-workings of the off-the-shelf- policies.

Section 10.3 discussed the related work for our methodology to obtain prediction models for utility-change values. In this thesis, we introduced an off-line learning-based solution whereby, provided randomly generated training data, three different model-free learners were employed to learn prediction models for rule-based adaptation. Our solution, in comparison to the state-of-the-art, has the following points of strength: the solution is off-line, thus does not introduce any runtime overhead during software adaptation allowing us to exclude problems of scalability and cold start. However, dynamicity of the software architecture and its context might render the learned models out of context, thus demanding the models to be re-trained. The choice of learning algorithms has been made such that hyperparameter optimization is automated. The solution is distinguished from the state-of-the-art as we mitigated the common issues of overfitting via employing ensemble machine learning methods, i.e., building a collection, or an ensemble of learners, as apposed to employing a single strong predictive model, for data-driven modeling tasks and combining their predictions.

Overall, this thesis goes beyond the state-of-the-art in architecture-based software self-adaptation by putting forward two alternative solutions, varying in expressiveness and development effort, while addressing the requirements for software self-adaptation relatively well. In contrast to existing approaches that often provide either high quality or cost-effective solution for software adaptation, this thesis fulfills both requirements in a scalable manner while supporting dynamic architectures as well as black-box systems.

11 CONCLUSION AND FUTURE WORK

11.1 CONCLUSION

We described the scope of the problem that is addressed by this thesis via a set of requirements for architecture-based self-adaption of software systems (summarized in Table 1.1). The solution is required to satisfy the system quality objectives at a desirable level ($\mathcal{R}1$) and in a cost-effective manner ($\mathcal{R}2$). Being concerned with the adaptations of large and highly configurable software systems, the solution should be scalable to cope with the complexity caused by the possible combinatorial explosion of the solution space ($\mathcal{R}3$). To address the systematic development of self-adaptive software that is capable of maintaining multiple quality objectives, cost-effectively and at runtime, we need models that explicitly capture the quality-cost trade-off policies in the adaptation mechanism ($\mathcal{R}4$). Such an explicit model enables the engineers of the adaptable systems to decide on how to reflect on the quality-cost trade-off according to the runtime condition. Adaptable software with dynamic architectures are expected to evolve during system execution, thus the solution is required to support dynamic architectures ($\mathcal{R}5$). The runtime uncertainty affecting the self-adaptive software hinders the expectation to have an omniscient decision-maker that knows user/system preferences at any given time and under different operation conditions. The solution should address the problem of initially unknown runtime knowledge of user and system preferences ($\mathcal{R}6$).

This thesis addresses these requirements via a twofold solution for incremental self-adaptation of dynamic software architectures based on (i) design-time combination (Venus) and (ii) runtime coordination (HypeZon) of rule- and optimization-based formalisms of adaptation policies. The proposed approaches build on the notion of pattern-based utility to evaluate dynamic architectures. Venus is a utility-driven rule-based scheme for engineering self-adaptation of large dynamic architectures. The scheme combines *Event-Condition-Action* (ECA) rule- and optimization-based formalisms in its design. The rule-based approach defines failures or performance issues that can be directly identified and localized as faults or bottlenecks in architectural *Runtime Model* (RTM); the RTMs are then evaluated by assigning utility values to fragments of the architecture (patterns). Consequently, the impact of the rule applications on the RTM can be mapped to utility values. In addition to considering the expected impact of the adaptation rules on utility, the cost of the rule applications affect the decision-making in Venus, thus the scheme obtains optimal adaptation decisions that maximize the utility ($\mathcal{R}1$) in a cost-effective manner ($\mathcal{R}2$). Scalability in Venus is achieved through incremental execution of the adaptation loop activities. Incremental processing of the changes are supported via leveraging the locality information of the change events, thus avoiding the search in possibly large adaptation spaces ($\mathcal{R}3$). Leveraging utility functions to steer the adaptations, Venus explicitly captures the quality-cost trade-off policies ($\mathcal{R}4$). As a result, the utility values capture system quality attributes, e.g., performance and response time, but also capture the cost of each adaptation rule that yields the envisioned improvement. Dynamic architectures are supported via realizing the architectural RTMs of the software as graphs that are then evaluated using our novel approach for pattern-based utility and are modified using graph-transformation rules ($\mathcal{R}5$). The problem of initially unknown knowledge of user and system preferences is addressed via proposing a methodology to systematically train utility-change prediction models for black-box systems ($\mathcal{R}6$).

The second part of the solution is HypeZon. HypeZon complements Venus by balancing the quality-cost trade-off in software self-adaptation via runtime coordination of off-the-shelf policies. HypeZon is integrated in our incremental adaptation engine and offers an alternative solution for the Plan activity of the adaptation loop. The scheme supports cost-effective achievement of system quality objectives while, compared to solutions such as Venus, reduces development cost, since it omits the effortful process of developing new algorithms/heuristics from scratch ($\mathcal{R}2$). However, hybrid solutions are by definition bounded by their individual constituent approaches in terms of cost and quality. Conforming to meta-self-aware architectures, we proposed two designs for HypeZon that support explicit separation of concerns, i.e., adaptation and policy coordination, at the architecture level; as a result, reusability, easier maintenance, and independent evolution of each level are supported in HypeZon. This allows for separate mechanisms for observing and reasoning logic to be employed by each level. HypeZon, analogously to Venus, is integrated in an incremental adaptation engine, thus supports incremental execution improving scalability ($\mathcal{R}3$). HypeZon satisfies the requirements for architecture-based self-adaptation listed in Table 1.1 on the same grounds as Venus with two exceptions; the quality (optimality) and scalability guarantees in HypeZon, i.e., $\mathcal{R}1$ and $\mathcal{R}5$ respectively, depend on its constituent policies. Finally, HypeZon is a generic hybrid planner, i.e., it considers the individual policies as black-box and can coordinate arbitrary policies (see $\mathcal{R}7$ for hybrid solutions in Table 10.1).

The majority of the planning mechanisms for architecture-based self-adaptation either provide optimality guarantees regarding system quality objectives ($\mathcal{R}1$) or are cost-effective ($\mathcal{R}2$). The few exceptions that satisfy both the quality and cost requirements do not consider adaptations of large and dynamic architectures ($\mathcal{R}5$). The state-of-the-art offers only partial solutions for quality-cost trade-off without providing any supporting evidence for scalability of the proposed solutions ($\mathcal{R}3$). The trade-off between the quality and cost of the adaptation is often explicitly captured via a form of an objective (utility) function or a goal model ($\mathcal{R}4$). Furthermore, apart from few exceptions, the planning mechanisms rely on availability of a well-defined and representative utility (reward) function that captures the runtime knowledege on the preferences ($\mathcal{R}6$). In contrast to other proposals for architecture-based self-adaptation, Venus satisfies the quality and cost requirements of software adaptation and in a scalable manner. The unique way of defining utility values for architectural fragments in the architectural RTM in combination with pattern-based realization of the adaptation issues and of the adaptation rules results in efficient and scalable execution of Venus over large and dynamic software architectures with increasing adaptation complexity. Therefore, Venus goes beyond the state-of-the-art in providing guarantees for optimality, cost-efficiency, scalability, and support for dynamic architectures. Furthermore, we extended the conventional supervised learning methods and developed a systematic methodology to train prediction models for changes of utility during adaptation. As a result, in contrast to the majority of the related work, both Venus and HypeZon may operate on systems with black-box models where prior knowledge is not available or becomes obsolete.

HypeZon contrasts the existing hybrid solutions for software self-adaptation because it offers explicit design at the architecture level and is designed to be generic and independent of the inner-workings of its individual constituent policies. Finally, in the context of re-engineering and using our solution, this thesis contrasts the existing work on two grounds: (i) it provides two complementary solutions for addressing the quality-cost trade-off; (ii) it decouples the design and engineering of the utility function element from the elements of the feedback loop. The former provides the user of our solutions to select the adaptation scheme that fits the complexity of the problem at hand; the latter allows the engineers to freely construct their desired utility function module as long as it complies to the provided input and required output format for rule-based systems—see Figure 1.2 for an overview of the utility function building block and the adaptation engine.

Therefore, this thesis makes the following contributions to the state-of-the-art in architecture-based self-adaptation of software systems. **C**1: incremental execution of the adaptation loop, thus attaining scalability. Both Venus and HypeZon are integrated in an adaptation engine that leverages state- and event-based principles to enable incremental adaptation. Event-based execution of the adaptation loop activities allows for only processing the changes, rather than the complete state, while the architectural RTM provide system state as context, thus supporting cost-effective execution of the adaptation and attaining scalability. **C**2: defining pattern-based utility functions for dynamic architectures. Via introducing the notion of pattern-based utility, utility values are defined for graph patterns that represent architectural fragments. Pattern-based utility supports incremental computation of impact of rule applications on the utility by constraining the computation effort to the modified architecture fragments. Consequently, dynamic architectures are evaluated with respect to the present positive and negative architectural utility patterns. **C**3: training utility-change prediction models for rule-based self-adaptive software. The problem of initially unknown knowledge to support adaptation is addressed via a methodology to systematically train prediction models for utility-changes in rule-based, self-adaptive software systems. The standard supervised learning process is modified and extended to find a proper approximation for the analytical utility-change function by means of prediction models. **C**4: combining rule- and optimization-based formalism of adaptation policies at design time, thus bringing together their benefits, i.e., timeliness and scalability of the former and optimality of the latter. The pattern-based characterization of adaptation rules and utility functions allows Venus to map rule applications to utility values; consequently, Venus can compute (predict) the impact of each rule application, in terms of utility and cost, during planning. **C**5: an optimal solution for self-adaptation with negligible runtime overhead. For a class of adaptation problems that satisfy the greedy choice property, Venus guarantees optimality of system utility and reward. Capturing the adaptation decisions via ECA rules, pattern-based characterization of runtime issues, rules, and utility values over architectural RTM, and finally, incremental execution of the adaptation loop activities, collectively, contribute to a solution that introduces negligible runtime overhead. **C**6: a robust solution. Venus is a robust approach, i.e., it consistently shows timeliness (high performance), scalability, high utility, and effectiveness in the presence of exceptional inputs, stressful environmental conditions, and even during violation of its validity assumptions. **C**7: a generic scheme for hybrid self-adaptation. HypeZon uses off-the-shelf policies for adaptation and balances the quality-cost trace-off during adaptation via runtime coordination of multiple policies. HypeZon is intended for the fragment of adaptation problems that either do not satisfy the greedy choice property, thus render VENUS sub-optimal, or when VENUS is deemed as over-engineering solution for the problem at hand. **C8**: coverage of a wide spectrum of self-adaptation problem space beyond the state-of-the-art. The solutions in this thesis are evaluated according to an evaluation methodology derived from our *Systematic Literature Review* (SLR) on self-healing systems. The methodology entails the evaluation of the solutions across a large and diverse set of input traces to verify the generalizability and conclusiveness of the results.

To provide evidence for the contributions of the thesis, we assessed the impact of the incremental execution of the adaptation engine on the runtime performance and scalability of the adaptations. We comprehensively evaluated VENUS with respect to the quality attributes of performance, scalability, optimality, and robustness. We investigated the impact of hybrid planning and different designs for HYPEZON on the quality and cost of the adaptation. To assess the generalizability of the results, large variety of the input traces were used across different architecture sizes. The solutions were instantiated to several self-adaptation problems based on two different application examples. VENUS was compared with several alternative solutions for planning and HYPEZON was compared against an alternative solution for hybrid planning. We discussed how VENUS and HYPEZON fulfill the requirements for architecture-based self-adaptation. Finally, we systematically evaluated the proposed methodology for training prediction models and assessed the quality and the performance of the methodology, of the prediction models, and of the model selection mechanism. The results from the evaluations indicate that: (i) we have achieved an incremental adaptation engine that can integrate arbitrary planning mechanisms and improves the base-line for the execution time. (ii) We have achieved in this thesis an optimal, cost-effective, scalable, and robust approach for self-adaptation of large and dynamic architectures with VENUS. (iii) We have trained utility-change prediction-models for rule-based systems that can approximate an analytically-defined optimum with minimal error. (iv) We have achieved with HYPEZON a generic and cost-effective hybrid approach for self-adaptation of large and dynamic architectures that improves the quality and timeliness of the adaptation.

11.2 FUTURE WORK

This section presents the outlook for the future work of the thesis. We outline the aspects that are the logical next steps to either realize the full potential of the current contributions or to address the existing limitations.

11.2.1 *Learning*

Closing the loop with online learning. Regarding the methodology to train utility-change prediction models, a natural next step is extending it to online learning. Online learning consists of training the system after deployment, with the intent of restoring the prediction accuracy of a machine learning component, i.e., the utility-change prediction models. The current limitation of our solution is that evolution of the adaptable software or its context might render the prediction models obsolete at some point during the software lifetime. The extension would entail closing the learning loop in the proposed methodology. This way, the accuracy of the prediction models are continu-

ously evaluated during system execution and re-training of the models can be initiated upon a certain prediction error threshold. Extending the methodology towards online learning require methods that support modularity in learning and account for data sparsity [231] and catastrophic forgetting [256]. Modularity entails part-wise training of the prediction models and only involves parts of the models that need to be improved. Data sparsity makes the online learning in self-adaptive systems challenging. The reason is that in the context of self-adaptive systems, and more specifically, self-healing systems, the runtime failures are in essence rare events to the extent that certain runtime issues might occur seldom during the software lifetime. Finally, catastrophic forgetting is the tendency of the learners to abruptly and drastically forget previously learned information upon learning new information. In the direction towards online learning, selecting the learning method should be based on these concerns. Some of the mitigation techniques comprise domain generalization [439] and its extensions based on techniques like meta-learning [411], domain adaptation [428], and representation learning [371].
Extending the scope of learning. In this thesis, we learned the impact of the adaptation rules on system utility, i.e., utility-change. In addition to the changes of utility, aspects such as the synthesis of new rules should be explored, thus extending the subject of learning. The rules in our instantiation of rule-based systems constitute the atomic units of change to a software architecture, therefore, software evolution might modify the architecture in ways that new kinds of change are applicable; this requires the addition of new rules. In turn, learning new rules requires data on how to change the system. This data should be in the form of (input, output) that result from observing the system before (input) and after (output) the change. This data is referred to as meta-data that is observable from a higher-level, i.e., meta-level. This includes using meta-learning methods. Meta-learning involves defining the meta-data to learn a new task/phenomenon by generalizing from existing ones in an unsupervised manner using reward functions learned from data [212].

11.2.2 Venus

Software improvement We have presented our contributions, i.e., utility functions and planning mechanisms, as building blocks of (or attached to) a MAPE-K-based adaptation engine that is engineered to support incremental execution. Building a self-adaptive system is a costly proposition if the important components such as the incremental monitoring, incremental analysis, and incremental execution have to be built from scratch. For this reason, improving the implementation to be presented as a framework with shared infrastructure for Monitor, Analyze, and Execute activities is beneficial. This process would promote reusability of incremental adaptation functionalities and reduce the cost of achieving self-adaptation for arbitrary self-adaptation policies that may be used as plugins to implement the Plan functionality.
Extending applicability of Venus. The presented adaptation scheme has limitations that should be addressed in future work. The limitations refer to the validity assumptions for Venus. Future extension of Venus should explore whether similar or sufficiently good, i.e., not necessarily optimal, results can be achieved by relaxing these assumptions. This direction has already started in the thesis by relaxing assumption (A2) (see Table 4.1) with probabilistic adaptation rules. As next steps, relaxing assumptions (A3b) and (A4) could be explored. Relaxing assumption (A3b) entails support

for the case where applying rules in Venus impacts the applicability of other rules. Relaxing assumption (A4) entails support for cases where executing a rule affects the impact of executing another rule on the overall utility. As a result, Venus would support dependencies among the adaptation rules. Regardless whether on the rule applicability level or rule impact on utility. This way, the applicability of Venus can be extended to software systems where defining independent rules is not feasible.

Support for integrated learning module. In line with the proposal for online learning, a foreseeable direction to extend Venus is to integrate the learning element in the design of Venus. The current design supports the deployment of the trained prediction models in an ad-hoc manner. Closing the loop via online learning requires an embedded element in the architecture of Venus. Venus may integrate the learning element as an additional loop to have separation of concerns at the architecture level. The separation would support easier testing, maintenance, and modifications to the loop. Furthermore, the potential for meta-self-adaptation together with meta-learning could be explored. In this thesis, we took a glimpse at meta-self-aware architectures with HypeZon. Combining these architectures with online learning should be investigated. As discussed in Section 11.2.1, meta-learning requires meta-data that is observable form a higher-level, i.e., meta-level. Therefore, meta-self-aware architectures with leaning capabilities is a promising direction to extend Venus towards supporting online learning. We have started a line of collaboration on the topic of learning in collective adaptive systems in [102, 103] that can contribute to this extension.

Proactive Venus. An interesting feature that we would like to add to Venus is incorporating proactive behavior. The scheme is designed to be reactive and uses change events as triggers. Reactiveness to software failures is necessary yet insufficient for some domains, e.g., safety-critical systems. Proactiveness decreases the aftereffects of changes, or improves control of change propagation. Proactiveness requires predictive capabilities. For this purpose, the *Left-hand Side* (LHS) of the adaptation rules should capture the symptoms that can predict an adaptation issue such that a proactive (preventive) adaptation action can be taken. We envision that the incremental behavior of the MAPE activities can support the proactive adaptation rules similarly to the reactive ones. There are various ways in which we may acquire predictive capabilities in Venus: model-based predictive approaches, e.g., model-predictive control [see 68], may be applied; predictive analytics methods, e.g., classifications, time-series analysis, regression methods, and linear and non-linear modeling also can be considered. Adopting a predictive approach in Venus requires the scheme to take into consideration the previous history data (from past monitoring data). Consequently, we should investigate efficient ways to capture, store, and retrieve history in Venus. We have started a line of collaboration on the topic of history-aware self-adaptation—see [362–364]. Future work on Venus should consider keeping the history of the adaptation as well as the system changes to enable proactive adaptations.

11.2.3 HypeZon

Evaluation. We have instantiated HypeZon on two application examples and for two off-the-shelf planning policies. HypeZon should be evaluated for multiple additional planning policies to further investigate its generalizability and its insensitivity to its

constituent policies. HypeZon variants include a set of control parameters whose full range has not been investigated during our evaluation. A more systematic evaluation of the variants that instantiates them in all their possible settings is beneficial and can improve algorithmic or design inefficiencies. So far, the focus of the evaluation has been on the functional aspects enabled by each design rather than the architectural aspects. Further evaluation of HypeZon should investigate the HypeZon variants regarding their differences in the architectural aspects. This allows for alternative features (besides the currently considered execution interval) to distinguish between the two variants of HypeZon and investigate the benefits of a certain variant over the other.
Technical features. One of the key areas of HypeZon that can be improved is its predictive aspect. Currently, HypeZon relies on average observation from limited number of system past executions and interpolates the measurements as expectations for future. Similarly to the discussion on adding prediction capabilities to Venus, applying predictive mechanisms to HypeZon improves its performance in case of non-stationary characteristics [see 392]. Because HypeZon uses expected system load during planning, techniques such as parametric and AI–based methods for load forecasting can enable HypeZon with for informed decisions regarding policy switch and control parameter tuning. Furthermore, HypeZon currently requires the expected utility and cost of the planning policies to be provided as input. Future improvements of HypeZon should remove this requirement by enabling the scheme to maintain performance profiles of the policies, i.e., their execution history, and instead use prediction methods to approximate the quality and cost of the policies for a given planning problem.

Balancing the quality-cost trade-off in HypeZon can be improved. Currently, we use step-wise reduction of the planning horizon until an affordable policy with the highest expected utility is found. Ideally, this process should be formulated as an optimization problem with constrains on the budget. This way, we can then claim that the optimal trade-off between the cost and quality is reached—this claim does not hold for the current instantiation of HypeZon. Similar improvements should be made for the execution horizon size tuning. The concern however is that this might add some additional overhead to the planning which should be taken into consideration.

Apart from the goal of explicitly separating the control loops at the architecture level that allowed for different execution timescales, we did not explore the potential of the two different designs for Venus; the aspect that should be investigated in the future work is exploring the potential of the meta-awareness loop in HypeZon, and HZ_e in particular. Certain tasks that are currently configured manually can be delegated to the meta-awareness loop. For instance, definition and modification of the policy switch thresholds should be done by the meta-loop as it holds a global view on the system and the adaptation loop. Additionally, to increase runtime flexibility of HypeZon, the ability to observe the adaptation loop could be navigated by the meta-loop.

Finally, concurrent planning in HypeZon is an aspect that is excluded from this thesis. Adding this feature to HypeZon can be studied in the future. While concurrent planning introduces the burden of consistency among multiple plans and between the plans and the planning problem, existing work suggests that it can be beneficial for optimization-based policies that require a fairly long time to provide an optimal plan [see 326]. Concurrent planning allows HypeZon to execute the time-intensive policies in the background while a fast, reactive policy runs in the foreground. The hybrid planner then switches to the optimal plan when ready. Going towards this direction however requires measures to obtain consistency between the output of the planners.

BIBLIOGRAPHY

[1] Agnar Aamodt and Enric Plaza. "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches." In: *AI Communications* 7.1 (1994), pp. 39–59. DOI: 10.3233/AIC-1994-7104.

[2] Alex Alexandridis, Panagiotis Patrinos, Haralambos Sarimveis, and George Tsekouras. "A Two-Stage Evolutionary Algorithm for Variable Selection in the Development of RBF Neural Network Models." In: *Chemometrics and Intelligent Laboratory Systems* 75.2 (Feb. 28, 2005), pp. 149–162. ISSN: 0169-7439. DOI: 10.1016/j.chemolab.2004.06.004. (Visited on 03/24/2022).

[3] Germán H. Alférez and Vicente Pelechano. "Dynamic Evolution of Context-Aware Systems with Models at Runtime." In: *Model Driven Engineering Languages and Systems*. Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 70–86. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_6.

[4] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. "Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control." In: *Proceedings of the 3rd Workshop on Scientific Cloud Computing*. ScienceCloud '12. New York, NY, USA: Association for Computing Machinery, June 18, 2012, pp. 31–40. ISBN: 978-1-4503-1340-7. DOI: 10.1145/2287036.2287044. (Visited on 05/16/2022).

[5] David M. Allen. "The Relationship between Variable Selection and Data Agumentation and a Method for Prediction." In: *Technometrics* 16.1 (1974), pp. 125–127. ISSN: 00401706. DOI: 10.2307/1267500. JSTOR: 1267500.

[6] Muhammad H. Alsuwaiyel. *Algorithms: Design Techniques and Analysis*. Vol. 5. Lecture Notes Series on Computing. World Scientific, 2021. ISBN: 978-981-02-3740-0.

[7] Ivan Dario Paez Anaya, Viliam Simko, Johann Bourcier, Noel Plouzeau, and Jean-Marc Jézéquel. "A Prediction-driven Adaptation Approach for Self-adaptive Sensor Networks." In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2014. New York, NY, USA: ACM, 2014, pp. 145–154. DOI: 10.1145/2593929.2593941.

[8] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. "Modeling Dimensions of Self-Adaptive Software Systems." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 27–47. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_2. (Visited on 01/21/2022).

[9] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. "Reflecting on Self-Adaptive Software Systems." In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. May 2009, pp. 38–47. DOI: 10.1109/SEAMS.2009.5069072.

[10] Konstantinos Angelopoulos, Alessandro Vittorio Papadopoulos, Vítor E. Silva Souza, and John Mylopoulos. "Model Predictive Control for Software Systems with CobRA." In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '16. New York, NY, USA: ACM, 2016, pp. 35–46. DOI: 10.1145/2897053.2897054.

[11] Konstantinos Angelopoulos, Vítor E. Silva Souza, and John Mylopoulos. "Dealing with Multiple Failures in Zanshin: A Control-Theoretic Approach." In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2014. New York, NY, USA: Association for Computing Machinery, June 2, 2014, pp. 165–174. ISBN: 978-1-4503-2864-7. DOI: 10.1145/2593929.2593936. (Visited on 07/14/2022).

[12] Karen Appleby, Sameh Fakhouri, Liana Fong, Germán Goldszmidt, Michael Kalantar, Srirama Krishnakumar, Donald P Pazel, John Pershing, and Benny Rochwerger. "Oceano-SLA Based Management of a Computing Utility." In: *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No. 01EX470)*. IEEE. 2001, pp. 855–868. DOI: 10.1109/INM.2001.918085.

[13] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation." In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. May 2015, pp. 13–23. DOI: 10.1109/SEAMS.2015.10.

[14] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Formal Design and Verification of Self-Adaptive Systems with Decentralized Control." In: *ACM Trans. Auton. Adapt. Syst.* 11.4 (Jan. 10, 2017), 25:1–25:35. ISSN: 1556-4665. DOI: 10.1145/3019598. (Visited on 10/25/2022).

[15] Rick Archibald and George Fann. "Feature Selection and Classification of Hyperspectral Images With Support Vector Machines." In: *IEEE Geoscience and Remote Sensing Letters* 4.4 (Oct. 2007), pp. 674–677. ISSN: 1558-0571. DOI: 10.1109/LGRS.2007.905116.

[16] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations." In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 121–135. ISBN: 978-3-642-16145-2. DOI: 10.1007/978-3-642-16145-2_9.

[17] Martin Arlitt and Tai Jin. "A Workload Characterization Study of the 1998 World Cup Web Site." In: *IEEE Network* 14.3 (May 2000), pp. 30–37. ISSN: 1558-156X. DOI: 10.1109/65.844498.

[18] Uwe Aßmann, Nelly Bencomo, Betty H. C. Cheng, and Robert B. France. "Models@run.Time (Dagstuhl Seminar 11481)." In: *Dagstuhl Reports* 1.11 (2012). Ed. by Uwe Aßmann, Nelly Bencomo, Betty H. C. Cheng, and Robert B. France, pp. 91–123. ISSN: 2192-5283. DOI: 10.4230/DagRep.1.11.91.

[19] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Apr. 12, 2010. ISBN: 978-1-4008-2873-9. DOI: `10.1515/9781400828739`.

[20] Karl Johan Åström and Björn Wittenmark. *Adaptive Control (2nd Ed.)* Reprint of the Addison-Wesley, Reading Massachusetts. Courier Corporation, 2013. ISBN: 0-486-31914-8.

[21] Colin Atkinson and Thomas Kuhne. "Model-Driven Development: A Metamodeling Foundation." In: *IEEE Software* 20.5 (Sept. 2003), pp. 36–41. ISSN: 1937-4194. DOI: `10.1109/MS.2003.1231149`.

[22] Özalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad Moorsel, and Maarten van Steen, eds. *Self-Star Properties in Complex Information Systems*. Vol. 3460. Lecture Notes in Computer Science, Hot Topics. 2005. URL: `https://doi.org/10.1007/b136551`.

[23] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Paola Simeoni. "Model-Based Performance Prediction in Software Development: A Survey." In: *IEEE Transactions on Software Engineering* 30.5 (May 2004), pp. 295–310. ISSN: 1939-3520. DOI: `10.1109/TSE.2004.9`.

[24] Carlos A. Bana e Costa and Jean-Claude Vansnick. "MACBETH — An Interactive Path towards the Construction of Cardinal Value Functions." In: *International Transactions in Operational Research* 1.4 (Oct. 1, 1994), pp. 489–500. ISSN: 0969-6016. DOI: `10.1016/0969-6016(94)90010-8`. (Visited on 02/16/2022).

[25] Franck Barbier, Eric Cariou, Olivier Le Goaer, and Samson Pierre. "Software Adaptation: Classification and a Case Study with State Chart XML." In: *IEEE Software* 32.5 (Sept. 2015), pp. 68–76. ISSN: 1937-4194. DOI: `10.1109/MS.2014.130`.

[26] Luciano Baresi and Carlo Ghezzi. "The Disappearing Boundary between Development-Time and Run-Time." In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA). FoSER '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 17–22. ISBN: 978-1-4503-0427-6. DOI: `10.1145/1882362.1882367`.

[27] Luciano Baresi and Reiko Heckel. "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective." In: *Graph Transformation*. Ed. by Andrea Corradini, Hartmut Ehrig, Hans -Jörg Kreowski, and Grzegorz Rozenberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 402–429. ISBN: 978-3-540-45832-6. DOI: `10.1007/3-540-45832-8_30`.

[28] André Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev. "Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field." In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (Apr. 2019), pp. 800–813. ISSN: 1558-2183. DOI: `10.1109/TPDS.2018.2870389`.

[29] André Bauer, Marwin Züfle, Nikolas Herbst, Albin Zehe, Andreas Hotho, and Samuel Kounev. "Time Series Forecasting for Self-Aware Systems." In: *Proceedings of the IEEE* 108.7 (July 2020), pp. 1068–1093. ISSN: 1558-2256. DOI: `10.1109/JPROC.2020.2983857`.

[30] Eric Baum and Frank Wilczek. "Supervised Learning of Probability Distributions by Neural Networks." In: *Neural Information Processing Systems*. Ed. by D. Anderson. American Institute of Physics, 1987. URL: `https://proceedings.neurips.cc/paper/1987/file/eccbc87e4b5ce2fe28308fd9f2a7baf3-Paper.pdf`.

[31] Mark Bearden, Sachin Garg, and Woei-jyh Lee. "Integrating Goal Specification in Policy-Based Management." In: *Policies for Distributed Systems and Networks*. Ed. by Morris Sloman, Emil C. Lupu, and Jorge Lobo. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 153–170. ISBN: 978-3-540-44569-2. DOI: `10.1007/3-540-44569-2_10`.

[32] Basil Becker and Holger Giese. "Modeling of Correct Self-Adaptive Systems: A Graph Transformation System Based Approach." In: *Proceedings of the 5th International Conference on Soft Computing as Transdisciplinary Science and Technology*. CSTST '08. New York, NY, USA: Association for Computing Machinery, Oct. 28, 2008, pp. 508–516. ISBN: 978-1-60558-046-3. DOI: `10.1145/1456223.1456326`. (Visited on 11/19/2021).

[33] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "Model-Based Performance Prediction with the Palladio Component Model." In: *Proceedings of the 6th International Workshop on Software and Performance*. WOSP '07. New York, NY, USA: Association for Computing Machinery, Feb. 5, 2007, pp. 54–65. ISBN: 978-1-59593-297-6. DOI: `10.1145/1216993.1217006`. (Visited on 10/17/2022).

[34] Nelly Bencomo. "On the Use of Software Models during Software Execution." In: *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*. MISE '09. USA: IEEE Computer Society, May 17, 2009, pp. 62–67. ISBN: 978-1-4244-3722-1. DOI: `10.1109/MISE.2009.5069899`. (Visited on 11/11/2021).

[35] Nelly Bencomo and Gordon Blair. "Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 183–200. ISBN: 978-3-642-02161-9. DOI: `10.1007/978-3-642-02161-9_10`. (Visited on 10/21/2022).

[36] Nelly Bencomo, Sebastian Götz, and Hui Song. "Models@run.Time: A Guided Tour of the State of the Art and Research Challenges." In: *Softw Syst Model* 18.5 (Oct. 1, 2019), pp. 3049–3082. ISSN: 1619-1374. DOI: `10.1007/s10270-018-00712-x`. (Visited on 11/11/2021).

[37] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. "Genie: Supporting the Model Driven Development of Reflective, Component-Based Adaptive Systems." In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. New York, NY, USA: Association for Computing Machinery, May 10, 2008, pp. 811–814. ISBN: 978-1-60558-079-1. DOI: `10.1145/1368088.1368207`. (Visited on 11/11/2021).

[38] Nelly Bencomo, Jon Whittle, Pete Sawyer, Anthony Finkelstein, and Emmanuel Letier. "Requirements Reflection: Requirements as Runtime Entities." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. New York, NY, USA: Association for Computing Machinery, May 1, 2010, pp. 199–202. ISBN: 978-1-60558-719-6. DOI: `10.1145/1810295.1810329`. (Visited on 11/11/2021).

[39] Mohamed N. Bennani and Daniel A. Menasce. "Resource Allocation for Autonomic Data Centers Using Analytic Performance Models." In: *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, June 2005, pp. 229–240. DOI: 10.1109/ICAC.2005.50.

[40] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. "Incremental Evaluation of Model Queries over EMF Models." In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 76–90. ISBN: 978-3-642-16145-2. DOI: 10.1007/978-3-642-16145-2_6.

[41] Jean Bézivin. "On the Unification Power of Models." In: *Software & Systems Modeling* 4.2 (May 1, 2005), pp. 171–188. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0079-0.

[42] Jean Bézivin. "Model Driven Engineering: An Emerging Technical Space." In: *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 36–64. ISBN: 978-3-540-46235-4. DOI: 10.1007/11877028_2. (Visited on 10/27/2021).

[43] Jean Bézivin, Sébastien Gérard, Pierre-Alain Muller, and Laurent Rioux. "MDA Components: Challenges and Opportunities." In: *Workshop on Metamodelling for MDA*. York, England, United Kingdom, 2003, pp. 23–41. URL: https://hal.archives-ouvertes.fr/hal-00448057.

[44] Jean Bézivin, Richard F. Paige, Uwe Aßmann, Bernhard Rumpe, and Douglas C. Schmidt. "08331 Manifesto – Model Engineering for Complex Systems." In: *Perspectives Workshop: Model Engineering of Complex Systems (MECS)*. Ed. by Uwe Aßmann, Jean Bézivin, Richard Paige, Bernhard Rumpe, and Douglas C. Schmidt. Vol. 8331. Dagstuhl Seminar Proceedings (DagSemProc). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2008, pp. 1–4. DOI: 10.4230/DagSemProc.08331.2. (Visited on 03/14/2023).

[45] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. "Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework." In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'06. Berlin, Heidelberg: Springer-Verlag, Oct. 1, 2006, pp. 425–439. ISBN: 978-3-540-45772-5. DOI: 10.1007/11880240_30. (Visited on 04/07/2022).

[46] George David Birkhoff. *Dynamical Systems*. Vol. 9. Bibliolife DBA of Bibilio Bazaar II LLC, 2015, 1927. ISBN: 1-296-02664-7.

[47] Gordon Blair, Nelly Bencomo, and Robert B. France. "Models@ Run.Time." In: *Computer* 42.10 (Oct. 2009), pp. 22–27. ISSN: 1558-0814. DOI: 10.1109/MC.2009.326.

[48] Han Bleichrodt, Jason N. Doctor, Martin Filko, and Peter P. Wakker. "Utility Independence of Multiattribute Utility Theory Is Equivalent to Standard Sequence Invariance of Conjoint Measurement." In: *Journal of Mathematical Psychology* 55.6

(Dec. 1, 2011), pp. 451–456. ISSN: 0022-2496. DOI: 10.1016/j.jmp.2011.08.001. (Visited on 02/18/2022).

[49] Anton A. Bougaev. "Pattern Recognition Based Tools Enabling Autonomic Computing." In: *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, June 2005, pp. 313–314. DOI: 10.1109/ICAC.2005.45.

[50] Craig Boutilier. "A POMDP Formulation of Preference Elicitation Problems." In: *Eighteenth National Conference on Artificial Intelligence*. USA: American Association for Artificial Intelligence, July 28, 2002, pp. 239–246. ISBN: 978-0-262-51129-2.

[51] Craig Boutilier, Rajarshi Das, Jeffrey O. Kephart, Gerald Tesauro, and William E. Walsh. "Cooperative Negotiation in Autonomic Systems Using Incremental Utility Elicitation." In: *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*. UAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 7, 2002, pp. 89–97. ISBN: 978-0-12-705664-7.

[52] Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. "Constraint-Based Optimization and Utility Elicitation Using the Minimax Decision Criterion." In: *Artificial Intelligence* 170.8 (June 1, 2006), pp. 686–713. ISSN: 0004-3702. DOI: 10.1016/j.artint.2006.02.003. (Visited on 02/16/2022).

[53] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. "A Survey of Self-Management in Dynamic Software Architecture Specifications." In: *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*. WOSS '04. New York, NY, USA: Association for Computing Machinery, Oct. 31, 2004, pp. 28–33. ISBN: 978-1-58113-989-1. DOI: 10.1145/1075405.1075411. (Visited on 11/12/2021).

[54] Richard Bradley. *Decision Theory with a Human Face*. Cambridge: Cambridge University Press, 2017. ISBN: 978-1-107-00321-7. DOI: 10.1017/9780511760105.

[55] Leo Breiman. "Random Forests." In: *Machine Learning* 45.1 (Oct. 1, 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. (Visited on 03/23/2022).

[56] Leo Breiman and Richard A. Olshen. *Classification and Regression Trees*. !st. Jan. 1, 1984. ISBN: 978-1-315-13947-0.

[57] Frederick P. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering." In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 1558-0814. DOI: 10.1109/MC.1987.1663532.

[58] Alan W. Brown. "Model Driven Architecture: Principles and Practice." In: *Softw Syst Model* 3.4 (Dec. 1, 2004), pp. 314–327. ISSN: 1619-1374. DOI: 10.1007/s10270-004-0061-2. (Visited on 10/18/2022).

[59] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit. "A Design Space for Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 33–50. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_2.

[60] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. "Engineering Self-Adaptive Systems through Feedback Loops." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70. ISBN: 978-3-642-02161-9. DOI: `10.1007/978-3-642-02161-9_3`.

[61] Robert Burbidge, Matthew Trotter, Bernard Buxton, and SI Holden. "Drug Design by Machine Learning: Support Vector Machines for Pharmaceutical Data Analysis." In: *Computers & Chemistry* 26.1 (Dec. 1, 2001), pp. 5–14. ISSN: 0097-8485. DOI: `10.1016/S0097-8485(01)00094-8`. (Visited on 10/10/2022).

[62] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. "Hyper-Heuristics: A Survey of the State of the Art." In: *Journal of the Operational Research Society* 64.12 (Dec. 1, 2013), pp. 1695–1724. ISSN: 0160-5682. DOI: `10.1057/jors.2013.71`. (Visited on 08/31/2022).

[63] Edmund K Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. "Hyper-Heuristics: An Emerging Direction in Modern Search Technology." In: *Handbook of Metaheuristics*. Ed. by Fred Glover and Gary A. Kochenberger. International Series in Operations Research & Management Science. Boston, MA: Springer US, 2003, pp. 457–474. ISBN: 978-0-306-48056-0. DOI: `10.1007/0-306-48056-5_16`. (Visited on 05/16/2022).

[64] Michael Buro. "Improving Heuristic Mini-Max Search by Supervised Learning." In: *Artificial Intelligence* 134.1 (Jan. 1, 2002), pp. 85–99. ISSN: 0004-3702. DOI: `10.1016/S0004-3702(01)00093-5`. (Visited on 02/18/2022).

[65] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. 476 pp. ISBN: 978-0-471-95869-7.

[66] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility." In: *Future Generation Computer Systems* 25.6 (June 1, 2009), pp. 599–616. ISSN: 0167-739X. DOI: `10.1016/j.future.2008.12.001`. (Visited on 08/16/2022).

[67] Ricardo Diniz Caldas, Arthur Rodrigues, Eric Bernd Gil, Genaína Nunes Rodrigues, Thomas Vogel, and Patrizio Pelliccione. "A Hybrid Approach Combining Control Theory and AI for Engineering Self-Adaptive Systems." In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: Association for Computing Machinery, June 29, 2020, pp. 9–19. ISBN: 978-1-4503-7962-5. URL: `https://doi.org/10.1145/3387939.3391595` (visited on 01/25/2022).

[68] Eduardo F Camacho and Carlos Bordons Alba. *Model Predictive Control*. Springer science & business media, 2013. ISBN: 1-85233-694-3.

[69] Javier Cámara and Rogério de Lemos. “Evaluation of Resilience in Self-Adaptive Systems Using Probabilistic Model-Checking.” In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. June 2012, pp. 53–62. DOI: 10.1109/SEAMS.2012.6224391.

[70] Javier Cámara, David Garlan, Bradley Schmerl, and Ashutosh Pandey. “Optimal Planning for Architecture-Based Self-Adaptation via Model Checking of Stochastic Games.” In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (New York, NY, USA). SAC ’15. ACM, 2015, pp. 428–435. URL: http://doi.acm.org/10.1145/2695664.2695680.

[71] Javier Cámara, Antónia Lopes, David Garlan, and Bradley Schmerl. “Adaptation Impact and Environment Models for Architecture-Based Self-Adaptive Systems.” In: *Science of Computer Programming*. Special Issue of the 11th International Symposium on Formal Aspects of Component Software 127 (Oct. 1, 2016), pp. 50–75. ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.12.006. (Visited on 09/27/2022).

[72] Javier Cámara, Bradley Schmerl, Gabriel A. Moreno, and David Garlan. “MOSAICO: Offline Synthesis of Adaptation Strategy Repertoires with Flexible Trade-Offs.” In: *Automated Software Engg.* 25.3 (Sept. 2018), pp. 595–626. ISSN: 0928-8910. DOI: 10.1007/s10515-018-0234-9.

[73] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. “Model-Based System Reconfiguration for Dynamic Performance Management.” In: *Journal of Systems and Software*. Software Performance 80.4 (Apr. 1, 2007), pp. 455–473. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.07.039. (Visited on 11/12/2021).

[74] Rich Caruana and Alexandru Niculescu-Mizil. “An Empirical Comparison of Supervised Learning Algorithms.” In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML ’06. New York, NY, USA: Association for Computing Machinery, June 25, 2006, pp. 161–168. ISBN: 978-1-59593-383-6. DOI: 10.1145/1143844.1143865. (Visited on 03/26/2022).

[75] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. “Self-Healing by Means of Automatic Workarounds.” In: *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems* (New York, NY, USA). SEAMS ’08. ACM, 2008, pp. 17–24. URL: http://doi.acm.org/10.1145/1370018.1370023.

[76] Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu. “Diagnosing Architectural Run-Time Failures.” In: *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2013, pp. 103–112. DOI: 10.1109/SEAMS.2013.6595497.

[77] X. Castillo, S. R. McConnel, and D. P. Siewiorek. “Derivation and Calibration of a Transient Error Reliability Model.” In: *IEEE Transactions on Computers* 31.07 (July 1, 1982), pp. 658–671. ISSN: 0018-9340. DOI: 10.1109/TC.1982.1676063. (Visited on 03/14/2023).

[78] Urszula Chajewska, Lise Getoor, Joseph Norman, and Yuval Shahar. "Utility Elicitation as a Classification Problem." In: *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*. UAI'98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., July 24, 1998, pp. 79–88. ISBN: 978-1-55860-555-8.

[79] Urszula Chajewska, Daphne Koller, and Ronald Parr. "Making Rational Decisions Using Adaptive Utility Elicitation." In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, July 30, 2000, pp. 363–369. ISBN: 978-0-262-51112-4.

[80] Kitty S Chan and Judith Bishop. "The Design of a Self-Healing Composition Cycle for Web Services." In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. May 2009, pp. 20–27. DOI: 10.1109/SEAMS.2009.5069070.

[81] Herve Chang, Leonardo Mariani, and Mauro Pezzè. "Self-Healing Strategies for Component Integration Faults." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*. Sept. 2008, pp. 25–32. DOI: 10.1109/ASEW.2008.4686290.

[82] Olivier Chapelle, Donald Metlzer, Ya Zhang, and Pierre Grinspan. "Expected Reciprocal Rank for Graded Relevance." In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. CIKM '09. New York, NY, USA: Association for Computing Machinery, Nov. 2, 2009, pp. 621–630. ISBN: 978-1-60558-512-3. DOI: 10.1145/1645953.1646033. (Visited on 03/31/2022).

[83] Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh, and Wenyun Zhao. "Self-Adaptation through Incremental Generative Model Transformations at Runtime." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 31, 2014, pp. 676–687. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568310. (Visited on 09/12/2022).

[84] DeJiu Chen and Zhonghai Lu. "A Model-Based Approach to Dynamic Self-assessment for Automated Performance and Safety Awareness of Cyber-Physical Systems." In: *Model-Based Safety and Assessment*. Ed. by Marco Bozzano and Yiannis Papadopoulos. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 227–240. ISBN: 978-3-319-64119-5. DOI: 10.1007/978-3-319-64119-5_15.

[85] Tao Chen and Rami Bahsoon. "Self-Adaptive and Online QoS Modeling for Cloud-Based Software Services." In: *IEEE Transactions on Software Engineering* 43.5 (May 2017), pp. 453–475. ISSN: 1939-3520. DOI: 10.1109/TSE.2016.2608826.

[86] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA: Association for Computing Machinery, Aug. 13, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. (Visited on 03/23/2022).

[87] Xue-wen Chen. "An Improved Branch and Bound Algorithm for Feature Selection." In: *Pattern Recogn. Lett.* 24.12 (Aug. 1, 2003), pp. 1925–1933. ISSN: 0167-8655. DOI: 10.1016/S0167-8655(03)00020-5. (Visited on 03/28/2022).

[88] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty." In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 468–483. ISBN: 978-3-642-04425-0. DOI: 10.1007/978-3-642-04425-0_36.

[89] Betty H.C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogério de Lemos, eds. *Software Engineering for Self-Adaptive Systems*. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer, 2009. URL: http://dx.doi.org/10.1007/978-3-642-02161-9.

[90] Shang-Wen Cheng. "Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation." PhD thesis. Pittsburgh, USA: School of Computer Science, Carnegie Mellon University, 2008.

[91] Shang-Wen Cheng and David Garlan. "Stitch: A Language for Architecture-Based Self-Adaptation." In: *Journal of Systems and Software* 85.12 (2012), pp. 2860–2875. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.02.060.

[92] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. "Making Self-Adaptation an Engineering Reality." In: *Self-Star Properties in Complex Information Systems*. Ed. by Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 158–173. ISBN: 978-3-540-32013-5. DOI: 10.1007/11428589_11.

[93] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. "Architecture-Based Self-Adaptation in the Presence of Multiple Objectives." In: *Proceedings of the 2006 International Workshop on Self-adaptation and Self-Managing Systems*. SEAMS '06. New York, NY, USA: Association for Computing Machinery, May 21, 2006, pp. 2–8. ISBN: 978-1-59593-403-1. DOI: 10.1145/1137677.1137679. (Visited on 09/16/2021).

[94] Michael R. Chernick and Robert A. LaBudde. *An Introduction to Bootstrap Methods with Applications to R*. 1st. Wiley Publishing, 2011. 240 pp. ISBN: 978-0-470-46704-6.

[95] David Christensen. "Fast Algorithms for the Calculation of Kendall's Tau." In: *Computational Statistics* 20.1 (Mar. 1, 2005), pp. 51–62. ISSN: 1613-9658. DOI: 10.1007/BF02736122. (Visited on 04/04/2022).

[96] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford. "Documenting Software Architectures: Views and Beyond." In: *25th International Conference on Software Engineering, 2003. Proceedings.* May 2003, pp. 740–741. DOI: 10.1109/ICSE.2003.1201264.

[97] Zack Coker, David Garlan, and Claire Le Goues. "SASS: Self-Adaptation Using Stochastic Search." In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. May 2015, pp. 168–174. DOI: 10.1109/SEAMS.2015.16.

[98] Eric Connally, Andrew M. Gleason, and Deborah Hughes-Hallett. *Functions Modeling Change, Test Bank: A Preparation for Calculus*. John Wiley & Sons, Incorporated, 2003. ISBN: 978-0-471-46827-1.

[99] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Third Edition. Cambridge, MA, USA: The MIT Press, 2009. ISBN: 978-0-262-03384-8.

[100] Carlos E. Cuesta, Pablo de la Fuente, and Manuel Barrio-Solárzano. "Dynamic Coordination Architecture through the Use of Reflection." In: *Proceedings of the 2001 ACM Symposium on Applied Computing*. SAC '01. New York, NY, USA: Association for Computing Machinery, Mar. 1, 2001, pp. 134–140. ISBN: 978-1-58113-287-8. DOI: 10.1145/372202.372298. (Visited on 11/11/2021).

[101] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. "Supervised Learning." In: *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*. Ed. by Matthieu Cord and Pádraig Cunningham. Berlin, Heidelberg: Springer, 2008, pp. 21–49. ISBN: 978-3-540-75171-7. DOI: 10.1007/978-3-540-75171-7_2. (Visited on 03/26/2022).

[102] Mirko D'Angelo, Simos Gerasimou, Sona Ghahremani, Johannes Grohmann, Ingrid Nunes, Evangelos Pournaras, and Sven Tomforde. "On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework." In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2019, pp. 13–24. DOI: 10.1109/SEAMS.2019.00012.

[103] Mirko D'Angelo, Sona Ghahremani, Simos Gerasimou, Johannes Grohmann, Ingrid Nunes, Sven Tomforde, and Evangelos Pournaras. "Learning to Learn in Collective Adaptive Systems: Mining Design Patterns for Data-driven Reasoning." In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). Aug. 2020, pp. 121–126. DOI: 10.1109/ACSOS-C51401.2020.00042.

[104] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. "The Ponder Policy Specification Language." In: *Policies for Distributed Systems and Networks*. Ed. by Morris Sloman, Emil C. Lupu, and Jorge Lobo. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 18–38. ISBN: 978-3-540-44569-2. DOI: 10.1007/3-540-44569-2_2.

[105] Dani Yogatama and Gideon Mann. "Efficient Transfer Learning Method for Automatic Hyperparameter Tuning." In: *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*. Ed. by Samuel Kaski and Jukka Corander. Vol. 33. PMLR, Apr. 2, 2014, pp. 1077–1085. URL: https://proceedings.mlr.press/v33/yogatama14.html.

[106] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. "A Highly-Extensible, XML-based Architecture Description Language." In: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. Proceedings Working IEEE/IFIP Conference on Software Architecture. Aug. 2001, pp. 103–112. DOI: 10.1109/WICSA.2001.948416.

[107] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. "Towards Architecture-Based Self-Healing Systems." In: *Proceedings of the First Workshop on Self-healing Systems*. WOSS '02. New York, NY, USA: Association for Computing Machinery, Nov. 18, 2002, pp. 21–26. ISBN: 978-1-58113-609-8. DOI: 10.1145/582128.582133. (Visited on 11/15/2021).

[108] Pierre-Charles David, Thomas Ledoux, et al. "Safe Dynamic Reconfigurations of Fractal Architectures with Fscript." In: *Proceeding of Fractal CBSE Workshop, ECOOP*. Vol. 6. Citeseer. 2006. URL: https://web.imt-atlantique.fr/x-info/ledoux/Publis/ws-fractal-ecoop06.pdf.

[109] Morris H DeGroot. *Optimal Statistical Decisions*. John Wiley & Sons, 2005. ISBN: 978-0-471-68029-1.

[110] Rogerio de Lemos et al. "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Rogerio de Lemos, Holger Giese, Hausi Müller, and Mary Shaw. Vol. 10431. Dagstuhl Seminar Proceedings (DagSemProc). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. DOI: 10.4230/DagSemProc.10431.3.

[111] Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. "Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511)." In: *Dagstuhl Reports* 3.12 (2014). Ed. by Rogerio de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, pp. 67–96. ISSN: 2192-5283. DOI: 10.4230/DagRep.3.12.67. (Visited on 03/16/2023).

[112] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–32. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_1.

[113] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances." In: *Software Engineering for Self-Adaptive Systems III. Assurances* (Cham). Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Springer International Publishing, 2017, pp. 3–30. ISBN: 978-3-319-74183-3.

[114] Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. "Modeling Features at Runtime." In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 138–152. ISBN: 978-3-642-16129-2. DOI: 10.1007/978-3-642-16129-2_11.

[115] Mahdi Derakhshanmanesh, Mehdi Amoui, Greg O'Grady, Jürgen Ebert, and Ladan Tahvildari. "GRAF: Graph-Based Runtime Adaptation Framework." In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. New York, NY, USA: Association for Computing Machinery, May 23, 2011, pp. 128–137. ISBN: 978-1-4503-0575-4. DOI: 10.1145/1988008.1988026. (Visited on 11/16/2021).

[116] Antinisca Di Marco, Paola Inverardi, and Romina Spalazzese. "Synthesizing Self-Adaptive Connectors Meeting Functional and Performance Concerns." In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Piscataway, NJ, USA). SEAMS '13. IEEE Press, 2013, pp. 133–142. URL: http://dl.acm.org/citation.cfm?id=2487336.2487358.

[117] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. "A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications." In: *Autom Softw Eng* 15.3 (Dec. 1, 2008), pp. 313–341. ISSN: 1573-7535. DOI: 10.1007/s10515-008-0032-x. (Visited on 01/27/2022).

[118] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. "A Survey of Autonomic Communications." In: *ACM Trans. Auton. Adapt. Syst.* 1.2 (Dec. 2006), pp. 223–259. ISSN: 1556-4665. DOI: 10.1145/1186778.1186782.

[119] Gabriel Dulac-Arnold, Nir Levine, Daniel J. Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. "Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks and Analysis." In: *Mach Learn* 110.9 (Sept. 1, 2021), pp. 2419–2468. ISSN: 1573-0565. DOI: 10.1007/s10994-021-05961-4. (Visited on 02/18/2022).

[120] Subhasri Duttagupta, Rupinder Virk, and Manoj Nambiar. "Predicting Performance in the Presence of Software and Hardware Resource Bottlenecks." In: *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2014)*. International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2014). July 2014, pp. 542–549. DOI: 10.1109/SPECTS.2014.6879991.

[121] Jack Edmonds. "Matroids and the Greedy Algorithm." In: *Mathematical programming* 1.1 (1971), pp. 127–136. DOI: 10.1007/BF01584082.

[122] Bradley Efron and Robert J Tibshirani. *An Introduction to the Bootstrap*. CRC press, 1994. ISBN: 978-0-412-04231-7.

[123] Jens Ehlers, Andre van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Self-Adaptive Software System Monitoring for Performance Anomaly Localization." In: *Proceedings of the 8th ACM International Conference on Autonomic Computing* (New York, NY, USA). ICAC '11. ACM, 2011, pp. 197–200. URL: http://doi.acm.org/10.1145/1998582.1998628.

[124] H Ehrig, H-J Kreowski, U Montanari, and G Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 3: Concurrency, Parallelism, and Distribution*. WORLD SCIENTIFIC, Aug. 1999. ISBN: 978-981-02-4021-9. DOI: 10.1142/4181.

[125] Ibrahim Elgendi, Md. Farhad Hossain, Abbas Jamalipour, and Kumudu S. Munasinghe. "Protecting Cyber Physical Systems Using a Learned MAPE-K Model." In: *IEEE Access* 7 (2019), pp. 90954–90963. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927037.

[126] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. "FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems." In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)* (New York, NY, USA). ACM, 2010, pp. 7–16. URL: `http://doi.acm.org/10.1145/1882291.1882296`.

[127] Edward Elliott. "Normative Decision Theory." In: *Analysis* 79.4 (Oct. 1, 2019), pp. 755–772. ISSN: 0003-2638. DOI: `10.1093/analys/anz059`. (Visited on 02/16/2022).

[128] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Service Technology. Upper Saddle River: Pearson Education Incorporated, 2016. ISBN: 0-13-452445-4.

[129] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. "A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems." In: *IEEE Transactions on Software Engineering* 39.11 (Nov. 1, 2013), pp. 1467–1493. ISSN: 0098-5589. DOI: `10.1109/TSE.2013.37`. (Visited on 03/14/2023).

[130] Gabriele Fanelli, Matthias Dantone, Juergen Gall, Andrea Fossati, and Luc Van Gool. "Random Forests for Real Time 3D Face Analysis." In: *Int J Comput Vis* 101.3 (Feb. 1, 2013), pp. 437–458. ISSN: 1573-1405. DOI: `10.1007/s11263-012-0549-0`. (Visited on 03/30/2022).

[131] Eugene A Feinberg and Adam Shwartz. *Handbook of Markov Decision Processes: Methods and Applications*. Vol. 40. Springer Science & Business Media, 2012.

[132] Gabriela Félix Solano, Ricardo Diniz Caldas, Genaína Nunes Rodrigues, Thomas Vogel, and Patrizio Pelliccione. "Taming Uncertainty in the Assurance Process of Self-Adaptive Systems: A Goal-Oriented Approach." In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2019, pp. 89–99. DOI: `10.1109/SEAMS.2019.00020`.

[133] Ji Feng, Yang Yu, and Zhi-Hua Zhou. "Multi-Layered Gradient Boosting Decision Trees." In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., Dec. 3, 2018, pp. 3555–3565. URL: `https://dl.acm.org/doi/10.5555/3327144.3327273`.

[134] Artur J. Ferreira and Mário A. T. Figueiredo. "Efficient Feature Selection Filters for High-Dimensional Data." In: *Pattern Recognition Letters* 33.13 (Oct. 1, 2012), pp. 1794–1804. ISSN: 0167-8655. DOI: `10.1016/j.patrec.2012.05.019`. (Visited on 03/28/2022).

[135] Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Jean-Yves Tigli, and Michel Riveill. "Models at Runtime: Service for Device Composition and Adaptation." In: *4th International Workshop Models@run.Time*. IEEE Computer Society and ACM. Denver, United States, Oct. 2009. URL: `https://hal.archives-ouvertes.fr/hal-00481778`.

[136] Roberto Rodrigues Filho and Barry Porter. "Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning." In: *ACM Trans. Auton. Adapt. Syst.* 12.3 (Sept. 20, 2017), 16:1–16:25. ISSN: 1556-4665. DOI: `10.1145/3092691`. (Visited on 10/10/2022).

[137] Antonio Filieri, Carlo Ghezzi, Alberto Leva, and Martina Maggio. "Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements." In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). Nov. 2011, pp. 283–292. DOI: 10.1109/ASE.2011.6100064.

[138] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated Multi-Objective Control for Self-Adaptive Software Design." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, Aug. 30, 2015, pp. 13–24. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786833. (Visited on 10/02/2022).

[139] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. "Supporting Self-Adaptation via Quantitative Verification and Sensitivity Analysis at Run Time." In: *IEEE Transactions on Software Engineering* 42.1 (Jan. 2016), pp. 75–99. ISSN: 1939-3520. DOI: 10.1109/TSE.2015.2421318.

[140] Antonio Filieri et al. "Software Engineering Meets Control Theory." In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. May 2015, pp. 71–82. DOI: 10.1109/SEAMS.2015.12.

[141] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. "Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java." In: *Theory and Application of Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 296–309. ISBN: 978-3-540-46464-8. DOI: 10.1007/978-3-540-46464-8_21.

[142] Peter C Fishburn. *Utility Theory for Decision Making*. Publications in Operations Research. McLean VA: Wiley, 1970. ISBN: 0-471-26060-6.

[143] Camilo Fitzgerald, Benjamin Klöpper, and Shinichi Honiden. "Utility-Based Self-Adaption with Environment Specific Quality Models." In: *Adaptive and Intelligent Systems*. Ed. by Abdelhamid Bouchachia. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 107–118. ISBN: 978-3-642-23857-4. DOI: 10.1007/978-3-642-23857-4_14.

[144] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. "Modeling and Validating Dynamic Adaptation." In: *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*. Ed. by Michel Chaudron. Vol. 5421. LNCS. Springer-Verlag, 2009, pp. 97–108. ISBN: 978-3-642-01648-6.

[145] Franck Fleurey and Arnor Solberg. "A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems." In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 606–621. ISBN: 978-3-642-04425-0. DOI: 10.1007/978-3-642-04425-0_47.

[146] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. "Using Architecture Models for Runtime Adaptability." In: *IEEE Software* 23.2 (Mar. 2006), pp. 62–70. ISSN: 1937-4194. DOI: 10.1109/MS.2006.61.

[147] Francois Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noel Plouzeau, and Jean-Marc Jezequel. "A Dynamic Component Model for Cyber Physical Systems." In: *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*. CBSE '12. New York, NY, USA: Association for Computing Machinery, June 25, 2012, pp. 135–144. ISBN: 978-1-4503-1345-2. DOI: 10.1145/2304736.2304759. (Visited on 11/11/2021).

[148] Robert France and Bernhard Rumpe. "Model-Driven Development of Complex Software: A Research Roadmap." In: *Future of Software Engineering (FOSE '07)*. Future of Software Engineering (FOSE '07). May 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[149] João M. Franco, Francisco Correia, Raul Barbosa, Mário Zenha-Rela, Bradley Schmerl, and David Garlan. "Improving Self-Adaptation Planning through Software Architecture-Based Stochastic Modeling." In: *Journal of Systems and Software* 115 (May 1, 2016), pp. 42–60. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.01.026. (Visited on 10/05/2022).

[150] David A. Freedman. *Statistical Models: Theory and Practice*. Cambridge University Press, Apr. 27, 2009. 459 pp. ISBN: 978-1-139-47731-4. Google Books: fW_9BV5Wpf8C.

[151] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. "The Failure Detector Abstraction." In: *ACM Comput. Surv.* 43.2 (Feb. 4, 2011), 9:1–9:40. ISSN: 0360-0300. DOI: 10.1145/1883612.1883616. (Visited on 01/19/2022).

[152] Simon French. *Decision Theory: An Introduction to the Mathematics of Rationality*. USA: Halsted Press, 1986. 448 pp. ISBN: 978-0-470-20308-8.

[153] Jerome H. Friedman. "Stochastic Gradient Boosting." In: *Computational Statistics & Data Analysis*. Nonlinear Methods and Data Mining 38.4 (Feb. 28, 2002), pp. 367–378. ISSN: 0167-9473. DOI: 10.1016/S0167-9473(01)00065-2. (Visited on 03/23/2022).

[154] Thomas Gabor, Lenz Belzner, Thomy Phan, and Kyrill Schmid. "Preparing for the Unexpected: Diversity Improves Planning Resilience in Evolutionary Algorithms." In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*. Sept. 2018, pp. 131–140. DOI: 10.1109/ICAC.2018.00023.

[155] Matthieu Gallet, Nezih Yigitbasi, Bahman Javadi, Derrick Kondo, Alexandru Iosup, and Dick Epema. "A Model for Space-Correlated Failures in Large-Scale Distributed Systems." In: *Euro-Par 2010 - Parallel Processing*. Ed. by Pasqua D'Ambra, Mario Guarracino, and Domenico Talia. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 88–100. ISBN: 978-3-642-15277-1. DOI: 10.1007/978-3-642-15277-1_10.

[156] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. "Modeling the Impact of Workload on Cloud Resource Scaling." In: *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. 2014 IEEE 26th International Symposium on Computer Architecture and

High Performance Computing. Oct. 2014, pp. 310–317. DOI: 10.1109/SBAC-PAD.2014.16.

[157] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers." In: *ACM Trans. Comput. Syst.* 30.4 (Nov. 1, 2012), 14:1–14:26. ISSN: 0734-2071. DOI: 10.1145/2382553.2382556. (Visited on 08/18/2022).

[158] Alan G. Ganek and Thomas A. Corbi. "The Dawning of the Autonomic Computing Era." In: *IBM Systems Journal* 42.1 (2003), pp. 5–18. ISSN: 0018-8670. DOI: 10.1147/sj.421.0005.

[159] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed. Pearson Deutschland, 2013. 1140 pp. ISBN: 978-1-292-02447-9.

[160] Vijay K. Garg and J. Roger Mitchell. "Implementable Failure Detectors in Asynchronous Systems." In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by Vikraman Arvind and Sundar Ramanujam. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 158–169. ISBN: 978-3-540-49382-2. DOI: 10.1007/978-3-540-49382-2_14.

[161] David Garlan. "Software Architecture: A Travelogue." In: *Future of Software Engineering Proceedings*. FOSE 2014. New York, NY, USA: Association for Computing Machinery, May 31, 2014, pp. 29–39. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593886. (Visited on 11/12/2021).

[162] David Garlan, Robert Allen, and John Ockerbloom. "Exploiting Style in Architectural Design Environments." In: *SIGSOFT Softw. Eng. Notes* 19.5 (Dec. 1, 1994), pp. 175–188. ISSN: 0163-5948. DOI: 10.1145/195274.195404. (Visited on 11/15/2021).

[163] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure." In: *Computer* 37.10 (Oct. 2004), pp. 46–54. ISSN: 1558-0814. DOI: 10.1109/MC.2004.175.

[164] David Garlan, Robert T Monroe, and David Wile. "Acme: Architectural Description of Component-Based Systems." In: *Foundations of component-based systems* 68 (2000), pp. 47–68. DOI: 10.1184/R1/6602945.v1.

[165] David Garlan and Bradley Schmerl. "Model-Based Adaptation for Self-Healing Systems." In: *Proceedings of the First Workshop on Self-healing Systems*. WOSS '02. New York, NY, USA: Association for Computing Machinery, Nov. 18, 2002, pp. 27–32. ISBN: 978-1-58113-609-8. DOI: 10.1145/582128.582134. (Visited on 09/24/2021).

[166] David Garlan and Bradley Schmerl. "Using Architectural Models at Runtime: Research Challenges." In: *Software Architecture*. Vol. 3047. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, pp. 200–205. DOI: 10.1007/978-3-540-24769-2_15.

[167] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. "Software Architecture-Based Self-Adaptation." In: *Autonomic Computing and Networking*. Ed. by Yanyong Zhang, Laurence Tianruo Yang, and Mieso K. Denko. Springer, 2009, pp. 31–55. URL: http://dx.doi.org/10.1007/978-0-387-89828-5_2.

[168] Kurt Geihs et al. "A Comprehensive Solution for Application-Level Adaptation." In: *Software: Practice and Experience* 39.4 (Mar. 25, 2009), pp. 385–422. ISSN: 0038-0644. DOI: 10.1002/spe.900. (Visited on 11/02/2022).

[169] John C. Georgas, Andre van der Hoek, and Richard N. Taylor. "Using Architectural Models to Manage and Visualize Runtime Adaptation." In: *Computer* 42.10 (Oct. 2009), pp. 52–60. ISSN: 1558-0814. DOI: 10.1109/MC.2009.335.

[170] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. "Self-Organising Software Architectures for Distributed Systems." In: *Proceedings of the First Workshop on Self-healing Systems*. WOSS '02. New York, NY, USA: Association for Computing Machinery, Nov. 18, 2002, pp. 33–38. ISBN: 978-1-58113-609-8. DOI: 10.1145/582128.582135. (Visited on 11/15/2021).

[171] Simos Gerasimou, Radu Calinescu, and Alec Banks. "Efficient Runtime Quantitative Verification Using Caching, Lookahead, and Nearly-Optimal Reconfiguration." In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2014. New York, NY, USA: Association for Computing Machinery, June 2, 2014, pp. 115–124. ISBN: 978-1-4503-2864-7. DOI: 10.1145/2593929.2593932. (Visited on 09/27/2022).

[172] Eva Gerbert-Gaillard and Philippe Lalanda. "Self-Aware Model-Driven Pervasive Systems." In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*. July 2016, pp. 221–222. DOI: 10.1109/ICAC.2016.26.

[173] Alfonso Gerevini and Ivan Serina. "Planning as Propositional CSP: From Walksat to Local Search Techniques for Action Graphs." In: *Constraints* 8.4 (Oct. 1, 2003), pp. 389–413. ISSN: 1572-9354. DOI: 10.1023/A:1025846120461. (Visited on 02/19/2022).

[174] Sona Ghahremani. *Training Datasets of Size 1K, 3K, and 9K for Four Utility Function Variants*. Feb. 2023. DOI: 10.5281/zenodo.7681170. (Visited on 03/14/2023).

[175] Sona Ghahremani, Christian M. Adriano, and Holger Giese. "Training Prediction Models for Rule-Based Self-Adaptive Systems." In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*. Sept. 2018, pp. 187–192. DOI: 10.1109/ICAC.2018.00031.

[176] Sona Ghahremani and Holger Giese. "Performance Evaluation for Self-Healing Systems: Current Practice & Open Issues." In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. June 2019, pp. 116–119. DOI: 10.1109/FAS-W.2019.00039.

[177] Sona Ghahremani and Holger Giese. "Evaluation of Self-Healing Systems: An Analysis of the State-of-the-Art and Required Improvements." In: *Computers* 9.1 (1 Mar. 2020), p. 16. ISSN: 2073-431X. DOI: 10.3390/computers9010016. (Visited on 02/08/2022).

[178] Sona Ghahremani and Holger Giese. "Hybrid Planning with Receding Horizon: A Case for Meta-self-awareness." In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). Sept. 2021, pp. 131–138. DOI: 10.1109/ACSOS-C52956.2021.00045.

[179] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Towards Linking Adaptation Rules to the Utility Function for Dynamic Architectures." In: *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. Sept. 2016, pp. 142–143. DOI: 10.1109/SASO.2016.21.

[180] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures." In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 59–68. DOI: doi:10.1109/ICAC.2017.35.

[181] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Improving Scalability and Reward of Utility-Driven Self-Healing for Large Dynamic Architectures." In: *ACM Trans. Auton. Adapt. Syst.* 14.3 (Feb. 25, 2020), 12:1–12:41. ISSN: 1556-4665. DOI: 10.1145/3380965.

[182] Omid Gheibi, Danny Weyns, and Federico Quin. "Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review." In: *ACM Trans. Auton. Adapt. Syst.* 15.3 (Aug. 18, 2021), 9:1–9:37. ISSN: 1556-4665. DOI: 10.1145/3469440. (Visited on 09/28/2022).

[183] Carlo Ghezzi. "Evolution, Adaptation, and the Quest for Incrementality." In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Ed. by Radu Calinescu and David Garlan. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 369–379. ISBN: 978-3-642-34059-8. DOI: 10.1007/978-3-642-34059-8_19.

[184] Carlo Ghezzi and Amir Molzam Sharifloo. "Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 191–213. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_8. (Visited on 10/02/2022).

[185] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. "Self-Healing Systems — Survey and Synthesis." In: *Decision Support Systems* 42.4 (2007), pp. 2164–2185. ISSN: 0167-9236. DOI: 10.1016/j.dss.2006.06.011.

[186] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. "Graph Transformations for MDE, Adaptation, and Models at Runtime." In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 137–191. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_5. (Visited on 11/15/2021).

[187] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, and Samuel Kounev. "Architectural Concepts for Self-aware Computing Systems." In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Cham: Springer International Publishing, 2017, pp. 109–147. ISBN: 978-3-319-47474-8. DOI: 10.1007/978-3-319-47474-8_5. (Visited on 05/12/2022).

[188] Holger Giese and Robert Wagner. "Incremental Model Synchronization with Triple Graph Grammars." In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 543–557. ISBN: 978-3-540-45773-2. DOI: 10.1007/11880240_38.

[189] Holger Giese and Robert Wagner. "From Model Transformation to Incremental Bidirectional Model Synchronization." In: *Softw Syst Model* 8.1 (Feb. 1, 2009), pp. 21–43. ISSN: 1619-1374. DOI: 10.1007/s10270-008-0089-9. (Visited on 11/15/2021).

[190] Heather J. Goldsby, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Danny Hughes. "Goal-Based Modeling of Dynamically Adaptive System Requirements." In: *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (Ecbs 2008)*. 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (Ecbs 2008). Mar. 2008, pp. 36–45. DOI: 10.1109/ECBS.2008.22.

[191] Ian Gorton. "Software Quality Attributes." In: *Essential Software Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 23–38. ISBN: 978-3-642-19176-3. DOI: 10.1007/978-3-642-19176-3_3.

[192] Donald W. Green and Robert H. Perry. *Perry's Chemical Engineers' Handbook*. 8th Revised edition. New York: McGraw-Hill Education Ltd, Oct. 23, 2007. 2400 pp. ISBN: 978-0-07-142294-9.

[193] Rean Griffith, Gail Kaiser, and Javier Alonso López. "Multi-Perspective Evaluation of Self-Healing Systems Using Simple Probabilistic Models." In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC '09. New York, NY, USA: Association for Computing Machinery, June 15, 2009, pp. 59–60. ISBN: 978-1-60558-564-2. DOI: 10.1145/1555228.1555245.

[194] Xiaodong Gu. "IDES: Self-adaptive Software with Online Policy Evolution Extended from Rainbow." In: *Computer and Information Science 2012*. Ed. by Roger Lee. Studies in Computational Intelligence. Berlin, Heidelberg: Springer, 2012, pp. 181–195. ISBN: 978-3-642-30454-5. DOI: 10.1007/978-3-642-30454-5_13. (Visited on 09/28/2022).

[195] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. "Variability-Aware Performance Prediction: A Statistical Learning Approach." In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). Nov. 2013, pp. 301–311. DOI: 10.1109/ASE.2013.6693089.

[196] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection." In: *J. Mach. Learn. Res.* 3 (Mar. 1, 2003), pp. 1157–1182. ISSN: 1532-4435.

[197] Robrecht Haesevoets, Danny Weyns, Tom Holvoet, and Wouter Joosen. "A Formal Model for Self-Adaptive and Self-Healing Organizations." In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. May 2009, pp. 116–125. DOI: 10.1109/SEAMS.2009.5069080.

[198] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. "Using Product Line Techniques to Build Adaptive Systems." In: *10th International Software Product Line Conference (SPLC'06)*. Aug. 2006. ISBN: 0-7695-2599-7. DOI: 10.1109/SPLINE.2006.1691586.

[199] Deshuai Han, Qiliang Yang, Jianchun Xing, Juelong Li, and Hongda Wang. "FAME: A UML-based Framework for Modeling Fuzzy Self-Adaptive Software." In: *Information and Software Technology* 76 (Aug. 1, 2016), pp. 118–134. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.04.014. (Visited on 10/25/2022).

[200] Jun Han and Claudio Moraga. "The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning." In: *From Natural to Artificial Neural Computation*. Ed. by José Mira and Francisco Sandoval. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 195–201. ISBN: 978-3-540-49288-7. DOI: 10.1007/3-540-59497-3_175.

[201] L.K. Hansen and P. Salamon. "Neural Network Ensembles." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12.10 (Oct. 1990), pp. 993–1001. ISSN: 1939-3539. DOI: 10.1109/34.58871.

[202] Frank E. Harrell. *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Springer Series in Statistics. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-19424-0. DOI: 10.1007/978-3-319-19425-7.

[203] Sara Hassan, Nelly Bencomo, and Rami Bahsoon. "Minimizing Nasty Surprises with Better Informed Decision-Making in Self-Adaptive Systems." In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. May 2015, pp. 134–145. DOI: 10.1109/SEAMS.2015.13.

[204] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. "Linear Methods for Regression." In: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Ed. by Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Springer Series in Statistics. New York, NY: Springer, 2009, pp. 43–99. ISBN: 978-0-387-84858-7. DOI: 10.1007/978-0-387-84858-7_3. (Visited on 03/15/2022).

[205] Tomasz Haupt. "Towards Mediation-Based Self-Healing of Data-Driven Business Processes." In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). June 2012, pp. 139–144. DOI: 10.1109/SEAMS.2012.6224400.

[206] Taliver Heath, Richard P. Martin, and Thu D. Nguyen. "Improving Cluster Availability Using Workstation Validation." In: *SIGMETRICS Perform. Eval. Rev.* 30.1 (June 2002), pp. 217–227. URL: http://doi.acm.org/10.1145/511399.511362.

[207] Robert Heinrich, Eric Schmieders, Reiner Jung, Kiana Rostami, Andreas Metzger, Wilhelm Hasselbring, Ralf Reussner, and Klaus Pohl. "Integrating Run-Time Observations and Design Component Models for Cloud System Analysis." In: *[Paper] In: 9th Workshop on Models@run.Time, September 30, 2014, Valencia, Spain . Proceedings of the 9th Workshop on Models@run.Time ; Pp. 41-46* . 9th Workshop on Models@run.Time. Vol. 1270. Valencia, Spain: CEUR, Sept. 2014, pp. 41–46. URL: http://ceur-ws.org/Vol-1270/ (visited on 11/12/2021).

[208] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN: 978-0-471-26637-2.

[209] Elia Henrichs, Veronika Lesch, Martin Straesser, Samuel Kounev, and Christian Krupitzer. "A Literature Review on Optimization Techniques for Adaptation Planning in Adaptive Systems: State of the Art and Research Directions." In: *Information and Software Technology* 149 (Sept. 1, 2022), p. 106940. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2022.106940. (Visited on 09/27/2022).

[210] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. 228 pp. ISBN: 978-0-262-08213-6.

[211] Paul Horn. "Autonomic Computing: IBM's Perspective on the State of Information Technology." In: *IBM White Paper* (2001).

[212] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. "Meta-Learning in Neural Networks: A Survey." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.9 (Sept. 2022), pp. 5149–5169. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2021.3079209.

[213] Gang Huang, Hong Mei, and Qian-xiang Wang. "Towards Software Architecture at Runtime." In: *SIGSOFT Softw. Eng. Notes* 28.2 (Mar. 1, 2003), p. 8. ISSN: 0163-5948. DOI: 10.1145/638750.638780. (Visited on 11/12/2021).

[214] Markus C. Huebscher and Julie A. McCann. "A Survey of Autonomic Computing—Degrees, Models, and Applications." In: *ACM Comput. Surv.* 40.3 (Aug. 2008). ISSN: 0360-0300. DOI: 10.1145/1380584.1380585.

[215] IBM. "An Architectural Blueprint for Autonomic Computing." In: *IBM White Paper*. Autonomic Computing 31.2006 (2006), pp. 1–6.

[216] IBM. *IBM ILOG CPLEX Optimizer*. Sept. 30, 2021. URL: https://www.ibm.com/analytics/cplex-optimizer (visited on 03/14/2023).

[217] "IEEE Standard for a Software Quality Metrics Methodology." In: *IEEE Std 1061-1992* (Mar. 1993), pp. 1–96. DOI: 10.1109/IEEESTD.1993.115124.

[218] M. Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. "DeltaIoT: A Self-Adaptive Internet of Things Exemplar." In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '17. Buenos Aires, Argentina: IEEE Press, May 20, 2017, pp. 76–82. ISBN: 978-1-5386-1550-8. DOI: 10.1109/SEAMS.2017.21. (Visited on 05/17/2022).

[219] Christian Igel. "Multi-Objective Model Selection for Support Vector Machines." In: *Evolutionary Multi-Criterion Optimization*. Ed. by Carlos A. Coello Coello, Arturo Hernández Aguirre, and Eckart Zitzler. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 534–546. ISBN: 978-3-540-31880-4. DOI: 10.1007/978-3-540-31880-4_37.

[220] Dmitry Ignatov and Andrey Ignatov. "Decision Stream: Cultivating Deep Decision Trees." In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI). Nov. 2017, pp. 905–912. DOI: 10.1109/ICTAI.2017.00140.

[221] Shigeru Imai. "Elastic Cloud Computing for QoS-aware Data Processing." Rensselaer Polytechnic Institute, Troy, NY, 2018.

[222] Alexandru Iosup, Catalin Dumitrescu, Dick Epema, Hui Li, and Lex Wolters. "How Are Real Grids Used? The Analysis of Four Grid Traces and Its Implications." In: *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing* (Washington, DC, USA). GRID '06. IEEE Computer Society, 2006, pp. 262–269. URL: https://doi.org/10.1109/ICGRID.2006.311024.

[223] Alexandru Iosup, Mathieu Jan, Ozan Sonmez, and Dick Epema. *On the Dynamic Resources Availability in Grids*. Research report. INRIA, 2007. URL: https://hal.inria.fr/inria-00143265.

[224] Dennis Ippoliti and Xiaobo Zhou. "A Self-Tuning Self-Optimizing Approach for Automated Network Anomaly Detection Systems." In: *Proceedings of the 9th International Conference on Autonomic Computing* (New York, NY, USA). ICAC '12. ACM, 2012, pp. 85–90. URL: http://doi.acm.org/10.1145/2371536.2371551.

[225] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. "Adaptive Resource Provisioning for Read Intensive Multi-Tier Applications in the Cloud." In: *Future Generation Computer Systems* 27.6 (June 1, 2011), pp. 871–879. ISSN: 0167-739X. DOI: 10.1016/j.future.2010.10.016. (Visited on 05/16/2022).

[226] Mohammad A. Islam, Shaolei Ren, A. Hasan Mahmud, and Gang Quan. "Online Energy Budgeting for Cost Minimization in Virtualized Data Center." In: *IEEE Transactions on Services Computing* 9.3 (May 2016), pp. 421–432. ISSN: 1939-1374. DOI: 10.1109/TSC.2015.2390231.

[227] "ISO/IEC/IEEE International Standard - Systems and Software Engineering – Vocabulary." In: *ISO/IEC/IEEE 24765:2010(E)* (Dec. 2010), pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[228] Ravishankar K. Iyer, S. E. Butner, and E. J. McCluskey. "A Statistical Failure/Load Relationship: Results of a Multicomputer Study." In: *IEEE Transactions on Computers* C-31.7 (July 1982), pp. 697–706.

[229] Paul Jaccard. "The Distribution of the Flora in the Alpine Zone." In: *New Phytologist* 11.2 (1912), pp. 37–50. ISSN: 1469-8137. DOI: 10.1111/j.1469-8137.1912.tb05611.x. (Visited on 03/06/2023).

[230] E. Jacquet-Lagreze and J. Siskos. "Assessing a Set of Additive Utility Functions for Multicriteria Decision-Making, the UTA Method." In: *European Journal of Operational Research* 10.2 (June 1, 1982), pp. 151–164. ISSN: 0377-2217. DOI: 10.1016/0377-2217(82)90155-2. (Visited on 02/16/2022).

[231] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Texts in Statistics. New York, NY: Springer US, 2021. ISBN: 978-1-07-161417-4. DOI: 10.1007/978-1-0716-1418-1. (Visited on 03/14/2023).

[232] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. "Transfer Learning for Improving Model Predictions in Highly Configurable Software." In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2017, pp. 31–41. DOI: 10.1109/SEAMS.2017.11.

[233] Kalervo Järvelin and Jaana Kekäläinen. "Cumulated Gain-Based Evaluation of IR Techniques." In: *ACM Trans. Inf. Syst.* 20.4 (Oct. 1, 2002), pp. 422–446. ISSN: 1046-8188. DOI: 10.1145/582415.582418. (Visited on 03/31/2022).

[234] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Additive Logistic Regression: A Statistical View of Boosting (With Discussion and a Rejoinder by the Authors)." In: *The Annals of Statistics* 28.2 (Apr. 1, 2000), pp. 337–407. DOI: 10.1214/aos/1016218223.

[235] Jerome H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine." In: *The Annals of Statistics* 29.5 (Oct. 1, 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451.

[236] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. "ATL: A Model Transformation Tool." In: *Science of Computer Programming*. Special Issue on Second Issue of Experimental Software and Toolkits (EST) 72.1 (June 1, 2008), pp. 31–39. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.08.002. (Visited on 01/31/2023).

[237] Ari Juels and Martin Wattenberg. "Stochastic Hillclimbing as a Baseline Method for Evaluating Genetic Algorithms." In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky, M. C. Mozer, and M. Hasselmo. Vol. 8. MIT Press, 1995. URL: https://proceedings.neurips.cc/paper/1995/file/36a1694bce9815b7e38a9dad05ad42e0-Paper.pdf.

[238] Elsy Kaddoum, Claudia Raibulet, Jean-Pierre Georgé, Gauthier Picard, and Marie-Pierre Gleizes. "Criteria for the Evaluation of Self-* Systems." In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. New York, NY, USA: Association for Computing Machinery, May 3, 2010, pp. 29–38. ISBN: 978-1-60558-971-8. DOI: 10.1145/1808984.1808988. (Visited on 11/23/2022).

[239] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and Acting in Partially Observable Stochastic Domains." In: *Artificial Intelligence* 101.1 (May 1, 1998), pp. 99–134. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(98)00023-X. (Visited on 10/04/2022).

[240] Abram Kagan and Lawrence A. Shepp. "Why the Variance?" In: *Statistics & Probability Letters* 38.4 (July 1, 1998), pp. 329–333. ISSN: 0167-7152. DOI: 10.1016/S0167-7152(98)00041-8. (Visited on 03/31/2022).

[241] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. "The Architecture Tradeoff Analysis Method." In: *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*. IEEE. 1998, pp. 68–78.

[242] Michael Kearns, Yishay Mansour, and Andrew Ng. "Approximate Planning in Large POMDPs via Reusable Trajectories." In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper/1999/file/4f398cb9d6bc79ae567298335b51ba8a-Paper.pdf.

[243] John Keeney and Vinny Cahill. "Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework." In: *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*. POLICY '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 3–. URL: `http://dl.acm.org/citation.cfm?id=826036.826854`.

[244] John Keeney, Vinny Cahill, and Mads Haahr. "Techniques for Dynamic Adaptation of Mobile Services." In: *The Handbook of Mobile Middleware*. Auerbach Publications, 2006. ISBN: 978-0-429-13308-4.

[245] R. L. Keeney, H. Raiffa, and David W. Rajala. "Decisions with Multiple Objectives: Preferences and Value Trade-Offs." In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.7 (July 1979), pp. 403–403. ISSN: 2168-2909. DOI: `10.1109/TSMC.1979.4310245`.

[246] Ralph L. Keeney. "Utility Independence and Preferences for Multiattributed Consequences." In: *Operations Research* 19.4 (1971), pp. 875–893. ISSN: 0030364X, 15265463. JSTOR: `169048`. URL: `http://www.jstor.org/stable/169048`.

[247] Ralph L. Keeney. "Utility Functions for Multiattributed Consequences." In: *Management Science* 18 (5-part-1 Jan. 1972), pp. 276–287. ISSN: 0025-1909. DOI: `10.1287/mnsc.18.5.276`. (Visited on 02/16/2022).

[248] Ralph L. Keeney. "Multiplicative Utility Functions." In: *Operations Research* 22.1 (1974), pp. 22–34. ISSN: 0030364X, 15265463. JSTOR: `169209`. URL: `http://www.jstor.org/stable/169209`.

[249] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Trade-Offs*. Cambridge: Cambridge University Press, 1993. ISBN: 978-0-521-43883-4. DOI: `10.1017/CBO9781139174084`.

[250] M. G. Kendall. "A New Measure of Rank Correlation." In: *Biometrika* 30.1/2 (1938), pp. 81–93. ISSN: 00063444. DOI: `10.2307/2332226`. JSTOR: `2332226`. (Visited on 04/03/2022).

[251] Stuart Kent. "Model Driven Engineering." In: *Integrated Formal Methods*. Ed. by Michael Butler, Luigia Petre, and Kaisa Sere. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 286–298. ISBN: 978-3-540-47884-3. DOI: `10.1007/3-540-47884-1_16`.

[252] J.O. Kephart and D.M. Chess. "The Vision of Autonomic Computing." In: *Computer* 36.1 (Jan. 2003), pp. 41–50. ISSN: 1558-0814. DOI: `10.1109/MC.2003.1160055`.

[253] J.O. Kephart and W.E. Walsh. "An Artificial Intelligence Perspective on Autonomic Computing Policies." In: *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.* Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004. Yorktown Heights, NY, USA: IEEE, 2004, pp. 3–12. ISBN: 978-0-7695-2141-1. DOI: `10.1109/POLICY.2004.1309145`. (Visited on 09/22/2021).

[254] Jeffrey O. Kephart and Rajarshi Das. "Achieving Self-Management via Utility Functions." In: *IEEE Internet Computing* 11.1 (Jan. 2007), pp. 40–48. ISSN: 1941-0131. DOI: `10.1109/MIC.2007.2`.

[255] Dongsun Kim and Sooyong Park. “Reinforcement Learning-Based Dynamic Adaptation Planning Method for Architecture-Based Self-Managed Software.” In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. May 2009, pp. 76–85. DOI: 10.1109/SEAMS.2009.5069076.

[256] James Kirkpatrick et al. “Overcoming Catastrophic Forgetting in Neural Networks.” In: *Proceedings of the National Academy of Sciences* 114.13 (Mar. 28, 2017), pp. 3521–3526. DOI: 10.1073/pnas.1611835114. (Visited on 03/14/2023).

[257] Kirkpatrick S., Gelatt C. D., and Vecchi M. P. “Optimization by Simulated Annealing.” In: *Science* 220.4598 (May 13, 1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. (Visited on 02/19/2022).

[258] Barbara A. Kitchenham, Guilherme H. Travassos, Anneliese von Mayrhauser, Frank Niessink, Norman F. Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. “Towards an Ontology of Software Maintenance.” In: *Journal of Software Maintenance: Research and Practice* 11.6 (1999), pp. 365–389. DOI: 10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W.

[259] Benjamin Klöpper, Shinichi Honiden, Jan Meyer, and Matthias Tichy. “Planning with Utility and State Trajectory Constraints in Self-Healing Automotive Systems.” In: *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. 2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems. Sept. 2010, pp. 74–83. DOI: 10.1109/SASO.2010.16.

[260] J.C. Knight. “Safety Critical Systems: Challenges and Directions.” In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. May 2002, pp. 547–550.

[261] Mykel J Kochenderfer. *Decision Making under Uncertainty: Theory and Application*. MIT press, 2015.

[262] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning.” In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 282–293. ISBN: 978-3-540-46056-5. DOI: 10.1007/11871842_29.

[263] S. Kodase, Shige Wang, and K.G. Shin. “Transforming Structural Model to Runtime Model of Embedded Software with Real-Time Constraints.” In: *Automation and Test in Europe Conference and Exhibition 2003 Design*. Automation and Test in Europe Conference and Exhibition 2003 Design. Mar. 2003, 170–175 suppl. DOI: 10.1109/DATE.2003.1186690.

[264] Sven Koenig. “Exploring Unknown Environments with Real-Time Search or Reinforcement Learning.” In: *Proceedings of the 11th International Conference on Neural Information Processing Systems*. NIPS’98. Cambridge, MA, USA: MIT Press, Dec. 1, 1998, pp. 1003–1009.

[265] M.M. Kokar, K. Baclawski, and Y.A. Eracar. “Control Theory-Based Foundations of Self-Controlling Software.” In: *IEEE Intelligent Systems and their Applications* 14.3 (May 1999), pp. 37–45. ISSN: 2374-9423. DOI: 10.1109/5254.769883.

[266] Mausam Kolobov and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Cham: Springer International Publishing, 2012. ISBN: 978-3-031-00431-5. DOI: 10.1007/978-3-031-01559-5.

[267] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. "The Case for Reflective Middleware." In: *Commun. ACM* 45.6 (June 1, 2002), pp. 33–38. ISSN: 0001-0782. DOI: 10.1145/508448.508470. (Visited on 11/10/2021).

[268] Derrick Kondo, Gilles Fedak, Franck Cappello, Andrew A. Chien, and Henri Casanova. "Characterizing Resource Availability in Enterprise Desktop Grids." In: *Future Generation Computer Systems* 23.7 (Aug. 1, 2007), pp. 888–903. ISSN: 0167-739X. DOI: 10.1016/j.future.2006.11.001. (Visited on 03/14/2023).

[269] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. "The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems." In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (Washington, DC, USA). CCGRID '10. IEEE Computer Society, 2010, pp. 398–407. URL: http://dx.doi.org/10.1109/CCGRID.2010.71.

[270] Philip Koopman. "Elements of the Self-Healing System Problem Space." In: *WADS 2003 Workshop on Software Architectures for Dependable Systems* (Portland, OR). May 2003, pp. 31–36. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1679&amp;context=isr.

[271] Samuel Kounev et al. "The Notion of Self-aware Computing." In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Cham: Springer International Publishing, 2017, pp. 3–16. ISBN: 978-3-319-47474-8. DOI: 10.1007/978-3-319-47474-8_1. (Visited on 11/15/2022).

[272] Jeff Kramer. "Is Abstraction the Key to Computing?" In: *Commun. ACM* 50.4 (Apr. 1, 2007), pp. 36–42. ISSN: 0001-0782. DOI: 10.1145/1232743.1232745. (Visited on 11/12/2021).

[273] Jeff Kramer and Jeff Magee. "Self-Managed Systems: An Architectural Challenge." In: *FOSE '07: 2007 Future of Software Engineering* (Washington, DC, USA). IEEE Computer Society, 2007, pp. 259–268.

[274] Ben J. A. Kröse. "Learning from Delayed Rewards." In: *Robotics and Autonomous Systems* 15.4 (1995), p. 233. ISSN: 0921-8890. URL: https://www.academia.edu/3294050/Learning_from_delayed_rewards.

[275] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. "A Survey on Engineering Approaches for Self-Adaptive Systems." In: *Pervasive and Mobile Computing* 17 (2015), pp. 184–206. ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2014.09.009.

[276] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. New York, NY: Springer, 2013. ISBN: 978-1-4614-6848-6. DOI: 10.1007/978-1-4614-6849-3. (Visited on 03/14/2023).

[277] Marta Kwiatkowska and David Parker. "Automated Verification and Strategy Synthesis for Probabilistic Systems." In: *Automated Technology for Verification and Analysis*. Ed. by Dang Van Hung and Mizuhito Ogawa. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2013, pp. 5–22. ISBN: 978-3-319-02444-8. DOI: 10.1007/978-3-319-02444-8_2.

[278] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. *Autonomic Computing: Principles, Design and Implementation*. Undergraduate Topics in Computer Science. London: Springer, 2013. ISBN: 978-1-4471-5006-0. DOI: 10.1007/978-1-4471-5007-7.

[279] I.D. Landau and Y.D. Landau. *Adaptive Control: The Model Reference Approach*. Control and System Theory. Taylor & Francis, 1979. ISBN: 978-0-8247-6548-4.

[280] Ioan Doré Landau, Rogelio Lozano, Mohammed M'Saad, and Alireza Karimi. *Adaptive Control: Algorithms, Analysis and Applications*. Springer Science & Business Media, June 6, 2011. 595 pp. ISBN: 978-0-85729-664-1.

[281] J. C. Laprie. "Dependability: Basic Concepts and Terminology." In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Ed. by J. C. Laprie. Dependable Computing and Fault-Tolerant Systems. Vienna: Springer, 1992, pp. 3–245. ISBN: 978-3-7091-9170-5. DOI: 10.1007/978-3-7091-9170-5_1. (Visited on 02/25/2022).

[282] Daniel Le Métayer. "Software Architecture Styles as Graph Grammars." In: *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '96. New York, NY, USA: Association for Computing Machinery, Oct. 1, 1996, pp. 15–23. ISBN: 978-0-89791-797-1. DOI: 10.1145/239098.239105. (Visited on 11/24/2021).

[283] David B. Leake. *Case-Based Reasoning: Experiences, Lessons and Future Directions*. 1st. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0-262-62110-X.

[284] Peter Lewis, Kirstie L. Bellman, Christopher Landauer, Lukas Esterle, Kyrre Glette, Ada Diaconescu, and Holger Giese. "Towards a Framework for the Levels and Aspects of Self-aware Computing Systems." In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Cham: Springer International Publishing, 2017, pp. 51–85. ISBN: 978-3-319-47474-8. DOI: 10.1007/978-3-319-47474-8_3. (Visited on 01/27/2022).

[285] Peter R Lewis, Marco Platzner, Bernhard Rinner, Jim Tørresen, and Xin Yao. "Self-Aware Computing Systems." In: *Natural Computing Series. https://doi.org/10.1007/978-3-319-39675-0* (2016).

[286] Tong Li, Fan Zhang, and Dan Wang. "Automatic User Preferences Elicitation: A Data-Driven Approach." In: *Requirements Engineering: Foundation for Software Quality*. Ed. by Erik Kamsties, Jennifer Horkoff, and Fabiano Dalpiaz. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 324–331. ISBN: 978-3-319-77243-1. DOI: 10.1007/978-3-319-77243-1_21.

[287] Sarah Lichtenstein and Paul Slovic, eds. *The Construction of Preference*. Cambridge: Cambridge University Press, 2006. ISBN: 978-0-521-83428-5. DOI: 10.1017/CBO9780511618031. (Visited on 03/16/2023).

[288] Bing Liu. "Supervised Learning." In: *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Ed. by Bing Liu. Berlin, Heidelberg: Springer, 2011, pp. 63–132. ISBN: 978-3-642-19460-3. DOI: 10.1007/978-3-642-19460-3_3. (Visited on 03/26/2022).

[289] Hua Liu, M. Parashar, and S. Hariri. "A Component-Based Programming Model for Autonomic Applications." In: *International Conference on Autonomic Computing, 2004. Proceedings.* International Conference on Autonomic Computing, 2004. Proceedings. May 2004, pp. 10–17. DOI: 10.1109/ICAC.2004.1301341.

[290] Huan Liu and Hiroshi Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Vol. 454. Springer Science & Business Media, 2012.

[291] Yue Liu, Yuan Wang, Yuan Li, Bofeng Zhang, and Gengfeng Wu. "Earthquake Prediction by RBF Neural Network Ensemble." In: *Advances in Neural Networks - ISNN 2004*. Ed. by Fu-Liang Yin, Jun Wang, and Chengan Guo. Berlin, Heidelberg: Springer, 2004, pp. 962–969. ISBN: 978-3-540-28648-6. DOI: 10.1007/978-3-540-28648-6_153.

[292] Markus Luckey and Gregor Engels. "High-Quality Specification of Self-Adaptive Software Systems." In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '13. San Francisco, California: IEEE Press, May 20, 2013, pp. 143–152. ISBN: 978-1-4673-4401-2.

[293] Frank D. Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. "Self-Adaptive Systems: A Survey of Current Approaches, Research Challenges and Applications." In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2013.07.033.

[294] Pattie Maes. "Concepts and Experiments in Computational Reflection." In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '87. New York, NY, USA: Association for Computing Machinery, Dec. 1, 1987, pp. 147–155. ISBN: 978-0-89791-247-1. DOI: 10.1145/38765.38821. (Visited on 11/01/2021).

[295] João Paulo Magalhães and Luis Moura Silva. "SHõWA: A Self-Healing Framework for Web-Based Applications." In: *ACM Trans. Auton. Adapt. Syst.* 10.1 (Mar. 9, 2015), 4:1–4:28. ISSN: 1556-4665. DOI: 10.1145/2700325. (Visited on 07/14/2022).

[296] Jeff Magee and Jeff Kramer. "Dynamic Structure in Software Architectures." In: *SIGSOFT Softw. Eng. Notes* 21.6 (Nov. 1996), pp. 3–14. ISSN: 0163-5948. DOI: 10.1145/250707.239104. (Visited on 09/16/2021).

[297] Sara Mahdavi-Hezavehi, Vinicius H.S. Durelli, Danny Weyns, and Paris Avgeriou. "A Systematic Literature Review on Methods That Handle Multiple Quality Attributes in Architecture-Based Self-Adaptive Systems." In: *Information and Software Technology* 90 (2017), pp. 1–26. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2017.03.013.

[298] Jacob Mattingley, Yang Wang, and Stephen Boyd. "Receding Horizon Control." In: *IEEE Control Systems Magazine* 31.3 (June 2011), pp. 52–65. ISSN: 1941-000X. DOI: 10.1109/MCS.2011.940571.

[299] Mausam, Piergiorgio Bertoli, and Daniel S. Weld. "A Hybridized Planner for Stochastic Domains." In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Jan. 6, 2007, pp. 1972–1978.

[300] Dennis McCarthy and Umeshwar Dayal. "The Architecture of an Active Database Management System." In: *SIGMOD Rec.* 18.2 (June 1, 1989), pp. 215–224. ISSN: 0163-5808. DOI: 10.1145/66926.66946. (Visited on 09/20/2021).

[301] Philip K McKinley, Seyed Masoud Sadjadi, Eric P Kasten, and Betty H.C. Cheng. *A Taxonomy of Compositional Adaptation*. Technical Report MSU-CSE-04-17. 2004.

[302] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. "Composing Adaptive Software." In: *Computer* 37.7 (July 1, 2004), pp. 56–64. ISSN: 0018-9162. DOI: 10.1109/MC.2004.48. (Visited on 09/24/2021).

[303] N. Medvidovic and R.N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages." In: *IEEE Transactions on Software Engineering* 26.1 (Jan. 2000), pp. 70–93. ISSN: 1939-3520. DOI: 10.1109/32.825767.

[304] Daniel Menasce, Hassan Gomaa, sam Malek, and Joao Sousa. "SASSY: A Framework for Self-Architecting Service-Oriented Systems." In: *IEEE Software* 28.6 (Nov. 2011), pp. 78–85. ISSN: 1937-4194. DOI: 10.1109/MS.2011.22.

[305] Danilo Filgueira Mendonça, Genaína Nunes Rodrigues, Raian Ali, Vander Alves, and Luciano Baresi. "GODA: A Goal-Oriented Requirements Engineering Framework for Runtime Dependability Analysis." In: *Information and Software Technology* 80 (Dec. 1, 2016), pp. 245–264. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.09.005. (Visited on 11/11/2021).

[306] Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. "Robustness Testing Techniques and Tools." In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad van Moorsel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 323–339. ISBN: 978-3-642-29032-9. DOI: 10.1007/978-3-642-29032-9_16.

[307] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. "Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, Aug. 30, 2015, pp. 1–12. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786853. (Visited on 05/13/2022).

[308] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. "Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation." In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*. July 2016, pp. 147–156. DOI: 10.1109/ICAC.2016.59.

[309] Gabriel A. Moreno, Ofer Strichman, Sagar Chaki, and Radislav Vaisman. "Decision-Making with Cross-Entropy for Self-Adaptation." In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017 IEEE/ACM 12th International Symposium on

Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2017, pp. 90–101. DOI: 10.1109/SEAMS.2017.7.

[310] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. "Models@ Run.Time to Support Dynamic Adaptation." In: *Computer* 42.10 (Oct. 2009), pp. 44–51. ISSN: 1558-0814. DOI: 10.1109/MC.2009.327.

[311] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. "Taming Dynamically Adaptive Systems Using Models and Aspects." In: *2009 IEEE 31st International Conference on Software Engineering*. May 2009, pp. 122–132. DOI: 10.1109/ICSE.2009.5070514.

[312] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. "An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability." In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 782–796. ISBN: 978-3-540-87875-9. DOI: 10.1007/978-3-540-87875-9_54.

[313] Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais, and Jean-Marc Jézéquel. "Unifying Runtime Adaptation and Design Evolution." In: *2009 Ninth IEEE International Conference on Computer and Information Technology*. Vol. 1. Oct. 2009, pp. 104–109. DOI: 10.1109/CIT.2009.94.

[314] Brice Morin, Grégory Nain, Olivier Barais, and Jean-Marc Jézéquel. "Leveraging Models From Design-time to Runtime. A Live Demo." In: 4th International Workshop on Models@Run.Time (at MODELS'09). 2009. URL: https://hal.inria.fr/inria-00468520 (visited on 09/12/2022).

[315] Gunter Mussbacher et al. "The Relevance of Model-Driven Engineering Thirty Years from Now." In: *Model-Driven Engineering Languages and Systems*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Cham: Springer International Publishing, 2014, pp. 183–200. ISBN: 978-3-319-11653-2.

[316] Sangeeta Neti and Hausi A. Muller. "Quality Criteria and an Analysis Framework for Self-Healing Systems." In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*. International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07). May 2007. DOI: 10.1109/SEAMS.2007.15.

[317] Arnold Neumaier. "Solving Ill-Conditioned and Singular Linear Systems: A Tutorial on Regularization." In: *SIAM Rev.* 40.3 (Jan. 1998), pp. 636–666. ISSN: 0036-1445. DOI: 10.1137/S0036144597321909.

[318] Thomas D. Nielsen and Finn V. Jensen. "Learning a Decision Maker's Utility Function from (Possibly) Inconsistent Behavior." In: *Artificial Intelligence* 160.1 (Dec. 1, 2004), pp. 53–78. ISSN: 0004-3702. DOI: 10.1016/j.artint.2004.08.003. (Visited on 10/10/2022).

[319] Mohammed Nuseir and Hilda Madanat. "4Ps: A Strategy to Secure Customers' Loyalty via Customer Satisfaction." In: *International Journal of Marketing Studies* 7.4 (4 July 30, 2015), p78. ISSN: 1918-719X. DOI: 10.5539/ijms.v7n4p78. (Visited on 03/16/2023).

[320] Object Management Group OMG. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification Version 1.2*. Feb. 2015. URL: `https://www.omg.org/spec/QVT` (visited on 03/14/2023).

[321] P. Oreizy, N. Medvidovic, and R.N. Taylor. "Architecture-Based Runtime Software Evolution." In: *Proceedings of the 20th International Conference on Software Engineering*. Apr. 1998, pp. 177–186. DOI: `10.1109/ICSE.1998.671114`.

[322] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. "An Architecture-Based Approach to Self-Adaptive Software." In: *IEEE Intelligent Systems* 14.03 (May 1, 1999), pp. 54–62. ISSN: 1541-1672. DOI: `10.1109/5254.769885`.

[323] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. "Runtime Software Adaptation: Framework, Approaches, and Styles." In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion '08. New York, NY, USA: Association for Computing Machinery, May 10, 2008, pp. 899–910. ISBN: 978-1-60558-079-1. DOI: `10.1145/1370175.1370181`.

[324] *Overengineering*. In: *Wikipedia*. Feb. 18, 2023. URL: `https://en.wikipedia.org/w/index.php?title=Overengineering&oldid=1140099554` (visited on 03/14/2023).

[325] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning." In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (Oct. 2010), pp. 1345–1359. ISSN: 1558-2191. DOI: `10.1109/TKDE.2009.191`.

[326] Ashutosh Pandey. "Hybrid Planning in Self-Adaptive Systems." In: (Mar. 2020). DOI: `10.1184/R1/11926836.v1`.

[327] Ashutosh Pandey, Gabriel A. Moreno, Javier Cámara, and David Garlan. "Hybrid Planning for Decision Making in Self-Adaptive Systems." In: *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. Sept. 2016, pp. 130–139. DOI: `10.1109/SASO.2016.19`.

[328] Ashutosh Pandey, Ivan Ruchkin, Bradley Schmerl, and Javier Cámara. "Towards a Formal Framework for Hybrid Planning in Self-Adaptation." In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2017, pp. 109–115. DOI: `10.1109/SEAMS.2017.14`.

[329] Ashutosh Pandey, Bradley Schmerl, and David Garlan. "Instance-Based Learning for Hybrid Planning." In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W). Sept. 2017, pp. 64–69. DOI: `10.1109/FAS-W.2017.122`.

[330] Vassilis A. Papavassiliou and Stuart Russell. "Convergence of Reinforcement Learning with General Function Approximators." In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., July 31, 1999, pp. 748–755.

[331] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. "Run-Time Adaptation of Mobile Applications Using Genetic Algorithms." In: *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2013, pp. 73–82. DOI: 10.1109/SEAMS.2013.6595494.

[332] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. "A Systematic Survey on the Design of Self-Adaptive Software Systems Using Control Engineering Approaches." In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). June 2012, pp. 33–42. DOI: 10.1109/SEAMS.2012.6224389.

[333] Relu Patrascu, Craig Boutilier, Rajarshi Das, Jeffrey O. Kephart, Gerald Tesauro, and William E. Walsh. "New Approaches to Optimization and Utility Elicitation in Autonomic Computing." In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*. AAAI'05. Pittsburgh, Pennsylvania: AAAI Press, July 9, 2005, pp. 140–145. ISBN: 978-1-57735-236-5.

[334] Nicolò Perino. "A Framework for Self-Healing Software Systems." In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013 35th International Conference on Software Engineering (ICSE). May 2013, pp. 1397–1400. DOI: 10.1109/ICSE.2013.6606726.

[335] Sanja Petrovic and Rong Qu. "Case-Based Reasoning as a Heuristic Selector in a Hyper-Heuristic for Course Timetabling Problems." In: *Knowledge-Based Intelligent Information Engineering Systems and Allied Technologies*. Vol. 82. Frontiers in Artificial Intelligence and Applications. IOS Press, 2002. ISBN: 978-1-58603-280-7.

[336] Mauro Pezzè. "From Off-Line to Continuous on-Line Maintenance." In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Sept. 2012, pp. 2–3. DOI: 10.1109/ICSM.2012.6405244.

[337] É. Piel, A. Gonzalez-Sanchez, H. G. Gross, and A. J. C. v. Gemund. "Spectrum-Based Health Monitoring for Self-Adaptive Systems." In: *2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems*. Oct. 2011, pp. 99–108.

[338] Gabriella Pigozzi, Alexis Tsoukiàs, and Paolo Viappiani. "Preferences in Artificial Intelligence." In: *Ann Math Artif Intell* 77.3 (Aug. 1, 2016), pp. 361–401. ISSN: 1573-7470. DOI: 10.1007/s10472-015-9475-5.

[339] Joelle Pineau, Geoff Gordon, and Sebastian Thrun. "Point-Based Value Iteration: An Anytime Algorithm for POMDPs." In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. IJCAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 9, 2003, pp. 1025–1030.

[340] Robert A. Pollak. "Additive von Neumann-Morgenstern Utility Functions." In: *Econometrica* 35.3/4 (1967), pp. 485–494. ISSN: 00129682, 14680262. JSTOR: 1905650. URL: http://www.jstor.org/stable/1905650.

[341] Barry Porter and Roberto Rodrigues Filho. "Losing Control: The Case for Emergent Software Systems Using Autonomous Assembly, Perception, and Learning." In: *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Sept. 2016, pp. 40–49. DOI: 10.1109/SASO.2016.10.

[342] Barry Porter, Roberto Rodrigues Filho, and Paul Dean. "A Survey of Methodology in Self-Adaptive Systems Research." In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). Aug. 2020, pp. 168–177. DOI: 10.1109/ACSOS49614.2020.00039.

[343] Harald Psaier and Schahram Dustdar. "A Survey on Self-Healing Systems: Approaches and Systems." In: *Computing* 91.1 (Jan. 1, 2011), pp. 43–73. ISSN: 1436-5057. DOI: 10.1007/s00607-010-0107-y.

[344] P. Pudil, F.J. Ferri, J. Novovicova, and J. Kittler. "Floating Search Methods for Feature Selection with Nonmonotonic Criterion Functions." In: *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Vol. 3 - Conference C: Signal Processing (Cat. No.94CH3440-5)*. Proceedings of the 12th IAPR International Conference on Pattern Recognition, Vol. 3 - Conference C: Signal Processing (Cat. No.94CH3440-5). Vol. 2. Oct. 1994, 279–283 vol.2. DOI: 10.1109/ICPR.1994.576920.

[345] Martin L. Puterman. "Chapter 8 Markov Decision Processes." In: *Handbooks in Operations Research and Management Science*. Vol. 2. Stochastic Models. Elsevier, Jan. 1, 1990, pp. 331–434. DOI: 10.1016/S0927-0507(05)80172-0. (Visited on 10/04/2022).

[346] Kashifuddin Qazi, Yang Li, and Andrew Sohn. "Workload Prediction of Virtual Machines for Harnessing Data Center Resources." In: *2014 IEEE 7th International Conference on Cloud Computing*. 2014 IEEE 7th International Conference on Cloud Computing. June 2014, pp. 522–529. DOI: 10.1109/CLOUD.2014.76.

[347] Yanjun Qi. "Random Forest for Bioinformatics." In: *Ensemble Machine Learning: Methods and Applications*. Ed. by Cha Zhang and Yunqian Ma. Boston, MA: Springer US, 2012, pp. 307–323. ISBN: 978-1-4419-9326-7. DOI: 10.1007/978-1-4419-9326-7_11. (Visited on 03/30/2022).

[348] Wenyi Qian, Xin Peng, Bihuan Chen, John Mylopoulos, Huanhuan Wang, and Wenyun Zhao. "Rationalism with a Dose of Empiricism: Combining Goal Reasoning and Case-Based Reasoning for Self-Adaptive Software Systems." In: *Requirements Eng* 20.3 (Sept. 1, 2015), pp. 233–252. ISSN: 1432-010X. DOI: 10.1007/s00766-015-0227-1. (Visited on 10/07/2022).

[349] J. R. Quinlan. "Induction of Decision Trees." In: *Mach Learn* 1.1 (Mar. 1, 1986), pp. 81–106. ISSN: 1573-0565. DOI: 10.1007/BF00116251. (Visited on 03/16/2023).

[350] Yang Qun, Yang Xian-chun, and Xu Man-wu. "A Framework for Dynamic Software Architecture-Based Self-Healing." In: *2005 IEEE International Conference on Systems, Man and Cybernetics*. Vol. 3. Oct. 2005, 2968–2972 Vol. 3.

[351] Rahul Raheja, Shang-Wen Cheng, David Garlan, and Bradley R. Schmerl. "Improving Architecture-Based Self-Adaptation Using Preemption." In: *Self-Organizing Architectures, First International Workshop SOAR 2009, Cambridge, UK, September 14, 2009, Revised Selected and Invited Papers*. Ed. by Danny Weyns, Sam Malek, Rogério de Lemos, and Jesper Andersson. Vol. 6090. Lecture Notes in Computer Science. Springer, 2010, pp. 21–37.

[352] Andres J. Ramirez, Betty H.C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. "Automatically Generating Adaptive Logic to Balance Non-Functional Tradeoffs during Reconfiguration." In: *Proceedings of the 7th International Conference on Autonomic Computing*. ICAC '10. New York, NY, USA: Association for Computing Machinery, June 7, 2010, pp. 225–234. ISBN: 978-1-4503-0074-2. DOI: `10.1145/1809049.1809080`. (Visited on 10/05/2022).

[353] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. "Applying Genetic Algorithms to Decision Making in Autonomic Computing Systems." In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC '09. New York, NY, USA: Association for Computing Machinery, June 15, 2009, pp. 97–106. ISBN: 978-1-60558-564-2. DOI: `10.1145/1555228.1555258`. (Visited on 10/05/2022).

[354] David Redlich, Gordon Blair, Awais Rashid, Thomas Molka, and Wasif Gilani. "Research Challenges for Business Process Models at Run-Time." In: *Models@run.Time: Foundations, Applications, and Roadmaps*. Ed. by Nelly Bencomo, Robert France, Betty H. C. Cheng, and Uwe Aßmann. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 208–236. ISBN: 978-3-319-08915-7. DOI: `10.1007/978-3-319-08915-7_8`. (Visited on 02/19/2022).

[355] Mark Richters and Martin Gogolla. "OCL: Syntax, Semantics, and Tools." In: *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*. Ed. by Tony Clark and Jos Warmer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 42–68. ISBN: 978-3-540-45669-8. DOI: `10.1007/3-540-45669-4_4`. (Visited on 06/30/2022).

[356] Peter Ross, Emma Hart, and Dave Corne. "Some Observations about GA-based Exam Timetabling." In: *Practice and Theory of Automated Timetabling II*. Ed. by Edmund Burke and Michael Carter. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 115–129. ISBN: 978-3-540-49803-2. DOI: `10.1007/BFb0055884`.

[357] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. "Online Planning Algorithms for POMDPs." In: *Journal of Artificial Intelligence Research* 32 (July 29, 2008), pp. 663–704. ISSN: 1076-9757. DOI: `10.1613/jair.2567`. (Visited on 10/27/2022).

[358] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. USA: Elsevier Science Inc., 2006. 978 pp. ISBN: 978-0-08-046380-3.

[359] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer Berlin / Heidelberg, 2009, pp. 164–182. URL: `http://dx.doi.org/10.1007/978-3-642-02161-9_9`.

[360] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav. “Composing Components and Services Using a Planning-Based Adaptation Middleware.” In: *Software Composition*. Ed. by Cesare Pautasso and Éric Tanter. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 52–67. ISBN: 978-3-540-78789-1. DOI: 10.1007/978-3-540-78789-1_4.

[361] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[362] Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. “A Scalable Querying Scheme for Memory-Efficient Runtime Models with History.” In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’20. New York, NY, USA: Association for Computing Machinery, Oct. 16, 2020, pp. 175–186. ISBN: 978-1-4503-7019-6. DOI: 10.1145/3365438.3410961.

[363] Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. “Incremental Execution of Temporal Graph Queries over Runtime Models with History and Its Applications.” In: *Softw Syst Model* 21.5 (Oct. 1, 2022), pp. 1789–1829. ISSN: 1619-1374. DOI: 10.1007/s10270-021-00950-6.

[364] Lucas Sakizloglou, Sona Ghahremani, Thomas Brand, Matthias Barkowsky, and Holger Giese. “Towards Highly Scalable Runtime Models with History.” In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’20. New York, NY, USA: Association for Computing Machinery, Sept. 18, 2020, pp. 188–194. ISBN: 978-1-4503-7962-5. DOI: 10.1145/3387939.3388614.

[365] M. Salama, R. Bahsoon, and N. Bencomo. “Chapter 11 - Managing Trade-Offs in Self-Adaptive Software Architectures: A Systematic Mapping Study.” In: *Managing Trade-Offs in Adaptable Software Architectures*. Ed. by Ivan Mistrik, Nour Ali, Rick Kazman, John Grundy, and Bradley Schmerl. Boston: Morgan Kaufmann, 2017, pp. 249–297. ISBN: 978-0-12-802855-1. DOI: 10.1016/B978-0-12-802855-1.00011-3.

[366] Mazeiar Salehie and Ladan Tahvildari. “A Coordination Mechanism for Self-healing and Self-optimizing Disciplines.” In: *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*. SEAMS ’06. New York, NY, USA: ACM, 2006, pp. 98–98. URL: http://doi.acm.org/10.1145/1137677.1137701.

[367] Mazeiar Salehie and Ladan Tahvildari. “Self-Adaptive Software: Landscape and Research Challenges.” In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (May 21, 2009), 14:1–14:42. ISSN: 1556-4665. DOI: 10.1145/1516533.1516538. (Visited on 06/28/2021).

[368] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. “Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T).” In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 342–352. DOI: 10.1109/ASE.2015.45.

[369] Matthias Schmid, Torsten Hothorn, Kelly O. Maloney, Donald E. Weller, and Sergej Potapov. “Geoadditive Regression Modeling of Stream Biological Condition.” In: *Environ Ecol Stat* 18.4 (Dec. 1, 2011), pp. 709–733. ISSN: 1573-3009. DOI: 10.1007/s10651-010-0158-4. (Visited on 03/30/2022).

[370] Julia Schmitt, Michael Roth, Rolf Kiefhaber, Florian Kluge, and Theo Ungerer. "Realizing Self-x Properties by an Automated Planner." In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. New York, NY, USA: ACM, 2011, pp. 185–186. URL: `http://doi.acm.org/10.1145/1998582.1998620`.

[371] Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. "Toward Causal Representation Learning." In: *Proceedings of the IEEE* 109.5 (May 2021), pp. 612–634. ISSN: 1558-2256. DOI: `10.1109/JPROC.2021.3058954`.

[372] Patrick Schratz, Jannes Muenchow, Eugenia Iturritxa, Jakob Richter, and Alexander Brenning. "Hyperparameter Tuning and Performance Assessment of Statistical and Machine-Learning Algorithms Using Spatial Data." In: *Ecological Modelling* 406 (Aug. 24, 2019), pp. 109–120. ISSN: 0304-3800. DOI: `10.1016/j.ecolmodel.2019.06.002`. (Visited on 03/24/2022).

[373] Andy Schürr. "Specification of Graph Translators with Triple Graph Grammars." In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 151–163. ISBN: 978-3-540-49183-5. DOI: `10.1007/3-540-59071-4_45`.

[374] Dale E. Seborg, Duncan A. Mellichamp, Thomas F. Edgar, and Francis J. Doyle. *Process Dynamics and Control*. 3rd. John Wiley & Sons, 2011.

[375] B. Selic. "The Pragmatics of Model-Driven Development." In: *IEEE Software* 20.5 (Sept. 2003), pp. 19–25. ISSN: 1937-4194. DOI: `10.1109/MS.2003.1231146`.

[376] Bran Selic. "The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective." In: *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 290–321. ISBN: 978-3-642-18023-1. DOI: `10.1007/978-3-642-18023-1_7`.

[377] S. Sendall and W. Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development." In: *IEEE Software* 20.5 (Sept. 2003), pp. 42–45. ISSN: 1937-4194. DOI: `10.1109/MS.2003.1231150`.

[378] Peter Sestoft. *Microbenchmarks in Java and C#*. URL: `http://www.itu.dk/people/sestoft/papers/benchmarking.pdf` (visited on 03/14/2023).

[379] Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, and Klaus Pohl. "Learning and Evolution in Dynamic Software Product Lines." In: *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2016, pp. 158–164. DOI: `10.1145/2897053.2897058`.

[380] Mary Shaw. "Beyond Objects: A Software Design Paradigm Based on Process Control." In: *SIGSOFT Softw. Eng. Notes* 20.1 (Jan. 1, 1995), pp. 27–38. ISSN: 0163-5948. DOI: `10.1145/225907.225911`. (Visited on 01/27/2022).

[381] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[382] Chang Shu and Donald H. Burn. "Artificial Neural Network Ensembles and Their Application in Pooled Flood Frequency Analysis." In: *Water Resources Research* 40.9 (2004). ISSN: 1944-7973. DOI: 10.1029/2003WR002816. (Visited on 03/30/2022).

[383] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, Aug. 30, 2015, pp. 284–294. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786845.

[384] Kornel Skałkowski and Krzysztof Zieliński. "Automatic Adaptation of SOA Systems Supported by Machine Learning." In: *Technological Innovation for the Internet of Things*. Ed. by Luis M. Camarinha-Matos, Slavisa Tomic, and Paula Graça. IFIP Advances in Information and Communication Technology. Berlin, Heidelberg: Springer, 2013, pp. 61–68. ISBN: 978-3-642-37291-9. DOI: 10.1007/978-3-642-37291-9_7.

[385] C.U. Smith and L.G. Williams. "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives." In: *IEEE Transactions on Software Engineering* 19.7 (July 1993), pp. 720–741. ISSN: 1939-3520. DOI: 10.1109/32.238572.

[386] Ronald D. Snee. "Validation of Regression Models: Methods and Examples." In: *Technometrics* 19.4 (Nov. 1977), pp. 415–428. ISSN: 0040-1706. DOI: 10.1080/00401706.1977.10489581.

[387] Hui Song, Stephen Barrett, Aidan Clarke, and Siobhán Clarke. "Self-Adaptation with End-User Preferences: Using Run-Time Models and Constraint Solving." In: *Model-Driven Engineering Languages and Systems*. Ed. by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 555–571. ISBN: 978-3-642-41533-3. DOI: 10.1007/978-3-642-41533-3_34.

[388] Hui Song, Xiaodong Zhang, Nicolas Ferry, Franck Chauvel, Arnor Solberg, and Gang Huang. "Modelling Adaptation Policies as Domain-Specific Constraints." In: *Model-Driven Engineering Languages and Systems*. Ed. by Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 269–285. ISBN: 978-3-319-11653-2. DOI: 10.1007/978-3-319-11653-2_17.

[389] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. "Extending Dynamic Software Product Lines with Temporal Constraints." In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2017, pp. 129–139. DOI: 10.1109/SEAMS.2017.6.

[390] Michael Stein, Alexander Frömmgen, Roland Kluge, Frank Löffler, Andy Schürr, Alejandro Buchmann, and Max Mühlhäuser. "TARL: Modeling Topology Adaptations for Networking Applications." In: *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2016, pp. 57–63. DOI: 10.1109/SEAMS.2016.014.

[391] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd ed. Addison-Wesley Professional, 2009. ISBN: 978-0-321-33188-5.

[392] Christopher Stewart, Terence Kelly, and Alex Zhang. "Exploiting Nonstationarity for Performance Prediction." In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. New York, NY, USA: Association for Computing Machinery, Mar. 21, 2007, pp. 31–44. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273002. (Visited on 03/02/2023).

[393] J. Strassner. "How Policy Empowers Business-Driven Device Management." In: *Proceedings Third International Workshop on Policies for Distributed Systems and Networks*. Proceedings Third International Workshop on Policies for Distributed Systems and Networks. June 2002, pp. 214–217. DOI: 10.1109/POLICY.2002.1011311.

[394] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. "Plan-Directed Architectural Change for Autonomous Systems." In: *Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. SAVCBS '07. New York, NY, USA: Association for Computing Machinery, Sept. 3, 2007, pp. 15–21. ISBN: 978-1-59593-721-6. DOI: 10.1145/1292316.1292318.

[395] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. "From Goals to Components: A Combined Approach to Self-Management." In: *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '08. New York, NY, USA: Association for Computing Machinery, May 12, 2008, pp. 1–8. ISBN: 978-1-60558-037-1. DOI: 10.1145/1370018.1370020. (Visited on 10/04/2022).

[396] Michael Szvetits and Uwe Zdun. "Systematic Literature Review of the Objectives, Techniques, Kinds, and Architectures of Models at Runtime." In: *Softw Syst Model* 15.1 (Feb. 1, 2016), pp. 31–69. ISSN: 1619-1374. DOI: 10.1007/s10270-013-0394-9. (Visited on 10/26/2022).

[397] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. "Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems." In: *Theory and Application of Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 179–193. ISBN: 978-3-540-46464-8. DOI: 10.1007/978-3-540-46464-8_13.

[398] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. "PLASMA: A Plan-Based Layered Architecture for Software Model-Driven Adaptation." In: *Proceedings of the IEEE/ACM International Conference on Automated*

Software Engineering. ASE '10. New York, NY, USA: Association for Computing Machinery, Sept. 20, 2010, pp. 467–476. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859092. (Visited on 05/16/2022).

[399] Abhijeet Tallavajhula, Sanjiban Choudhury, Sebastian Scherer, and Alonzo Kelly. "List Prediction Applied to Motion Planning." In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016 IEEE International Conference on Robotics and Automation (ICRA). May 2016, pp. 213–220. DOI: 10.1109/ICRA.2016.7487136.

[400] D. Tang and Ravishankar K. Iyer. "Dependability Measurement and Modeling of a Multicomputer System." In: *IEEE Transactions on Computers* 42.1 (Jan. 1993), pp. 62–75.

[401] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation." In: *2006 IEEE International Conference on Autonomic Computing*. 2006 IEEE International Conference on Autonomic Computing. June 2006, pp. 65–73. DOI: 10.1109/ICAC.2006.1662383.

[402] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das, and Mohamed N. Bennani. "On the Use of Hybrid Reinforcement Learning for Autonomic Resource Allocation." In: *Cluster Comput* 10.3 (Sept. 1, 2007), pp. 287–299. ISSN: 1573-7543. DOI: 10.1007/s10586-007-0035-6. (Visited on 10/07/2022).

[403] Matthias Tichy and Holger Giese. "A Self-optimizing Run-Time Architecture for Configurable Dependability of Services." In: *Architecting Dependable Systems II*. Ed. by Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 25–50. ISBN: 978-3-540-25939-8. DOI: 10.1007/978-3-540-25939-8_2.

[404] Matthias Tichy and Benjamin Klöpper. "Planning Self-adaption with Graph Transformations." In: *Applications of Graph Transformations with Industrial Relevance*. Ed. by Andy Schürr, Dániel Varró, and Gergely Varró. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 137–152. ISBN: 978-3-642-34176-2. DOI: 10.1007/978-3-642-34176-2_13.

[405] Frank Trollmann, Johannes Fähndrich, and Sahin Albayrak. "Hybrid Adaptation Policies: Towards a Framework for Classification and Modelling of Different Combinations of Adaptation Policies." In: *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '18. New York, NY, USA: Association for Computing Machinery, May 28, 2018, pp. 76–86. ISBN: 978-1-4503-5715-9. DOI: 10.1145/3194133.3194137. (Visited on 05/12/2022).

[406] Tom van Dijk. "Analysing and Improving Hash Table Performance." In: *10th Twente Student Conference on IT*. Twente: University of twente, 2009. URL: https://www.tvandijk.nl/pdf/bscthesis.pdf (visited on 03/14/2023).

[407] Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Springer science & business media, 1999. ISBN: 0-387-98780-0.

[408] Dániel Varró. "Model Transformation by Example." In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 410–424. ISBN: 978-3-540-45773-2. DOI: 10.1007/11880240_29.

[409] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. "Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework." In: *Softw Syst Model* 15.3 (July 1, 2016), pp. 609–629. ISSN: 1619-1374. DOI: 10.1007/s10270-016-0530-4. (Visited on 11/29/2021).

[410] Paolo Viappiani and Craig Boutilier. "Regret-Based Optimal Recommendation Sets in Conversational Recommender Systems." In: *Proceedings of the Third ACM Conference on Recommender Systems*. RecSys '09. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2009, pp. 101–108. ISBN: 978-1-60558-435-5. DOI: 10.1145/1639714.1639732. (Visited on 02/16/2022).

[411] Ricardo Vilalta and Youssef Drissi. "A Perspective View and Survey of Meta-Learning." In: *Artificial Intelligence Review* 18.2 (June 1, 2002), pp. 77–95. ISSN: 1573-7462. DOI: 10.1023/A:1019956318069. (Visited on 01/23/2023).

[412] Attila Vizhanyo, Aditya Agrawal, and Feng Shi. "Towards Generation of Efficient Transformations." In: *Generative Programming and Component Engineering*. Ed. by Gabor Karsai and Eelco Visser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 298–316. ISBN: 978-3-540-30175-2. DOI: 10.1007/978-3-540-30175-2_16.

[413] Thomas Vogel. "Model-Driven Engineering of Self-Adaptive Software." PhD thesis. Universität Potsdam, 2018.

[414] Thomas Vogel. "mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization." In: *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '18. New York, NY, USA: Association for Computing Machinery, May 28, 2018, pp. 101–107. ISBN: 978-1-4503-5715-9. DOI: 10.1145/3194133.3194161. (Visited on 11/17/2021).

[415] Thomas Vogel and Holger Giese. "Adaptation and Abstract Runtime Models." In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. New York, NY, USA: Association for Computing Machinery, May 3, 2010, pp. 39–48. ISBN: 978-1-60558-971-8. DOI: 10.1145/1808984.1808989. (Visited on 09/16/2021).

[416] Thomas Vogel and Holger Giese. "Model-Driven Engineering of Self-Adaptive Software with EUREMA." In: *ACM Trans. Auton. Adapt. Syst.* 8.4 (Jan. 1, 2014), 18:1–18:33. ISSN: 1556-4665. DOI: 10.1145/2555612.

[417] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. "Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems." In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC '09. New York, NY, USA: Association for Computing Machinery, June 15, 2009, pp. 67–68. ISBN: 978-1-60558-564-2. DOI: 10.1145/1555228.1555249. (Visited on 11/19/2021).

[418] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. "Incremental Model Synchronization for Efficient Run-Time Monitoring." In: *Models in Software Engineering*. Ed. by Sudipto Ghosh. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 124–139. ISBN: 978-3-642-12261-3. DOI: 10.1007/978-3-642-12261-3_13.

[419] John von Neumann, Oskar Morgenstern, and Ariel Rubinstein. *Theory of Games and Economic Behavior*. Princeton University Press, 1944. ISBN: 978-0-691-13061-3.

[420] Peter Walley. *Statistical Reasoning With Imprecise Probabilities*. London ; New York: Chapman & Hall/CRC, Mar. 1, 1991. 720 pp. ISBN: 978-0-412-28660-5.

[421] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. "Utility Functions in Autonomic Systems." In: *International Conference on Autonomic Computing, 2004. Proceedings.* International Conference on Autonomic Computing, 2004. Proceedings. May 2004, pp. 70–77. DOI: 10.1109/ICAC.2004.1301349.

[422] Shengquan Wang, Dong Xuan, R. Bettati, and Wei Zhao. "Providing Absolute Differentiated Services for Real-Time Applications in Static-Priority Scheduling Networks." In: *IEEE/ACM Transactions on Networking* 12.2 (Apr. 2004), pp. 326–339. ISSN: 1558-2566. DOI: 10.1109/TNET.2004.826286.

[423] Tianhan Wang and Craig Boutilier. "Incremental Utility Elicitation with Minimax Regret Decision Criterion." In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. IJCAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 9, 2003, pp. 309–316.

[424] Michel Wermelinger and José Luiz Fiadeiro. "A Graph Transformation Approach to Software Architecture Reconfiguration." In: *Science of Computer Programming*. Special Issue on Applications of Graph Transformations (GRATRA 2000) 44.2 (Aug. 1, 2002), pp. 133–155. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(02)00036-9. (Visited on 11/16/2021).

[425] Danny Weyns and Tanvir Ahmad. "Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review." In: *Software Architecture*. Ed. by Khalil Drira. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 249–265. ISBN: 978-3-642-39031-9. DOI: 10.1007/978-3-642-39031-9_22.

[426] Danny Weyns, M. Usman Iftikhar, Sam Malek, and Jesper Andersson. "Claims and Supporting Evidence for Self-Adaptive Systems: A Literature Study." In: *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. Zurich, Switzerland: IEEE Press, June 4, 2012, pp. 89–98. ISBN: 978-1-4673-1787-0.

[427] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. "On Patterns for Decentralized Control in Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 76–107. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_4. (Visited on 01/28/2022).

[428] Garrett Wilson and Diane J. Cook. "A Survey of Unsupervised Deep Domain Adaptation." In: *ACM Trans. Intell. Syst. Technol.* 11.5 (July 5, 2020), 51:1–51:46. ISSN: 2157-6904. DOI: 10.1145/3400066. (Visited on 01/23/2023).

[429] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2.

[430] Murray Woodside, Greg Franks, and Dorina C. Petriu. "The Future of Software Performance Engineering." In: *Future of Software Engineering (FOSE '07)*. Future of Software Engineering (FOSE '07). May 2007, pp. 171–187. DOI: 10.1109/FOSE.2007.32.

[431] Di Wu and Ling Shao. "Leveraging Hierarchical Parametric Networks for Skeletal Joints Based Action Segmentation and Recognition." In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '14. USA: IEEE Computer Society, June 23, 2014, pp. 724–731. ISBN: 978-1-4799-5118-5. DOI: 10.1109/CVPR.2014.98. (Visited on 10/10/2022).

[432] Tao Ye and Shivkumar Kalyanaraman. "A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration." In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '03. New York, NY, USA: Association for Computing Machinery, June 10, 2003, pp. 196–205. ISBN: 978-1-58113-664-7. DOI: 10.1145/781027.781052. (Visited on 10/10/2022).

[433] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. "Towards Machine Learning-Based Auto-tuning of MapReduce." In: *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems. Aug. 2013, pp. 11–20. DOI: 10.1109/MASCOTS.2013.9.

[434] Jianfeng Zhan, Lei Wang, Xiaona Li, Weisong Shi, Chuliang Weng, Wenyao Zhang, and Xiutao Zang. "Cost-Aware Cooperative Resource Provisioning for Heterogeneous Workloads in Data Centers." In: *IEEE Transactions on Computers* 62.11 (Nov. 2013), pp. 2155–2168. ISSN: 1557-9956. DOI: 10.1109/TC.2012.103.

[435] Du Zhang. "Machine Learning in Value-Based Software Test Data Generation." In: *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*. 2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06). Nov. 2006, pp. 732–736. DOI: 10.1109/ICTAI.2006.77.

[436] Yanyong Zhang, Mark S. Squillante, Anand Sivasubramaniam, and Ramendra K. Sahoo. "Performance Implications of Failures in Large-Scale Cluster Scheduling." In: *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004. Revised Selected Papers*. Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Springer Berlin Heidelberg, 2005, pp. 233–252.

[437] Tianqi Zhao, Wei Zhang, Haiyan Zhao, and Zhi Jin. "A Reinforcement Learning-Based Framework for the Generation and Evolution of Adaptation Rules." In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. July 2017, pp. 103–112. DOI: 10.1109/ICAC.2017.47.

[438] Yujun Zheng, Chuanqing Xu, and Jinyun Xue. "A Simple Greedy Algorithm for a Class of Shuttle Transportation Problems." In: *Optim Lett* 3.4 (Sept. 1, 2009), pp. 491–497. ISSN: 1862-4480. DOI: 10.1007/s11590-009-0126-9. (Visited on 02/03/2022).

[439] Kaiyang Zhou, Ziwei Liu, Yu Qiao, Tao Xiang, and Chen Change Loy. "Domain Generalization: A Survey." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), pp. 1–20. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2022.3195549.

[440] Shlomo Zilberstein. "Using Anytime Algorithms in Intelligent Systems." In: *AIMag* 17.3 (Mar. 15, 1996), p. 73. DOI: 10.1609/aimag.v17i3.1232. (Visited on 10/06/2022).

Part IV

APPENDIX

A TECHNICAL SUPPLEMENT

This chapter introduces the *Story Diagram* (SD) formalism and its application in this thesis to operationalize the adaptation engine activities.

A.1 INTRODUCTION TO STORY DIAGRAM FORMALISM

Structural changes of models in this thesis are implemented via a mechanism based on graph transformations [27]. We used the *Story-driven Modeling* (SDM) tool[1] that is a model execution engine comprising an editor, interpreter, and debugger for SDs. Modifications to the RTM are done via SDs as the modeling language that realizes the graph (model) transformation rules. SDs, originally introduced by [141], combine the notion of UML Activity Diagrams and graph transformations, whereby each action node of an activity describes and executes a graph transformation. An SD structures an initial, a final, and one or more action node(s); all action nodes of an activity are structured in a control flow. Action nodes are realized by *Story Patterns* (SPs) (yellow nodes) or Call Actions (gray nodes). An SP describes and executes a graph transformation while a Call Action invokes either another SD or code. Code may replace the graph transformation if code is the more suitable formalism to specify the behavior, e.g., to realize mathematical algorithms that do not work on the graph structure of the model. Consequently, an SD specifies an activity as a flow of action nodes that are usually graph transformations. An SD has one or more input parameters and may also have output parameters to define return values.

SDs are similar to *Event-Condition-Action* (ECA) rules and capture the *Left-hand Side* (LHS) and *Right-hand Side* (RHS) within the action nodes. They allow to specify patterns at the same level of abstraction as the RTM. the Analyze, Plan, and Execute activities of the MAPE-K loop in this thesis are realized via SD-based rules and are referred to as analysis, planning, and execution rules, respectively. In Chapter 6, we briefly introduced the SD formalism via the examples of the Analyze, Plan, and Execute rules for the `overBudget` issue in `Znn.com`—see Figure 6.4, Figure 6.5a, and Figure 6.5b, respectively. In this chapter, we present the Analyze, Plan, and Execute rules for the `CF2` issue in mRUBiS as another set of examples to discuss the SD formalism—see Figure A.1, Figure A.2, and Figure A.3. Each of these activities are SDs with one initial and one final node and two input parameters.

A graph transformation expressed in an SP defines a graph rewriting, i.e., an in-place model transformation. Therefore, an SP refers to the elements of a model, which are represented in the abstract syntax of the corresponding language within the SP. An SP denotes the model elements in three forms: (i) black nodes and edges without any annotation, (ii) red nodes and edges annotated with «DESTROY» respectively «destroy», and (iii) green nodes and edges annotated with «CREATE» respectively «create». The

1 Story-Driven Modeling (SDM) Tools: `http://www.mdelab.de/mdelab-projects/story-diagram-tools`– accessed 18 March 2023.

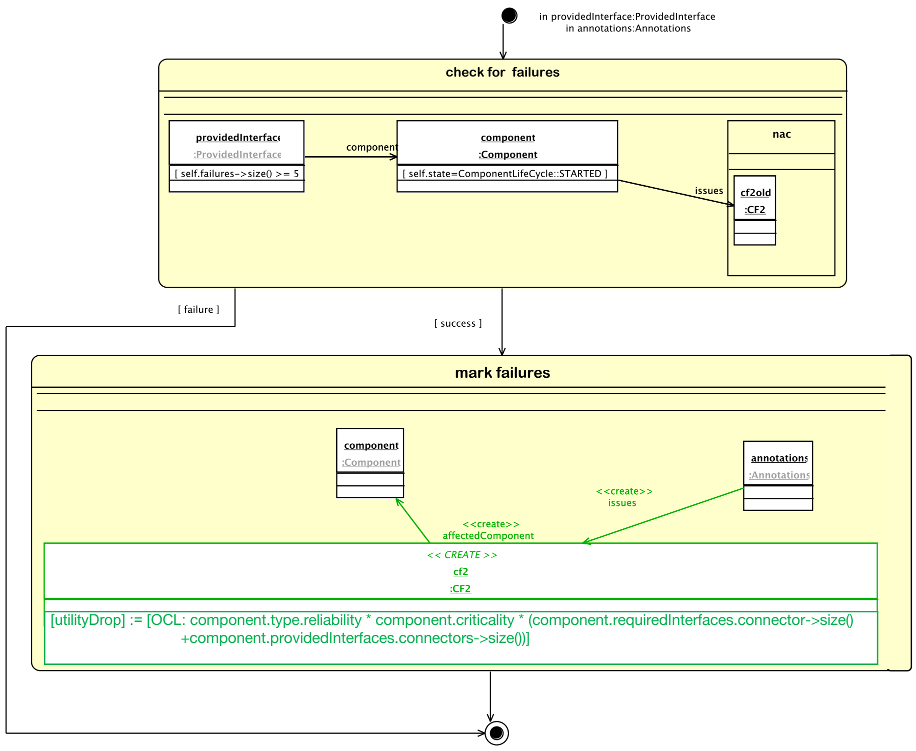

in providedInterface:ProvidedInterface
in annotations:Annotations
check for failures
providedInterface
:ProvidedInterface
[self.failures->size() >= 5]
component
component
:Component
[self.state=ComponentLifeCycle::STARTED]
nac
issues
cf2old
:CF2
[failure]
[success]
mark failures
component
:Component
annotations
:Annotations
<<create>>
issues
<<create>>
affectedComponent
<< CREATE >>
cf2
:CF2
[utilityDrop] := [OCL: component.type.reliability * component.criticality * (component.requiredInterfaces.connector->size() +component.providedInterfaces.connectors->size())]

Figure A.1: Analyze SD for CF2 issue.

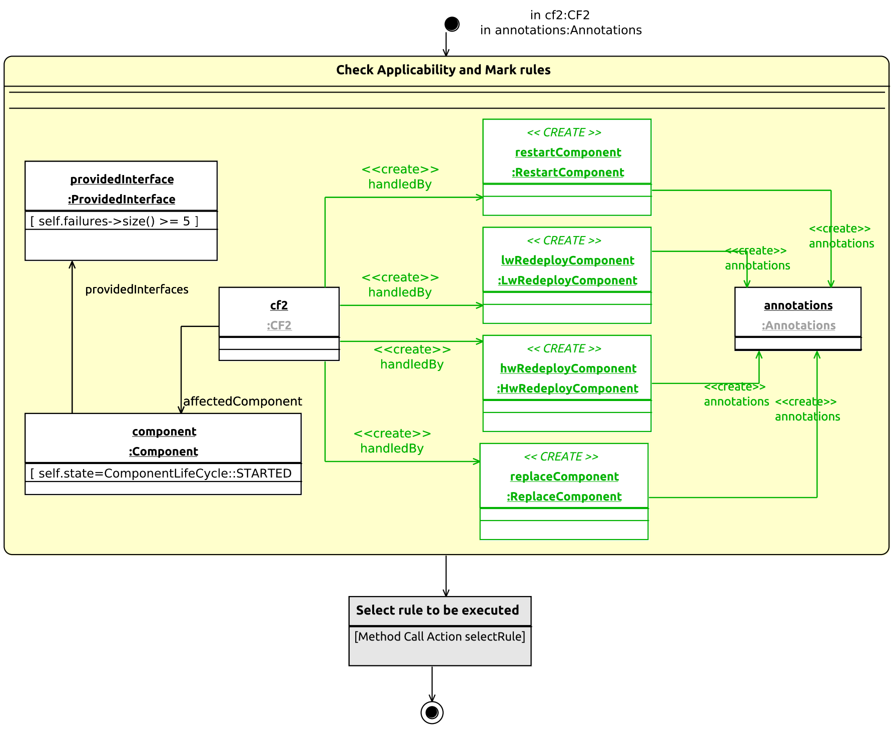


Figure A.2: Plan SD for CF2 issue.

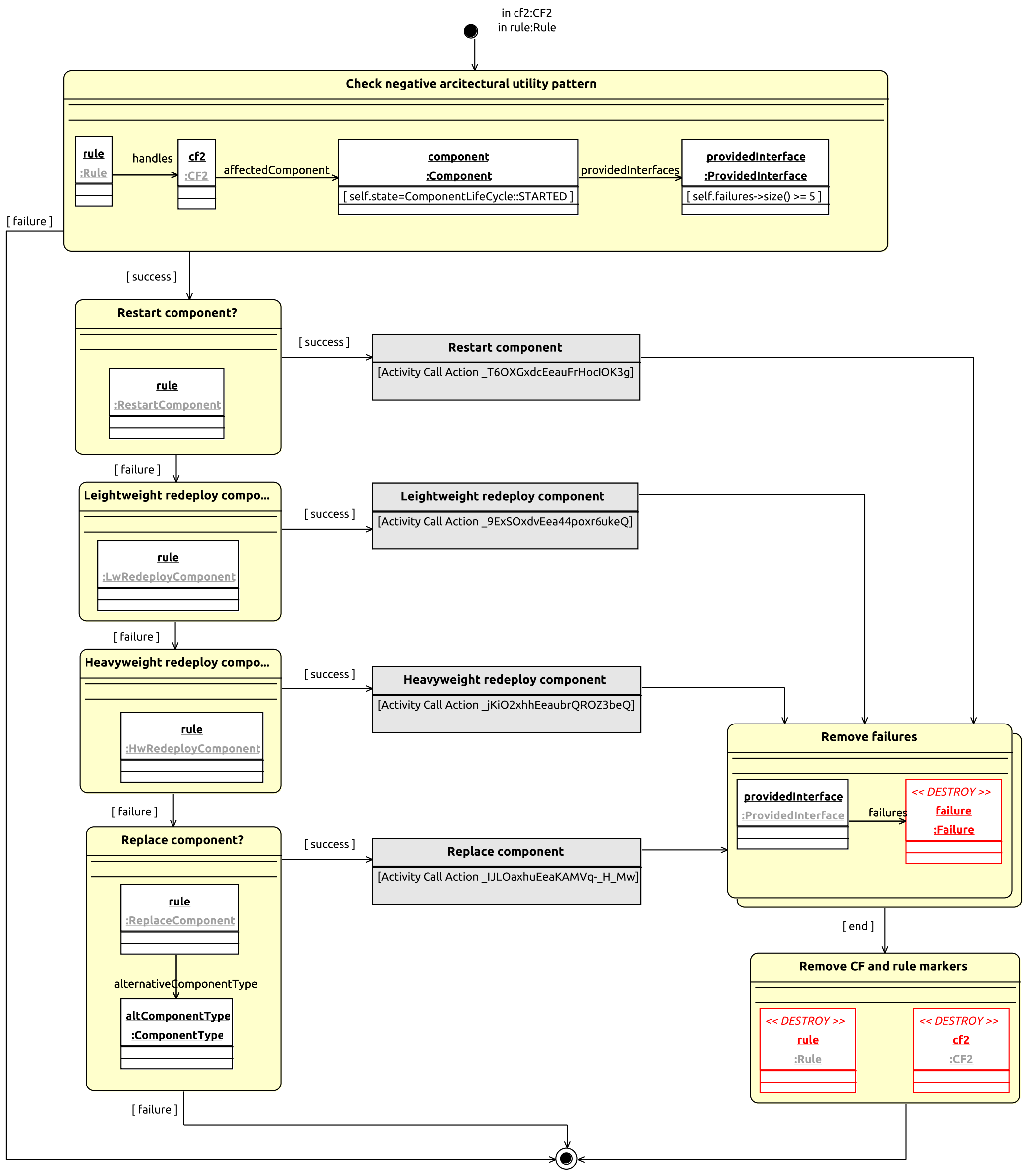

in cf2:CF2
in rule:Rule
Check negative arcitectural utility pattern
rule :Rule
handles
cf2 :CF2
affectedComponent
component :Component
[self.state=ComponentLifeCycle::STARTED]
providedInterfaces
providedInterface :ProvidedInterface
[self.failures->size() >= 5]
[failure]
[success]
Restart component?
rule :RestartComponent
[success]
Restart component
[Activity Call Action _T6OXGxdcEeauFrHocIOK3g]
[failure]
Leightweight redeploy compo...
rule :LwRedeployComponent
[success]
Leightweight redeploy component
[Activity Call Action _9ExSOxdvEea44poxr6ukeQ]
[failure]
Heavyweight redeploy compo...
rule :HwRedeployComponent
[success]
Heavyweight redeploy component
[Activity Call Action _jKiO2xhhEeaubrQROZ3beQ]
[failure]
Replace component?
rule :ReplaceComponent
alternativeComponentType
altComponentType :ComponentType
[success]
Replace component
[Activity Call Action _lJLOaxhuEeaKAMVq-_H_Mw]
[failure]
Remove failures
providedInterface :ProvidedInterface
failures
<< DESTROY >> failure :Failure
[end]
Remove CF and rule markers
<< DESTROY >> rule :Rule
<< DESTROY >> cf2 :CF2

Figure A.3: Execute SD for CF2 issue.

application of an SP entails finding in the model a match for the pattern consisting of the black and red elements. Upon finding a match, the black elements remain unchanged, the red elements are removed, and the green elements are added to the model. Thereby, attribute values of the green and black nodes can be initialized respectively modified. The pattern and the side effect, i.e., the RHS, can be extended with constraints expressed in the *Object Constraint Language* (OCL).

Considering the examples in this chapter, the SDs operate on an architectural RTM. The SD in Figure A.1 uses its input parameter, i.e., the ProvidedInterface element, to find the Component whose providedInterface has five or more exceptions, i.e., matches for a CF2 issue in mRUBiS—see the constraint in the ProvidedInterface element expressed in the OCL. The nac node in the check for failures SP checks for the negative application condition, i.e., existing markings of the same issue (CF2) for the the same Component. If the SD cannot match the pattern in the first SP, the control flow continues along the [failure] edge and terminates, otherwise, the mark failures SP is executed to mark the failure in the RTM via the annotations—see the [success] edge in Figure A.1. The mark failures SP in Figure A.1 creates a new CF2 marker in the object Annotations. CF2 refers to the faulty Component via the affectedComponent relationship. Upon creation, the attribute UtilityDrop of the CF2 element is also set via an OCL expression. The SD in Figure A.1 particularly illustrates how the control flow is branched depending on the success of identifying a match and how constraints extending a pattern are specified.

The Plan SD in Figure A.2 takes the specific CF2 marker as input and creates markers for the applicable rules that can potentially resolve CF2. The rules of four different types are linked to CF2 through the handledBy relationships. The rule markers are also added to the Annotations element. The second action node in the SD is a Call Action node that executes a code snippet to select the rule for execution. After the code is executed, the SD terminates. This SD particularly illustrates how nodes and edges are added to the model and how Call Actions are used to invoke code.

Finally, the Execute SD in Figure A.3 uses its input parameters, i.e., the CF2 marker and the Rule marker, to execute the rule on the Component. The first SP checks for the pattern again, if it cannot be matched for the given inputs, the control flow continues along the [failure] edge and terminates. If the pattern is matched, the control flow continues to match the type of the Rule via executing four SPs. A match in one of the four SPs containing the specific rule type continues along the [success] edge and executes a Call Action node that invokes another SD, e.g., Restart component action node, to execute the corresponding rule. Next, the control flow continues to Remove Failures SP. The SP is cascaded such that it identifies all matches of its pattern, that is, it finds and destroys all Failure elements by navigating the failures relationship from the ProvidedInterface element. Once the Remove Failures SP is executed for all the matches, the Rule and CF2 markers are destroyed, thereby, removed from the RTM in the final SP and the SD terminates. Consequently, upon a successful execution of the SD in Figure A.3, a CF2 issue in mRUBiS is resolved and removed from the RTM. This SD particularly illustrates how nodes and edges are removed from the model and how Call Actions are used to invoke other SDs.

To summarize, SDs enable in-place model transformation, i.e., modifications of models. Since SDs are executable, we use them to operationalize the Analyze, Plan, and Execute activities of our adaptation engine for each specific application example. In the following, we show listings of executing SDs from Java.

A.2 EXAMPLE OF LOADING SDS FROM JAVA

The adaptation engine implementation includes Java code snippets wrapped around the SDM tools including an editor, an interpreter, and a debugger for the SDs. Listing A.1 shows the initialization of the SD environment for `Znn.com` where the SD rules, i.e., files with the .mlsdm extensions, to detect the four potential issues affecting `Znn.com`, i.e., `overBudget`, `lowQuality`, `latency`, and `underUtilized` are loaded.

Listing A.1: Loading SD rules for Analyze, Plan, and Execute in `Znn.com`.

```
        EnvSetUp.initialize();
        /*
        * load rules
        */
        //Analyze
        Activity A_Budget= EnvSetUp
        .getStoryDiagramActivityFromFile("A_Budget.mlsdm");
        Activity A_Quality= EnvSetUp
        .getStoryDiagramActivityFromFile("A_Quality.mlsdm");
        Activity A_Latency= EnvSetUp
        .getStoryDiagramActivityFromFile("A_Latency.mlsdm");
        Activity A_Utilization= EnvSetUp
        .getStoryDiagramActivityFromFile("A_Utilization.mlsdm");

        //Plan
        Activity P_Budget= EnvSetUp
        .getStoryDiagramActivityFromFile("P_Budget.mlsdm");
        Activity P_Quality= EnvSetUp
        .getStoryDiagramActivityFromFile("P_Quality.mlsdm");
        Activity P_Latency= EnvSetUp
        .getStoryDiagramActivityFromFile("P_Latency.mlsdm");
        Activity P_Utilization= EnvSetUp
        .getStoryDiagramActivityFromFile("P_Utilization.mlsdm");

        //Execute
        Activity E_Budget= EnvSetUp
        .getStoryDiagramActivityFromFile("E_Budget.mlsdm");
        Activity E_Quality= EnvSetUp
        .getStoryDiagramActivityFromFile("E_Quality.mlsdm");
        Activity E_Latency= EnvSetUp
        .getStoryDiagramActivityFromFile("E_Latency.mlsdm");
        Activity E_Utilization= EnvSetUp
        .getStoryDiagramActivityFromFile("E_Utilization.mlsdm");
```

We have implemented Monitor activity independently and external to the approaches—see Section 4.3.2. To make the monitoring runtime-efficient, we use a change tracking mechanism enabled by the notification feature of the EMF, which provides notifications about the individual changes of any EMF-based models [391]. Thus, the monitoring activity consumes the EMF events notifying about changes of the RTM—see attaching event listeners to the *architecturalRTM* in Listing A.2.

Listing A.2: Simulation initialization including method calls for Analyze and Plan activities.

```
        /*attach event listener*/
        architecturalRTM.eAdapters().add(new EventListener());
        Annotations annotations = architecturalRTM.getAnnotations();
        /*call the simulator to generate traffic for the ZNN website*/
        simulation.simulateTraffic.generateTraffic(architecturalRTM,interpreter);
        /*validate the simulator*/
        simulation.InitialValidation.validate(architecturalRTM);

        // Analyze
        analyze(interpreter, annotations, A_Budget, A_Quality, A_Latency,
            A_Utilization);

        // Plan

        plan(interpreter, annotations, P_Budget, P_Quality, P_Latency,
            P_Utilization);
```

The implementation of the remaining MAPE activities then consists of Java code snippets that consumes the change events for the architectural RTM. Based on these events, relevant checks are conducted via invoking the SD interpreter to execute the corresponding SDs. Listing A.3 shows excerpt of the *analyze* method that is called in Listing A.2. The code checks if a change event suggests that the pre-conditions of *A_Budget* and *A_Utilization* analyze SDs should be evaluated—see the corresponding SD for *A_Budget* in Figure 6.4.

Listing A.3: Excerpt of Analyze method in `Znn.com`.

```java
private static void analyze(MLSDMInterpreter interpreter, Annotations annotations,
    Activity A_Budget, sActivity A_Quality, Activity A_Latency, Activity
    A_Utilization) throws SDMException {

    /*for each event in queue check which Analyze SD is triggered*/
    while (!EventQueue.EVENTS.isEmpty()) {
        Notification notification = EventQueue.EVENTS.poll();
        Object notifier = notification.getNotifier();
            Object feature = notification.getFeature();

        /*change of Pool serverCount --> check for overBudget or underUtilized
            issues*/
        if(feature==ZnnPackage.Literals.SERVER_POOL__SERVER_COUNT){
                ServerPool serverPool = (ServerPool) notifier;
                Collection<Variable<EClassifier>> parameters = new ArrayList<
                    Variable<EClassifier>>();
                parameters.add(createParameter("serverPool", serverPool));
                parameters.add(createParameter("annotations", annotations));

                /* execute A_Budget SD*/
                interpreter.executeActivity(A_Budget, parameters);
                /* execute A_Utilization SD*/
                interpreter.executeActivity(A_Utilization, parameters);}
```

Finally, once the execution of the adaptation is completed, we compute the overall utility of the RTM, i.e., the architecture, and validate the model to check for any unhandled issue—see Listing A.4. The SD interpreter is used to parse the architectural RTM and check if any adaptation issue exists—see Listing A.2.

Listing A.4: Calling Execute SDs in `Znn.com`.

```java
//Execute
execute(interpreter, allIssues, E_Budget, E_Quality, E_Latency, E_Utilization);
overalUtility=computeOverallUtility(architecturalRTM);
if (!validateArcitectureForIssues(architecturalRTM) ) {
        System.out.print("Issues remaning in the model.");
        /*end of one adaptation loop run*/
```

B

EVALUATION SUPPLEMENT

This chapter presents supplementary technical and evaluation materials for our methodology to train utility-change prediction models and complements the evaluation results in Chapter 8.

B.1 TRAINING AND TESTING

All the datasets are publicly available [see 174]. We selected the regression trees as training model for two reasons: it automatically selects the features to be part of the model and it captured discontinuities and non-linearities. Among various options for decision trees, we used XGB with 10-fold Cross Validation. To validate the results from training, we make different splits of the original training dataset: 70/30, 80/20, and 90/10—see the results of training with these splits in Figure B.1.

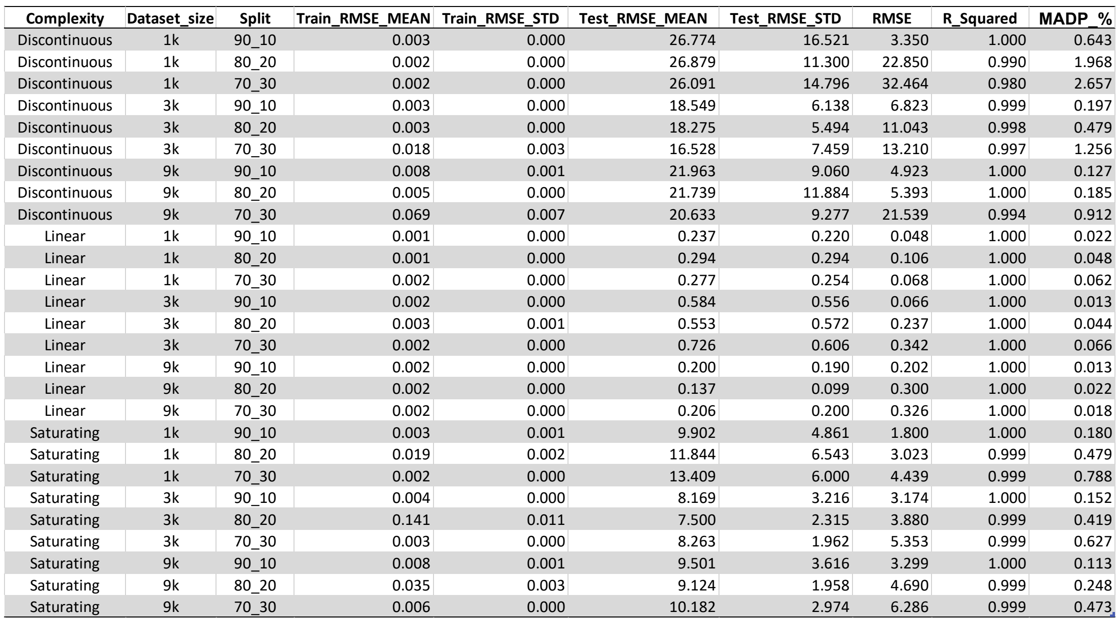
B.2 VALIDATING

For validation we measured the prediction error, i.e., *Mean Absolute Deviation Percent* (MADP), for data that was not used during training. Column "MADP_%" in Figure B.1 shows the results of validating the prediction models using 30%, 20%, and 10% of the original data. We can see that even though 90/10 split shows lower MADP, the difference is very small, i.e., less than 1%. To be pessimistic, we decided to use the 70/30 split (larger prediction error).

XGB model: We first investigated how XGB performed with different dataset sizes under different splits of data between training and validation. Since the results of 70/30 split was better, we adopted this split for the other two method types (GBM and RF). We also trained XGB with other dataset sizes, i.e., 10K and 100k, however, the values for MADP saturated after the 10K dataset—see Figure B.2.

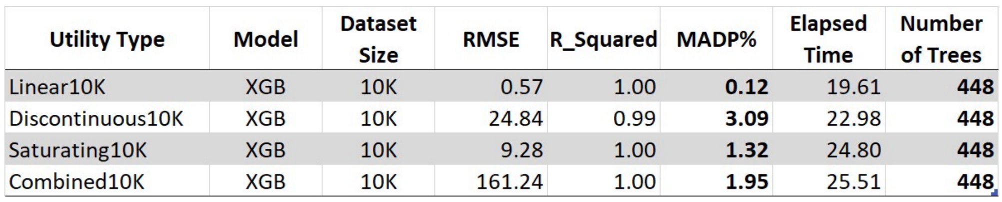
RF model: We investigate if the performance of the RF model would improve with larger dataset sizes in Figure B.3a. We trained a forest with 200 trees to investigate if we could improve MADP without impacting much the execution time, the results did not improve while the training time ("elapsed time" column) increased significantly—see Figure B.3.

GBM model: GBM required the largest number of trees, compared to the other models. For this reason, it also took the longer to be trained as the "elapsed time" column shows. We trained with 10K and 15K trees—see Figure B.4 for the results.



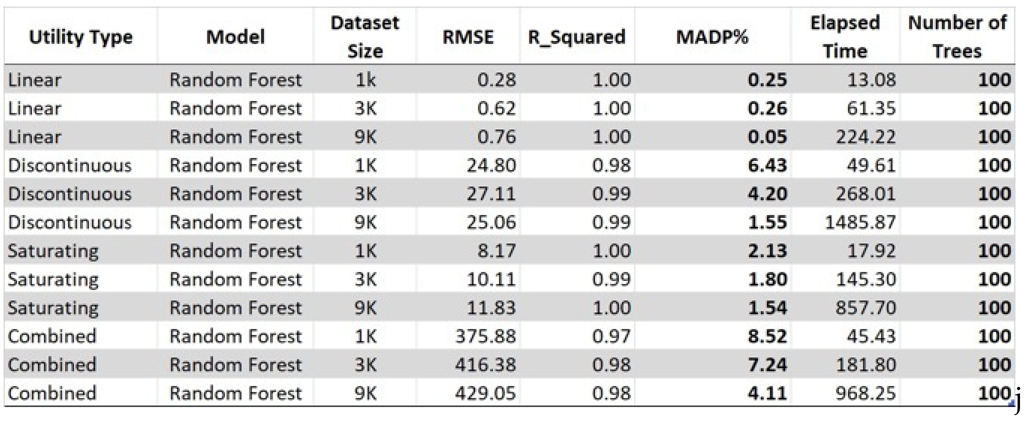
Complexity	Dataset_size	Split	Train_RMSE_MEAN	Train_RMSE_STD	Test_RMSE_MEAN	Test_RMSE_STD	RMSE	R_Squared	MADP_%
Discontinuous	1k	90_10	0.003	0.000	26.774	16.521	3.350	1.000	0.643
Discontinuous	1k	80_20	0.002	0.000	26.879	11.300	22.850	0.990	1.968
Discontinuous	1k	70_30	0.002	0.000	26.091	14.796	32.464	0.980	2.657
Discontinuous	3k	90_10	0.003	0.000	18.549	6.138	6.823	0.999	0.197
Discontinuous	3k	80_20	0.003	0.000	18.275	5.494	11.043	0.998	0.479
Discontinuous	3k	70_30	0.018	0.003	16.528	7.459	13.210	0.997	1.256
Discontinuous	9k	90_10	0.008	0.001	21.963	9.060	4.923	1.000	0.127
Discontinuous	9k	80_20	0.005	0.000	21.739	11.884	5.393	1.000	0.185
Discontinuous	9k	70_30	0.069	0.007	20.633	9.277	21.539	0.994	0.912
Linear	1k	90_10	0.001	0.000	0.237	0.220	0.048	1.000	0.022
Linear	1k	80_20	0.001	0.000	0.294	0.294	0.106	1.000	0.048
Linear	1k	70_30	0.002	0.000	0.277	0.254	0.068	1.000	0.062
Linear	3k	90_10	0.002	0.000	0.584	0.556	0.066	1.000	0.013
Linear	3k	80_20	0.003	0.001	0.553	0.572	0.237	1.000	0.044
Linear	3k	70_30	0.002	0.000	0.726	0.606	0.342	1.000	0.066
Linear	9k	90_10	0.002	0.000	0.200	0.190	0.202	1.000	0.013
Linear	9k	80_20	0.002	0.000	0.137	0.099	0.300	1.000	0.022
Linear	9k	70_30	0.002	0.000	0.206	0.200	0.326	1.000	0.018
Saturating	1k	90_10	0.003	0.001	9.902	4.861	1.800	1.000	0.180
Saturating	1k	80_20	0.019	0.002	11.844	6.543	3.023	0.999	0.479
Saturating	1k	70_30	0.002	0.000	13.409	6.000	4.439	0.999	0.788
Saturating	3k	90_10	0.004	0.000	8.169	3.216	3.174	1.000	0.152
Saturating	3k	80_20	0.141	0.011	7.500	2.315	3.880	0.999	0.419
Saturating	3k	70_30	0.003	0.000	8.263	1.962	5.353	0.999	0.627
Saturating	9k	90_10	0.008	0.001	9.501	3.616	3.299	1.000	0.113
Saturating	9k	80_20	0.035	0.003	9.124	1.958	4.690	0.999	0.248
Saturating	9k	70_30	0.006	0.000	10.182	2.974	6.286	0.999	0.473

Figure B.1: XGB trained with 500 trees over different datasets and data splits.



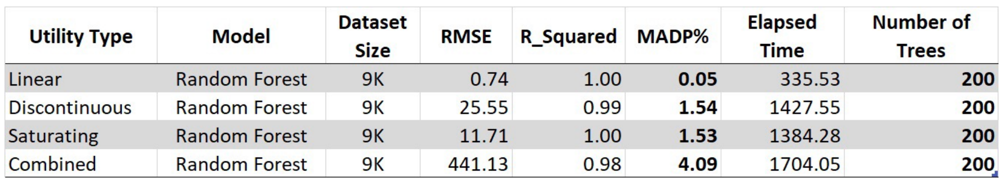
Utility Type	Model	Dataset Size	RMSE	R_Squared	MADP%	Elapsed Time	Number of Trees
Linear10K	XGB	10K	0.57	1.00	**0.12**	19.61	**448**
Discontinuous10K	XGB	10K	24.84	0.99	**3.09**	22.98	**448**
Saturating10K	XGB	10K	9.28	1.00	**1.32**	24.80	**448**
Combined10K	XGB	10K	161.24	1.00	**1.95**	25.51	**448**

Figure B.2: XGB trained with 500 trees with 10K data points.



Utility Type	Model	Dataset Size	RMSE	R_Squared	MADP%	Elapsed Time	Number of Trees
Linear	Random Forest	1k	0.28	1.00	**0.25**	13.08	**100**
Linear	Random Forest	3K	0.62	1.00	**0.26**	61.35	**100**
Linear	Random Forest	9K	0.76	1.00	**0.05**	224.22	**100**
Discontinuous	Random Forest	1K	24.80	0.98	**6.43**	49.61	**100**
Discontinuous	Random Forest	3K	27.11	0.99	**4.20**	268.01	**100**
Discontinuous	Random Forest	9K	25.06	0.99	**1.55**	1485.87	**100**
Saturating	Random Forest	1K	8.17	1.00	**2.13**	17.92	**100**
Saturating	Random Forest	3K	10.11	0.99	**1.80**	145.30	**100**
Saturating	Random Forest	9K	11.83	1.00	**1.54**	857.70	**100**
Combined	Random Forest	1K	375.88	0.97	**8.52**	45.43	**100**
Combined	Random Forest	3K	416.38	0.98	**7.24**	181.80	**100**
Combined	Random Forest	9K	429.05	0.98	**4.11**	968.25	**100**

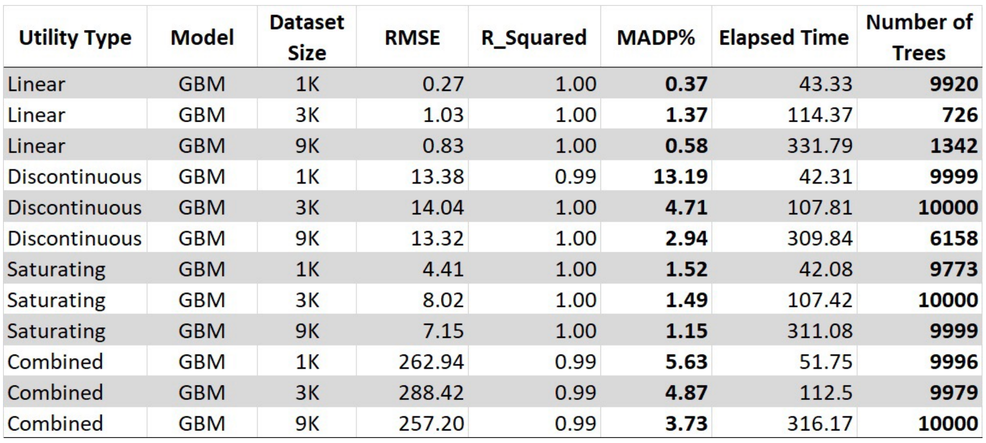
(a) RF across different dataset sizes.



Utility Type	Model	Dataset Size	RMSE	R_Squared	MADP%	Elapsed Time	Number of Trees
Linear	Random Forest	9K	0.74	1.00	**0.05**	335.53	**200**
Discontinuous	Random Forest	9K	25.55	0.99	**1.54**	1427.55	**200**
Saturating	Random Forest	9K	11.71	1.00	**1.53**	1384.28	**200**
Combined	Random Forest	9K	441.13	0.98	**4.09**	1704.05	**200**

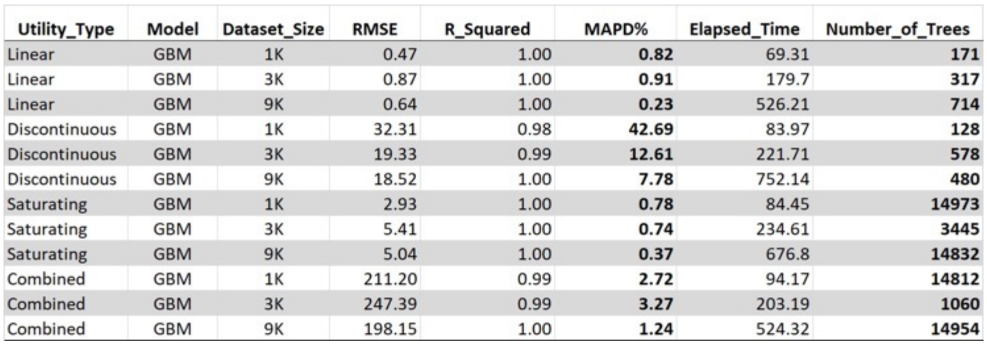
(b) RF with 200 trees.

Figure B.3: RF across different dataset sizes (top) and with 200 trees (bottom).



Utility Type	Model	Dataset Size	RMSE	R_Squared	MADP%	Elapsed Time	Number of Trees
Linear	GBM	1K	0.27	1.00	**0.37**	43.33	**9920**
Linear	GBM	3K	1.03	1.00	**1.37**	114.37	**726**
Linear	GBM	9K	0.83	1.00	**0.58**	331.79	**1342**
Discontinuous	GBM	1K	13.38	0.99	**13.19**	42.31	**9999**
Discontinuous	GBM	3K	14.04	1.00	**4.71**	107.81	**10000**
Discontinuous	GBM	9K	13.32	1.00	**2.94**	309.84	**6158**
Saturating	GBM	1K	4.41	1.00	**1.52**	42.08	**9773**
Saturating	GBM	3K	8.02	1.00	**1.49**	107.42	**10000**
Saturating	GBM	9K	7.15	1.00	**1.15**	311.08	**9999**
Combined	GBM	1K	262.94	0.99	**5.63**	51.75	**9996**
Combined	GBM	3K	288.42	0.99	**4.87**	112.5	**9979**
Combined	GBM	9K	257.20	0.99	**3.73**	316.17	**10000**

(a) GBM with 10K trees.



Utility_Type	Model	Dataset_Size	RMSE	R_Squared	MAPD%	Elapsed_Time	Number_of_Trees
Linear	GBM	1K	0.47	1.00	**0.82**	69.31	**171**
Linear	GBM	3K	0.87	1.00	**0.91**	179.7	**317**
Linear	GBM	9K	0.64	1.00	**0.23**	526.21	**714**
Discontinuous	GBM	1K	32.31	0.98	**42.69**	83.97	**128**
Discontinuous	GBM	3K	19.33	0.99	**12.61**	221.71	**578**
Discontinuous	GBM	9K	18.52	1.00	**7.78**	752.14	**480**
Saturating	GBM	1K	2.93	1.00	**0.78**	84.45	**14973**
Saturating	GBM	3K	5.41	1.00	**0.74**	234.61	**3445**
Saturating	GBM	9K	5.04	1.00	**0.37**	676.8	**14832**
Combined	GBM	1K	211.20	0.99	**2.72**	94.17	**14812**
Combined	GBM	3K	247.39	0.99	**3.27**	203.19	**1060**
Combined	GBM	9K	198.15	1.00	**1.24**	524.32	**14954**

(b) GBM with 15K trees.

Figure B.4: GBM across different dataset sizes.

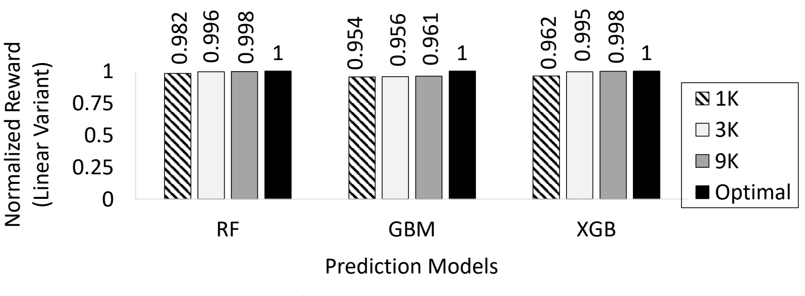
Table B.1: Final prediction models.

Method	**Number of Trees**	**Specific hyper-parameters**

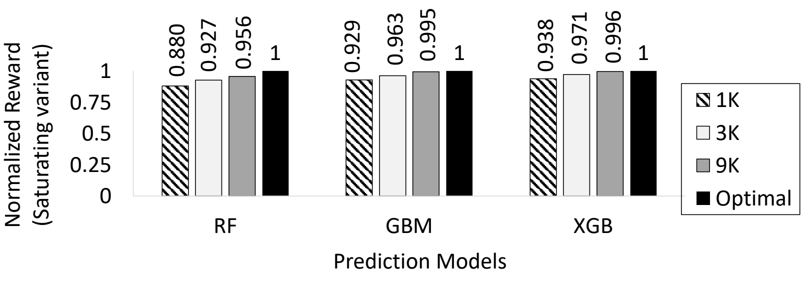
XGB	500	objective="reg:linear"
		base_score=0.5
		early_stopping_rounds=500
		metrics="RMSE"

GBM	15K	distribution="Gaussian"
		interaction.depth=10
		n.minobsinnode=5
		shrinkage=1
		bag.fraction=1

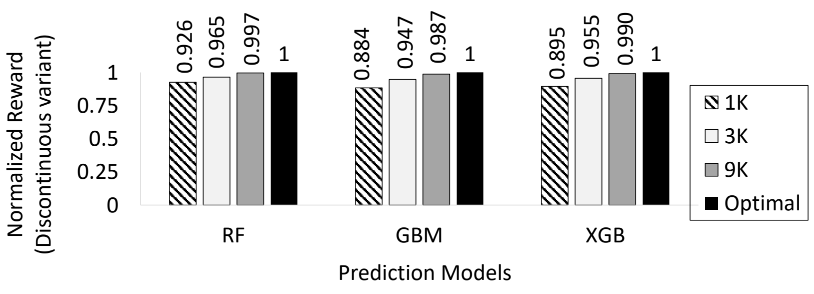
RF	100	node.size=5
		metrics="RMSE"



(a) Linear varinat.



(b) Saturating varinat.



(c) Discontinuous varinat.

Figure B.5: Normalized reward across prediction models for Linear, Saturating, and Continuous variants computed with *DEUG* trace.

B.3 FINAL PREDICTION MODELS

Table B.1 shows the features of the final prediction models. The models are exported to pmml format[1].To evaluate the prediction model performance, we measure the normalized reward of the adaptations applied to the simulator variants and the runtime effort of the models. The normalized reward is calculated according to Equation 8.1 for two scalability scenarios: different datasets sizes and mRUBiS architectures sizes. Prediction models trained with larger dataset sizes acquire larger values of normalized reward. Figure B.5 shows results for the Linear, Saturating, and Discontinuous variants. The results also suggest that selecting the larger datasets, i.e., 9K, is a correct decision. The maximum reward loss of the 9K models compared to the optimal reward belongs to the RF model in the Combined variant (Normalized reward = 0.945) in Figure 8.5. The reward loss in the other variants as shown in Figure B.5 is below 5.5%

1 `http://dmg.org/pmml/v4-0-1/GeneralStructure.html`—accessed 07 March 2023.

C PUBLICATIONS

The contributions of this thesis have been published in peer-reviewed conferences and Journals. In the following, we provide an overview of the publications in the context of the thesis as well as the additional publications.
Publications on Venus

- Sona Ghahremani, Holger Giese, and Thomas Vogel. "Towards Linking Adaptation Rules to the Utility Function for Dynamic Architectures." In: *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. Sept. 2016, pp. 142–143. DOI: `10.1109/SASO.2016.21`
- Sona Ghahremani, Holger Giese, and Thomas Vogel. "Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures." In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 59–68. DOI: `doi:10.1109/ICAC.2017.35`
- Sona Ghahremani, Holger Giese, and Thomas Vogel. "Improving Scalability and Reward of Utility-Driven Self-Healing for Large Dynamic Architectures." In: *ACM Trans. Auton. Adapt. Syst.* 14.3 (Feb. 25, 2020), 12:1–12:41. ISSN: 1556-4665. DOI: `10.1145/3380965`

Publication on Learning Methodology

- Sona Ghahremani, Christian M. Adriano, and Holger Giese. "Training Prediction Models for Rule-Based Self-Adaptive Systems." In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*. Sept. 2018, pp. 187–192. DOI: `10.1109/ICAC.2018.00031`

Publications on Evaluation Methodology

- Sona Ghahremani and Holger Giese. "Performance Evaluation for Self-Healing Systems: Current Practice & Open Issues." In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. June 2019, pp. 116–119. DOI: `10.1109/FAS-W.2019.00039`
- Sona Ghahremani and Holger Giese. "Evaluation of Self-Healing Systems: An Analysis of the State-of-the-Art and Required Improvements." In: *Computers* 9.1 (1 Mar. 2020), p. 16. ISSN: 2073-431X. DOI: `10.3390/computers9010016`. (Visited on 02/08/2022)

Publication on HypeZon

- Sona Ghahremani and Holger Giese. "Hybrid Planning with Receding Horizon: A Case for Meta-self-awareness." In: *2021 IEEE International Conference on Autonomic*

Computing and Self-Organizing Systems Companion (ACSOS-C). 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). Sept. 2021, pp. 131–138. DOI: `10.1109/ACSOS-C52956.2021.00045`

Additional publications
In parallel to this thesis, I have been involved with two additional lines of research that resulted in the following publications in the context of *learning in collective adaptive systems* and *history-aware self-adaptation*.

- Mirko D'Angelo, Simos Gerasimou, Sona Ghahremani, Johannes Grohmann, Ingrid Nunes, Evangelos Pournaras, and Sven Tomforde. "On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework." In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). May 2019, pp. 13–24. DOI: `10.1109/SEAMS.2019.00012`

- Mirko D'Angelo, Sona Ghahremani, Simos Gerasimou, Johannes Grohmann, Ingrid Nunes, Sven Tomforde, and Evangelos Pournaras. "Learning to Learn in Collective Adaptive Systems: Mining Design Patterns for Data-driven Reasoning." In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). Aug. 2020, pp. 121–126. DOI: `10.1109/ACSOS-C51401.2020.00042`

- Lucas Sakizloglou, Sona Ghahremani, Thomas Brand, Matthias Barkowsky, and Holger Giese. "Towards Highly Scalable Runtime Models with History." In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '20. New York, NY, USA: Association for Computing Machinery, Sept. 18, 2020, pp. 188–194. ISBN: 978-1-4503-7962-5. DOI: `10.1145/3387939.3388614`

- Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. "A Scalable Querying Scheme for Memory-Efficient Runtime Models with History." In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. New York, NY, USA: Association for Computing Machinery, Oct. 16, 2020, pp. 175–186. ISBN: 978-1-4503-7019-6. DOI: `10.1145/3365438.3410961`

- Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. "Incremental Execution of Temporal Graph Queries over Runtime Models with History and Its Applications." In: *Softw Syst Model* 21.5 (Oct. 1, 2022), pp. 1789–1829. ISSN: 1619-1374. DOI: `10.1007/s10270-021-00950-6`

INDEX