Fachgebiet Software and Embedded Systems Engineering
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

Safe, Intelligent and Explainable Self-Adaptive Systems

vorgelegt von
Verena Klös,
Master of Science in Computer Science

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Andreas Vogelsang
Technische Universität Berlin

Gutachter: Prof. Dr. Sabine Glesner
Technische Universität Berlin

Prof. Dr. Sibylle Schupp
Technische Universität Hamburg

Prof. Dr. Kurt Schneider
Leibniz Universität Hannover

Tag der wissenschaftlichen Aussprache: 10.03.2020

Berlin, 2021

Abstract

Intelligent cyber-physical systems, such as self-driving cars, smart homes or e-health solutions, will increasingly influence our daily lives. They will deal with increasingly uncertain and changing environments and simultaneously must adhere to strict safety requirements. In addition, we need to trust those systems, as we will hand over control on our daily life to them. This increasing list of non-functional requirements makes the design of intelligent cyber-physical systems (CPS) a challenging task.

In this thesis, we advance research on intelligent CPS by introducing intelligent self-adaptive systems that autonomously evolve their adaptation logic in response to changes in the system structure, their environment and their goals. We make their widespread integration possible, by introducing a safe design framework, based on a novel methodology. Our framework enables the integrated design and formal verification of intelligent and explainable self-adaptive systems. Our key idea is to combine a resource-efficient process for self-adaptation with dynamic evolution of the adaptation logics and continuous verification activities. To obtain trust in those systems, we additionally collect structured runtime knowledge to build an explanation basis for autonomous decisions.

Our main contributions are an efficient and comprehensible rule- and distance-based adaptation process, a quantitative and context-dependent goal model that provides the basis for our adaptation process, a resource-efficient runtime evolution of adaptation logics that combines a continuous evaluation and observation-based optimization, of adaptation rules and a stochastic search-based learning of new comprehensible adaptation rules, and a continuous verification methodology that is based on a formalization of our rule- and distance-based adaptation process in timed automata.

We have implemented our framework and evaluated its applicability and performance on three case studies from different domains, namely a smart temperature control system, an autonomous drone delivery system and a self-organizing production system.

With our framework, we support the design of systems that are flexible enough to deal with dynamically changing operation contexts, and, at the same time, provide safety assurances and explainability of their autonomous decisions. It is the only approach that provides an integrated solution to this crucial research topic.

Zusammenfassung

Intelligente cyber-physikalische Systeme, wie z.B. autonom fahrende Autos, Smart Homes oder E-Health Systeme, werden unser zukünftiges leben beeinflussen. Sie müssen mit zunehmend unsicheren und sich verändernden Umgebungen umgehen und dabei strikte Sicherheitsanforderungen erfüllen. Ihnen die Kontrolle über unser Leben zu überlassen, erfordert Vertrauen. Diese steigende Anzahl von zusätzlichen Anforderungen macht den Entwurf von intelligenten cyber-physischen Systemen (CPS) zu einer Herausforderung.
Als Lösung für diese Herausforderungen führen wir intelligente selbst-adaptive Systeme ein, die ihre Adaptionslogik autonom an Änderungen in ihrem System, der Umgebung und ihren Zielen anpassen. Wir präsentieren eine neue Methodik zum integrierten Entwurf und zur formalen Verifikation von intelligenten und erklärbaren selbst-adaptiven Systemen und betten unsere Methodik in ein sicheres Framework ein. Unsere Kernidee ist die Kombination eines ressourcensparenden Selbstadaptionsprozesses mit der dynamischen Anpassung der Adaptionslogik zur Laufzeit und einem kontinuierlichen Verifikationsprozess. Zur Steigerung des Vertrauens in diese Systeme speichern wir Laufzeitdaten über autonome Entscheidungen und stellen diese strukturiert zur Verfügung.
Unsere Hauptbeiträge sind ein effizienter und verständlicher regelbasierter Adaptionsprozess mit einer neuartigen Distanzmetrik als Entscheidungsgrundlage, ein quantitatives und kontextabhängiges Zielmodel, das die Basis für unseren Adaptionsprozess bildet, eine ressourcensparende Anpassung der Adaptionslogik zur Laufzeit, die eine kontinuierliche Evaluierung und beobachtungsbasierte Optimierung von Adaptionsregeln mit einem simulationsbasierten Lernen neuer Adaptionsregeln kombiniert, und eine kontinuierliche Verifikationsmethodik, die auf einer Formalisierung unseres regel- und distanzbasierten Adaptionsprozesses in zeitbehafteten Automaten beruht.
Wir haben die Anwendbarkeit und Leistungsfähigkeit unseres Frameworks mit drei Fallstudien aus verschiedenen Domänen evaluiert: einem intelligenten Temperaturregler, einem autonomen Drohnenlieferservice und einem selbstorganisierenden Produktionssystem.
Mit unserem Framework unterstützen wir den Entwurf von Systemen, die einerseits flexibel genug sind, um mit sich dynamisch ändernden Umgebungen umzugehen, und, andererseits Sicherheitszusicherungen und Erklärungen ihrer autonomen Entscheidungen geben können.

Danksagung

Diese Arbeit wäre ohne die Unterstützung einiger besonderer Menschen nicht zustande gekommen. Zuerst möchte ich Prof. Dr. Sabine Glesner für ihre jahrelange Unterstützung danken. Sie hat mir schon frühzeitig eine Promotion nahegelegt, mir einen wunderbaren Rahmen an ihrem Fachgebiet geboten und immer an mich geglaubt. Mein Dank gilt auch meinen Zweitgutachtern, Prof. Dr. Sibylle Schupp und Prof. Dr. Kurt Schneider, deren Fragen und Anregungen mir sehr geholfen haben, die Bedeutung der Erklärbarkeit für meine Systeme zu erkennen und der Arbeit damit den letzten Schliff zu geben. Ich danke ihnen für ihr Interesse an meiner Arbeit und ihre hilfreichen Kommentare.

Ich danke meinen wunderbaren Kollegen für ihre Unterstützung, ihre Freundschaft und ihr stets offenes Ohr, sowie für die wunderbare Gemeinschaft in der Arbeitsgruppe. Mein besonderer Dank gilt Dr. Thomas Göthel und Dr. Paula Herber für die inspirierenden Diskussionen, die gute Zusammenarbeit und stete Unterstützung, auch in der schwierigen Zeit des Zusammenschreibens. Sie haben diese Arbeit von den ersten Ideen bis zur letzten Seite begleitet. Besonders die Zusammenarbeit mit Dr. Thomas Göthel hat meine Arbeit enorm voran gebracht und zahlreiche Veröffentlichungen ermöglicht. Ich danke ihm dafür, dass er stets bereit war über meine Ideen zu diskutieren, diese gemeinsam reifen zu lassen und kurzfristige Veröffentlichungspläne zu unterstützen. Danke für die tolle Zeit!

Diese Arbeit wäre ohne die Hilfe meiner Familie nicht zustande gekommen. Ich danke meinen Eltern für ihre großartige Unterstützung und ihre Geduld. Sie sind immer für mich da, wenn ich sie brauche - Danke für alles! Auch meiner Tochter Clara danke ich für Ihre Geduld und dafür, dass sie immer an mich glaubt.

Contents

1 Introduction

We live in a decade of self-driving cars, smart homes and the internet of things. We rely on personal assistants like Siri and Alexa and trust autonomous cancer classification systems that perform better than human experts. All these intelligent cyber-physical systems (CPS) are ubiquitous in our daily lives. They solve complex problems even experts do not fully understand. This thesis is motivated by one of the most challenging research questions in the area of autonomous intelligent systems: *How can we ensure that "increasingly capable AI systems are robust and beneficial"?* [Fut15, Nev18]

In this thesis, we focus on intelligent CPS that autonomously adapt themselves to changes in the system, their environment and goals. We interpret robustness of such self-adaptive systems as *systems being able to maintain their goals in ever-changing operational contexts*. For a definition of beneficial systems, we follow Stuart Russels definition: *Machines are beneficial to the extent that their actions can be expected to achieve our objectives* [Rus18]. Hence, it is crucial to ensure that these systems behave as intended, and to obtain trust in the correctness and quality of autonomous decisions, e.g. by providing comprehensible explanations [SSP+17]. Thus, we refine our research question: **How can we design intelligent self-adaptive systems that are flexible enough to cope with ever-changing operational contexts and how can we ensure safety and explainability of their autonomous decisions?**

The aim of this thesis is to provide a design framework for intelligent self-adaptive systems that are able to achieve **safe and robust autonomous adaptation decisions** in ever-changing operational contexts, and to provide **comprehensible explanations** for these decisions.

We require our approach to fulfill the following criteria:

1. **Continuous learning of (timed) adaptation logics** To maintain their goals in ever-changing operational contexts, we require intelligent self-adaptive systems to continuously evaluate their adaptation logic and to learn adaptation logics that enable them to cope with run-time changes in environment, system topology and capabilities, and goals. To account for time constraints and to enable proactive adaptation in time, we require the adaptation logic to include the latency of adaptation actions.

2. **Independence between adaptation logics and system goals** Continuous learning of adaptation logics should be robust with respect to dynamic goal changes, i.e. the learning results should still be applicable after goal changes. This means that the expected effect of adaptations and its evaluation w.r.t. the system goals have to be strictly separated. Thus, the effect must not directly refer to system goals. However, if new parameters become restricted by the goals or previously restricted variables become unrestricted, the adaptation logic still has to be updated via learning.

3. **Continuous analysis of safety properties** Our approach should enable verification during the design process of self-learning self-adaptive systems to detect errors at early design stages. In addition, it should include run-time verification procedures to enable continuous evaluation of safety requirements.

4. **Explainability of autonomous decisions** Knowledge models form the basis for autonomous decisions. To achieve explainability, we require autonomous decisions to be explainable based on their underlying decision basis. Furthermore, all models have to be human readable and comprehensible.

5. **Resource-efficiency** Self-adaptation and learning should be efficient in terms of time and computational overhead to be applicable in real-time and embedded devices where resources, such as computation time, energy, and memory, are restricted.

In our work, we develop a framework that enables the integrated design and formal verification of intelligent self-adaptive systems. Our key idea is fourfold: (a) We base our adaptation process on novel **adaptation rules that explicitly encode timed effect expectations** on the environment. (b) We continuously **evaluate, optimize and learn** comprehensible timed adaptation rules at run-time. (c) We apply **online and offline verification** to ensure that learning results do not compromise important system properties, e.g. safety, before actually changing the adaptation rule base. (d) We provide an explanation basis that enables **traceability and explainability of autonomous adaptation and learning decisions**.

In more detail, our framework comprises the following main contributions:

1. **Rule- and distance-based adaptation process:** Our key idea is to explicitly encode timed effect expectations on environment and system parameters within the adaptation rules and to evaluate their contribution w.r.t. the goals separately to achieve modularity and reusability in the context of dynamic goal changes. To this end, we evaluate the expected system state after adaptation and apply a novel notion of distance between a system state and the goals. We use a condition-action-effect structure for our timed adaptation rules that supports the comprehensibility of autonomous decisions. With this solution, we achieve a) flexible

and comprehensible adaptation logics, b) efficient adaptation planning, and c) robustness w.r.t. dynamic goal changes.

2. **Quantitative and context-dependent goal model:** We propose a hierarchical goal-model to encode context-dependent adaptation goals, e.g. setpoints or optimisation objectives, and their dependencies at run-time. We provide a modular distance evaluation that enables us to quantify the context-dependent achievement of goals during analysis and planning of adaptations. Thus, our quantitative goal-model provides a basis to find an optimal trade-off between multiple, possibly conflicting context-dependent goals.
   We have published a preliminary version of our goal model in [KGLG17] and an extended version in [KGG18d].

3. **Resource-efficient runtime evolution of adaptation logics:** We propose a novel approach for run-time evolution of timed adaptation rules. It combines a continuous accuracy evaluation, an optimization of adaptation rules and a stochastic search-based learning of new comprehensible rules. To enable online learning of new adaptation rules, we apply a genetic algorithm that uses a model simulation for the fitness evaluation of adaptation options.With simulation-based learning, we avoid the costs and risks of active exploration in the real system. We define a stepwise rule generalization process to profit from learning results in similar situations. We thereby achieve flexibility and are able to resolve uncertainties about the operational context and the effect of adaptations at run-time. We store structured analysis and learning results to provide processed data for explaining autonomous rule learning.

4. **Continuous verification methodology:** We integrate verification activities in the design process and in the evolution process at run-time to ensure safety of dynamically evolving self-adaptive systems. We combine resource intensive formal verification and lightweight safety monitoring. The former is applied at design time to verify the correctness of an initial set of adaptation rules w.r.t. the expected environment behavior, and, additionally, at runtime, to verify that changes to the adaptation rules do not compromise important properties (e.g. safety). The latter is used to cope with the inherent uncertainty concerning the behavior of the environment by continuously monitoring and validating the satisfaction of safety properties at runtime. To enable formal verification, we provide a formalization of our timed adaptation rules and our proposed rule- and distance-based adaptation process in timed automata.
   We have published our formal verification approach in [KGG16].

We have published previous versions of our overall framework in [KGG15, KGG18b, KGG18c]. In [KGG15], we have presented the general architecture and adaptation process together with our requirements on knowledge models that are used within the framework. In [KGG18b], we have presented our run-time evolution with a focus on

online learning of new adaptation rules. There, as well as in [KGG18c], we have also discussed the comprehensibility and explainability of our framework.

With our approach, we achieve **robustness** of intelligent self-adaptive systems by applying continuous optimization and learning of comprehensible timed adaptation rules at run-time. We enable the design of **beneficial** systems with two means: a) we enable systems to continuously verify a set of given objectives by integrating dedicated verification processes, and b) we provide an explanation basis to obtain **trust** in the **correctness** and quality of autonomous decisions.

In summary, **our approach has the following advantages**: Our combination of resource-efficient rule-based adaptation and on-line evolution of adaptation rules provides **flexibility for uncertain environments**, while **reducing costly learning to a minimum** that can also be moved to external servers. Explicit timing information for adaptation rules allow for latency-aware proactive adaptation and for the application in real-time systems. We cope with dynamic goal changes due to the modular structure and distance evaluation of our goal model. These solutions enable our systems to **maintain their system goals in ever-changing operational contexts**, thus being robust w.r.t. our definition. We **ensure that our self-learning self-adaptive systems behave as intended** by embedding formal analysis into the design process and into the evolution process for the adaptation logic. We **provide an explanation basis** for autonomous adaptation and evolution decisions to obtain trust in their correctness. We evaluate our framework with three case studies from different domains, namely a smart temperature control system, an autonomous drone delivery system, and a self-organizing production system.

This thesis is structured as follows. In Chapter 2, we define relevant terms and introduce concepts that are necessary to understand this thesis. We discuss related work in Chapter 3. Afterwards, we provide an overview of our approach in Chapter 4 and introduce our smart temperature control system that is also used as running example. In Chapter 5, we describe our adaptation process and our design of the knowledge models. We describe our goal model that plays a central role in our adaptation process in Chapter 6. In Chapter 7, we present our runtime evolution of adaptation rules and in Chapter 8, we describe our verification approach. Afterwards, we describe our implementation and discuss experimental results in Chapter 9. In Chapter 10, we conclude our thesis and outline future work.

2 Background

In this chapter, we provide preliminaries that are the foundation of this thesis. We start with a general introduction of self-adaptive systems. Afterwards, we describe the MAPE-K feedback loop, which is a prominent reference architecture for self-adaptive systems and which we use as a basis for our framework. In the following, we introduce the modeling language UPPAAL Timed Automata and how it enables formal verification with model checking. We also provide the necessary preliminaries on model checking and temporal logics. Within this thesis, we use timed automata for formal verification and provide a formalization of our adaptation layer in timed automata. Afterwards, we introduce the modeling language SystemC, its simulation semantics and briefly describe the *SystemC to timed Automata Transformation Engine* which we use for our formal verification. We use SystemC for our runtime models and provide an implementation of our adaptation layer in SystemC. In the end, we explain the basic principles of genetic algorithms, which we use for rule learning at runtime, and classification rule learning, which we employ for runtime optimization of adaptation rules.

2.1 Self-Adaptive Systems

Self-adaptive systems are software systems that are able to adapt their behavior to ensure safety and correctness even in case of unpredicted runtime changes in the operational environment, the system topology (e.g. changing components or services) or the system goals. To detect critical changes, a self-adaptive system continuously monitors and analyzes itself, the environment, and the current degree of its goal achievements. Based on the analysis results, the system autonomously decides whether and how to adapt itself to achieve the system goals (again). If goal satisfaction can be quantified with some utility function, the algorithm searches for the best available option. This behavior was inspired by feedback control systems that control a physical target system in order to keep its output as close as possible to a reference value by controlling its input. Feedback control uses the error between measured output and reference value to calculate the new input based on some control laws. To enable the systematic design of self-adaptive systems, the integration of a similar feedback loop into the adaptation software was proposed and several reference architectures were developed (see Chapter 3 for an overview). To adopt

feedback loop control for software systems, a self-adaptive system consists of a managing part that is responsible for the adaptation, and a managed part that corresponds to the target system. The latter is responsible for the functional behavior of the system. System goals serve as reference value and decision algorithms serve as control laws. As it is often not possible to anticipate all possible runtime changes and system states at design time, the decisions on whether and how to adapt are based on runtime knowledge. To this end, available control parameters, like system parameters (e.g., speed of a robot, or parameters for calculating the necessary flow temperature for heating), the system structure (e.g. number of active servers in a web application) or behavioral modes[1] (e.g., an eco mode, a frost-protection mode of a heating, or a cruise control mode of a car) are identified at design time. In addition, adaptation operations on these parameters (e.g., parameter adjustment, restructuring methods, or mode switch routines) are defined. This knowledge is encoded in a suitable and processable way in the system, e.g. as rules or executable routines, together with a possibility to evaluate the expected outcome of applying such an operation, e.g. in form of models or expected rewards. These adaptation parameters, adaptation operations and evaluation models, as well as the autonomous decision algorithm build the *adaptation logics* of a self-adaptive system.

Adaptation Logics

In the literature, different approaches for designing the adaptation logics exist. Existing approaches can be classified according to the following aspects:

- *reactive vs. proactive adaptation*
  Reactive adaptation means that the system adapts as a *reaction* to a situation that is not conform with the system goals. In this case, some goal is already violated and the adaptation algorithm chooses an adaptation operation (or a series of operations) to re-establish goal conformance.

  In contrast, proactive adaptation approaches rely on a prediction model on the future environment behavior. They continuously evaluate whether the system will violate the system goals in the future and choose an adaptation operation to prevent this. If the prediction is accurate, these approaches perform better than reactive approaches in case of adaptation operations that cannot change the system behavior instantaneously but take some time. In this case, adaptation can be planed ahead to avoid latencies.

- *parameter (and mode) adaptation vs. architecture-based adaptation*
  Depending on the kind of parameters that should be changed by adaptation, different aspects have to be encoded in the adaptation logics. For parameter (and mode) adaptation, adaptation operations specify how parameters can be changed (e.g. step size, minimal and maximal values, prerequisites). These

[1] with behavioral modes we refer to predefined configurations for a specific behavior

operations can be easily encoded in rules, which are sometimes also referred to as tactics (e.g., in [GCH+04, MCGS15], that consists of a precondition (e.g., `valueOf(p1)` $\geq$ minValue $\wedge$ `prerequisitesFulfilled()`) and the operation (e.g., `increaseValueOf(p1)`). Mode changes can be encoded in similar rules by using mode switching routines as operations.

For architectural changes that can be freely chosen (i.e. the system does not have to stick to predefined architectural variants, which could be easily encoded as modes), adaptation operations are usually more complex and may also have side effects that influence the prerequisites of other operations. To cope with that, architecture-based adaptation usually uses different models that specify allowed architectural changes (e.g., based on meta models) and that allow for evaluating changes before executing them in the system. These models are used at runtime to guide the adaptation. Thus, they are usually referred to as *models at runtime*.

- *offline vs online planning*
  To choose between different applicable adaptation operations (i.e., their prerequisites are satisfied), different adaptation planning approaches exist. They can be roughly classified into offline approaches that construct a set of adaptation rules at design time and quickly choose between those rules at runtime, and online approaches that search for an optimal adaptation at runtime. Offline approaches are usually rule-based, whereas online approaches are based on model simulations to evaluate the effect of different adaptation operations. Depending on the used formalism and the size and complexity of the models, this simulation can take some time and needs sufficient memory space. Examples are timed automata (e.g., used in [IW14]) or probabilistic models such as Markov Decision Procedures (e.g., used in [MCGS15]) or stochastic multiplayer games (e.g., used in [CGSP15]). For both, online and offline approaches, reinforcement learning or evolutionary approaches are often used. They learn the expected utility of each adaptation operation and use this value as heuristic. As these utility values are only representative for the observed situations, they cannot be reused if the environment behavior changes, or the utility function changes due to goal changes.

A detailed discussion on different adaptation approaches can be found in Chapter 3.

In this thesis, we provide a novel definition of adaptation rules that contain a relative definition of the expected effect on observable parameters such that the expected effect can be evaluated in the current context without the need for extensive model simulation or utility exploration. Furthermore, the effect expectations are separated from the system goals and are still valid after dynamic goal changes.

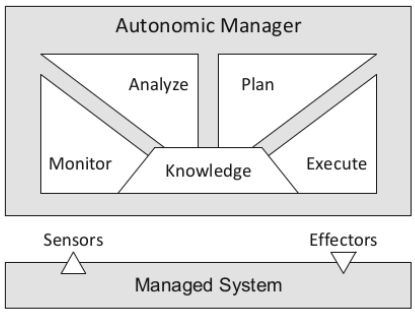


Figure 2.1: MAPE-K Loop [KC03]

2.2 MAPE-K Framework

The MAPE-K feedback loop is a very popular reference framework for self-adaptive systems. It was initially proposed as an IT management solution by IBM in 2003 [KC03, IBM04]. The vision was to achieve self-organization and -optimization by dividing the system into *autonomic* elements (such as databases, web servers, or physical servers) that cooperate with other autonomic elements to accomplish system-level goals. Each autonomic element consists of an autonomic manager and a managed element (cf. Fig. 2.1). The autonomic manager continuously monitors and analyzes the behavior of the managed element, compares it to the objectives, and plans and executes actions to change the behavior of the manged element if necessary. These activities are driven by knowledge about the managed element and by the individual objectives of the autonomic element. Individual objectives and interactions between autonomic elements have to support the desired system-level behavior. The term *autonomic* was chosen to express that the elements have some self-x properties (e.g. self-configuration, self-healing,..) but the changes chosen by the manager can be predefined at design time. Thus, these managers do not have to be able to make *autonomous* decisions in the sense of reasoning or using artificial intelligence.

The behavior of the autonomic manager is represented as a control loop consisting of the four activities monitor, analyze, plan, execute. Based on the initial letters of the activities and the underlying knowledge, this architecture is typically referred to as the MAPE-K loop. The monitor phase of this loop collects information from the sensors provided by the managed elements and from its operational environment. Based on this data, the analyze phase evaluates the situation and determines any anomalies or problems. If a problem is detected, the plan phase generates an adaptation plan to solve it. Finally, the execute phase applies the generated adaptation plan on the actual system. All phases share a common knowledge base that contains knowledge about the system and its context, e.g. the software architecture, operational environment, hardware infrastructure and individual goals. The MAPE-K reference architecture has the advantage that it offers a clear structure, which helps to focus on the design of each

activity, that it is easy to understand, and general enough to capture different self-x systems and their different planning strategies (from static rules to autonomous strategy inference). Thus, it is was adopted and extended by many researchers. More details about the advantages of using explicit feedback loops in self-adaptive systems, as well as a discussion of different examples for feedback loops, are given by [BSG+09].

We use this popular architecture as a basis for our framework. We define an explicit structure of the knowledge base and define suitable knowledge representations for our criteria. Furthermore, we add a second layer above the manager layer to enable runtime evaluation and the evolution of adaptation logics.

In the following, we introduce timed automata as modeling mechanisms and the UPPAAL toolsuite for modeling, simulation and verification of timed automata. We use UPPAAL Timed Automata as formal modeling language to support our simulation-based learning and our formal verification of newly learned rules. In our verification approach we use the UPPAAL model checker and define important properties for self-adaptive systems in the temporal logic TCTL. Thus, we also give a short introduction on model checking in general and a detailed introduction on temporal logics.

2.3 Modeling and Formal Verification with UPPAAL Timed Automata

UPPAAL [BDL04] is a tool suite for modeling, simulation, and verification of timed automata (TA). TA extend finite automata by a set of real-valued clocks $C : \mathbb{R}_{\geq 0}$ that start with an initial value of zero and are uniformly advanced [BY04]. Transitions and the duration of stay in a location depend on clock constraints, which are conjunctive formulas of atomic constraints of the form $x \sim n$ or $x - y \sim n$ with $x, y \in C, \sim \in \{\geq, \geq, =, \leq, \leq\}$ and $n \in \mathbb{N}$. Clock constraints on transitions (*guards*) *enable* progress when satisfied, whereas clock constraints in locations (*invariants*) *enforce* progress by restricting the time in a location. During a transition, clocks can be reset to zero. Concurrent processes are modeled as networks of communicating TA executed with an interleaving semantics. They are well-suited to model communicating real-time processes, provide a modular structure and a formal semantics.

Language Elements

Timed automata (TA) consist of *locations*, *transitions* that connect locations, and clocks. The UPPAAL modeling language *UPPAAL Timed Automata* extends TA with e.g., bounded integer variables, binary and broadcast channels, and urgent (∪) and committed (c) locations. Binary channels enable a blocking one-to-one synchronization between sender (?) and receiver (!), whereas broadcast channels enable non-blocking one-to-many synchronization. Channels can also be marked as urgent to enforce synchronization as soon as possible. Urgent and committed locations are used to model locations where

no time may pass. Leaving a committed location has priority over non-committed locations. Furthermore, UPPAAL provides a C-like action-language that can be used to express guards and updates on variables in form of method calls at transitions. Clocks and variables can be declared as global or local within a template. To model similar components or processes of a system, UPPAAL allows to model parameterized templates of single timed automata. With this, different instants of an automaton can be defined in the system declaration. UPPAAL models are represented as XML-files, which can be easily manipulated, e.g for verification experiments with different parameter values.

To illustrate the main modeling concepts, we use a small example UPPAAL timed automaton (UTA) as shown in Fig. 2.2. The initial location is marked by ◎. The outgoing edge of this location is labeled with `request?` to enable receiving on this channel for synchronization with a concurrent timed automaton. `ack!` denotes sending on the respective channel. `x` is a clock variable that is first set to zero and then used in two clock constraints: the invariant `x <= maxtime` denotes that the corresponding location must be left before `x` becomes greater than maxtime, and the guard `x >= mintime` denotes that the corresponding edge can only be taken if `x >= mintime`. Both clock constraints together restrict the automaton to leave the location exactly at `x = mintime`. The symbols Ⓤ and Ⓒ depict urgent and committed locations as explained above.

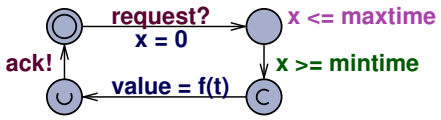


Figure 2.2: Example Timed Automaton

To model dynamic behavior, the UPPAAL variant UPPAAL SMC (Statistical Model Checker) [DLL+15] can be used. It allows for the integration of dynamic behavior in terms of ordinary differential equations (ODEs). UPPAAL SMC is designed for real-time systems with stochastic semantics and provides a stochastic simulator with trace recording and a stochastic model checker. The stochastic simulator provides the possibility to directly specify objects of interest for which value changes should be recorded in the simulation trace using the command `simulate N [≤ x] {observed variables/parameters}` that simulates the model N times for x time units and records the values of the observed variables and parameters.

Semantics

The semantics of a timed automaton is defined as a transition system on semantic states that consist of a location l and a clock valuation u, which maps all clocks $c \in C$ to non-negative real values. The initial state $s_0 = (l_0, u_0)$ consists of the initial location l_0 and a clock valuation u_0 that maps all clocks to zero. Transitions between states

describe semantic steps that can either be a time step (1) or a discrete transition (2) in the underlying timed automaton:

1. $(l, u) \xrightarrow{d} (l, u + d)$ *iff* $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(l)$
2. $(l, u) \xrightarrow{a} (l', u')$ *iff* $\exists l \xrightarrow{g,a,r} l'$ such that $u \in g \wedge u' = [r \mapsto 0]u \wedge u' \in I(l')$

where (l, u) is a semantic state, $u \in g$ denotes that all clock values satisfy the guard g, a is an action (i.e. a synchronization or a variable assignment) and $u' = [r \mapsto 0]u$ denotes the reset of all clocks defined in $r \subseteq C$ to zero ([BY04]).

The semantic state of a network of timed automata is a composition of the semantic states of each automaton. A semantic step of the network can be either a time step, a step of a single automaton or a synchronization between two automata (on the sending and receiving events c! and c?) or between one sender and an arbitrary number of receivers in case of a broadcast synchronization as additionally supported by UPPAAL.

The modeling of concurrency and time leads to an exponential increase in the number of states. To enable model checking of timed automata, the semantic state space has to be final and to improve performance the state space should be as small as possible. To achieve this, a symbolic semantic that defines an equivalence over semantic states can be used to reduce the state space.

In [AD94], *region equivalence* was introduced. The main idea of this approach is (a) to exploit the fact that only clock constraints that compare clock values with integer-values are allowed in TA, and (b) to define a ceiling $k(x)$ for each clock $x \in C$ based on the greatest integer value that constraints this clock. Thus, two clock assignments u and v can be considered equivalent if (i) their integer part is equal or both are greater than the clock ceiling, and (ii) both have a fractional part of zero or both have a fractional part greater than zero and (iii) the relation to all other clocks is preserved under both assignments, i.e. $\forall x, y$: if $u(x) \leq k(x) \wedge u(y) \leq k(y)$ then $\{u(x)\} \leq \{u(y)\}$ *iff* $\{v(x)\} \leq \{v(y)\}$ where $\{d\}$ denotes the fractional part of a real number d.

Region equivalence can be used for a finite-state partitioning of the infinite state space of timed automata, because for a fixed amount of clocks which all have a maximal constant, the number of regions is finite. Region equivalence implies bisimilarity w. r. t. the untimed bisimulation for any location or location vector of a timed automaton or a network of timed automata. The resulting finite-state model is called region automaton or region graph. However, the number of states is exponential to the number of clocks in the automaton. To further reduce the semantic state space, *zone equivalence* as first described in [Dil89] can be used as a more efficient representation of the state space. The main idea is to abstract regions into a zone, if they can perform the same discrete transitions. Formally, a zone can be described as a convex conjunction of atomic clock constraints that describe bounds on individual clocks and the differences between pairs of clocks. As zones provide a coarser abstraction than regions, they reduce the semantic state space. Furthermore, zones can be efficiently stored in memory as Difference Bound

Matrices. For more details on the semantics and model checking algorithms for TA, we refer to [BY04].

Model Checking and Temporal Logic

UPPAAL models can be verified using the build-in model checker. Model checking is a formal verification technique to automatically verify that a model M satisfies a logical property Φ. This is usually denoted with $M \models \Phi$. The model M is usually given as a labeled transition system (LTS) and converted to a Kripke structure, which represents the semantic state space. A Kripke structure is a graph with vertices representing all reachable states of the system and edges representing transitions between these states. Furthermore, it has a labeling function that maps each state to the set of atomic propositions that hold in it. Each execution path in the Kripke structure is a possible execution trace of the system. The general model checking approach is to check whether the Kripke structure fulfills Φ by checking all states. Thus, the whole state space has to be explored. However, this is often not directly feasible as concurrency and time lead to an exponential increase in the number of states. During the last decades several techniques have been proposed to face this state space explosion problem. For timed automata, which have an infinite state space due to their real-valued clocks, a symbolic semantic has been defined to reduce the semantic state space as explained above. If the property is not satisfied in the model, a counterexample, i.e. a trace from the initial state of the model to the location violating the property, is generated.

Temporal Logic CTL* Properties of interest for labeled transition systems (LTS) describe properties of states and properties over several states. To specify such properties, temporal logics can be used. To this end, the LTS can be unfolded into a computation tree with the initial location as root node. Properties over a computation tree can be specified using the the computation tree logic CTL*. A CTL* formula consists of a *state property*, i.e. a property that holds in a state, *temporal operators* that describe a temporal order of properties, and *path quantifiers* that specify whether the property should hold on all or on some paths. State properties are defined using propositional logic (atomic propositions and logical connectives). In CTL*, the following temporal operators exist:

- **X** Φ (ne**X**t): property Φ holds in the next state.
- **F** Φ (**F**inally): property Φ holds in some future states including the current state.
- **G** Φ (**G**lobally): property Φ holds in all future states including the current state.
- Φ **U** Ψ (**U**ntil): property Φ holds in all future states including the current state, until Ψ holds eventually.

There exits two different path quantifiers:

- **A** (**A**lways): The property holds for all paths.
- **E** (**E**xist): The property holds for at least one path.

The temporal operators can be freely combined in CTL*. CTL* has two important subsets: linear time logic (LTL) and computation tree logic (CTL). LTL can be used to describe properties on a single path or a set of paths. Time is considered to be linear and, thus, path quantifiers are not used in LTL. In CTL, time is not considered to be linear and properties describe the behavior on the computation tree. However, each path quantifier must be immediately followed by a temporal operator.
To specify time constraints within a CTL formula, the extension **timed CTL** (TCTL) can be used. In TCTL, CTL is extended by the possibility to introduce formula clocks and constraints over formula clocks and over automata clocks. Formula clocks can, e.g., be used to specify the timing in an *Until*-Formula or in a formula with *Finally*. For example $\mathbf{A}(\Phi\ \mathbf{U}_{\leq 3}\Psi)$ means that Φ holds until within 3 time units Ψ becomes true. The same formula can be expressed using the *freeze* operator for introducing a formula clock (c in Φ) of TCTL: z in $\mathbf{A}((\Phi \wedge z \leq 3)\ \mathbf{U}\ \Psi)$. To specify that a property becomes valid within a time bound, $\mathbf{AF}_{\leq t}\Phi$ can be used. With $\mathbf{AF}_{=t}\Phi$, we can specify that the property Φ becomes valid exactly after t time units. In TCTL the temporal operator **X** (neXt) does not exist.

TCTL Subset of UPPAAL In UPPAAL, properties that should be checked by the model checker are called Queries. Queries can be expressed in a subset of timed CTL, where the temporal operators are restricted to **F** and **G** and quantifiers may not be nested. The only exception is the liveness operator Φ - -> Ψ (Leads-To), which is equivalent to the formula **AG** ($\Phi \implies$ **AF** Ψ) and means *whenever* Φ *is satisfied,* Ψ *is satisfied eventually*. The temporal operators are denoted in box notation in UPPAAL with **F** = <> and **G** = []. In summary, the following five general properties exist in the UPPAAL TCTL subset:

- **AG** Φ (written as **A**[])
- **AF** Φ (written as **A**<>)
- **EG** Φ (written as **E**[])
- **EF** Φ (written as **E**<>)
- **AG** ($\Phi \implies$ **AF** Ψ) (written as Φ - -> Ψ)

State formulas are given by predicates constraining clocks and integer variables, or stating that an automaton is in a specific location. Global variables and clocks are identified by their name and local variables, clocks and locations have to be specified by

a prefix that identifies the template instance. In contrast to TCTL, UPPAAL does not allow to introduce formula clocks. However, this can often be modeled by introducing additional automata clocks and specifying constraints on those. To model the freeze operator in the Leads-To operator Φ - -> Ψ , the introduced clock has to be reset in an automaton as soon as Φ becomes true. This is only possible, if Φ becomes true after taking a transition, e.g. due to a synchronization or a clock invariant. Otherwise, we have to introduce an additional observer automaton that periodically (with some period d) checks whether Φ has become true, and resets a clock if this is the case. Thus, in contrast to TCTL, it is not generally possible to check $AG\ (\Phi \rightarrow AF_{\leq t}\ \Psi)$, but it is possible to check the weaker untimed property $AG\ (\Phi \rightarrow AF\ \Psi)$ or the stronger property $AG\ (\Phi \rightarrow AF_{\leq t-d}\ \Psi)$. Note, that it has to be $\leq t - d$, because we might observe Φ up to d time units after the property had become true. To check whether a deadlock exists in the given system, the UPPAAL query language provides the formula `deadlock` which becomes true, if the system contains a deadlock. If a property does not hold, the UPPAAL model checker provides a counter example trace. Traces consist of the visited states (consisting of automata locations and variable and clock assignments) and the transitions between those states.

Above, we have introduce UPPAAL timed automata, which we use as formal modeling language to support our simulation-based learning and our formal verification of newly learned rules. We have also given a short introduction on model checking and a detailed introduction on temporal logics, which provides the basis for our formal verification. In the following, we introduce the system level design language SystemC, which we use as modeling language for simulation-based learning, as well as for the implementation of our prototype of the framework.

2.4 Modeling with SystemC

SystemC is a system level design language and a framework for HW/SW co-simulation, which was was developed as an open industrial standard by the Open SystemC Initiative (OSCI - now Accellera), and has been approved by the IEEE Standards Association as IEEE 1666-2011 [IEE11]. Corporate Members of the OSCI were amongst others: ARM, Cadence, CoWare, Intel, Mentor Graphics, NXP, STMicroelectronics and Synopsys. SystemC is widely used in industry in many different fields of application. It is available as open-source code and is realized as C++ library, thus easy to use for C++ programmers. It provides language elements for the description of hardware and software on different levels of abstraction. It supports an efficient design space exploration and performance evaluation throughout the whole design process even for large and complex HW/SW systems. Thus, it is widely used for HW/SW Co-Design.

wait()	wait for an event from the static sensitivity list or that has been defined with next_trigger before
wait(SC_ZERO_TIME)	wait for one delta cycle
wait(t)	wait for t time units
wait(e)	wait for event e
wait(t, e)	wait for event e or timeout after t time units
wait(e1 & e2)	wait for both events
wait (e1 | e2)	wait for one these events

Table 2.1: The `wait()` method

Language Elements

In SystemC computation and communication are strictly separated. SystemC designs consist of communicating processes that are capsuled in modules. Communication is capsuled in channels and ports connect modules and channels.

Modules are a structural element that can be both hardware and software components. They contain variables and C++ methods, but also processes to model the behavior of the component, ports as entry points for the communication with other modules, and other modules to build hierarchical components. Modules can be parameterized to model different instances.

Processes encapsulate C++ methods. They have a (possibly empty) static sensitivity list of events. If one of these events is notified, the process becomes runnable and will be executed. During runtime, dynamic sensitivities can be declared to overwrite the static sensitivity list temporarily. This can be used, for example, to declare that a process should wait for $300ms$ before continuing. There exist two types of processes: SC_METHOD and SC_THREAD. The main difference between both types is that an SC_METHOD process runs from the start of the encapsulated method until its end and cannot be suspended during execution. SC_THREAD processes can be suspended and reactivated afterwards. Thus, their encapsulated methods usually contain a while-loop.

Events can be used to model the timing behavior of and synchronization between processes. They influence the execution of processes that are statically or dynamically sensitive to them. Static sensitivity is declared during process declaration with the keyword `sensitive`. Dynamic sensitivity is declared at runtime by using the `wait()` and `next_trigger()` methods. Both methods can be called with several arguments as shown in Table 2.1. Note, that calling `wait()` will suspend the process until the event is notified and can, thus, only be used in SC_THREADS. There exist three types of events: Timed waiting will create timed events, channels have default events for, e.g., notifying a process that a value has been written, and using the keyword `sc_event` event objects can be declared to synchronize processes within a module.

Communication is capsuled in *channels* which implement an interface. There exist basic channel types like `sc_signal` (modeling a hardware signal) or `sc_fifo` (modeling a buffered communication), and users can define own channels that are modeled as modules. Channels are connected to modules via *ports*, which provide dedicated entry points for the communication and are bound to an interface, which has to be implemented by the channel. This structure allows for an easy exchange of components (i.e. modules or channels) without requiring changes in the connected components as long as the channels all implement the same interface.

SystemC also provides additional *data types* for the modeling of hardware components (e.g., 4-valued logic, bit-vectors and fixed-point numbers) and timing behavior (e.g., clocks and time values).

Simulation Semantics

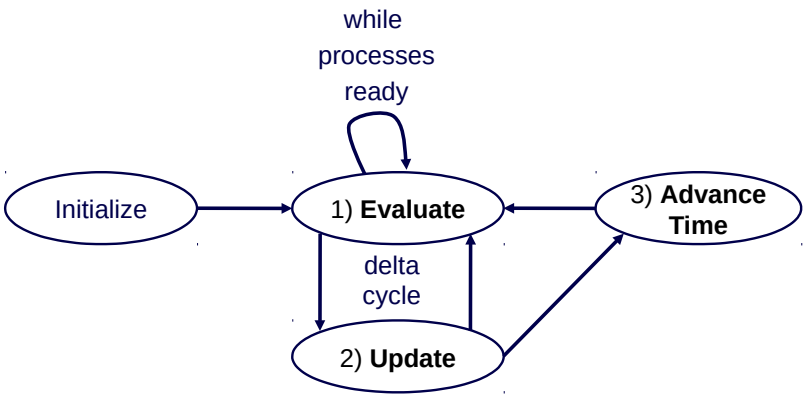


Figure 2.3: SystemC Simulation Semantics

To explore and evaluate different design decisions, SystemC comes with an event-driven simulation kernel that enables simulation of concurrent processes in a simulated real-time environment. The event-triggered execution of processes and the simulation time are controlled by the cooperative SystemC scheduler. It implements a synchronous semantics based on delta-cycles, which impose a partial order on parallel processes. Figure 2.3 shows the behavior of the SystemC scheduler. After the initialization of processes and channels, the scheduler performs delta-cycles as long as processes are runnable. A delta-cycle consists of the two phases *evaluate* and *update*. Within the evaluate phase, all runnable processes are executed. Write requests and event notifications are collected. Immediate event notifications are executed in the same evaluation phase and processes that are sensitive to those events become runnable. The execution order of processes is not specified in the SystemC standard. After all runnable processes have been executed, the update phase starts and collected changes are executed on the channels. All processes that are sensitive to corresponding channel events and processes with zero-delay notification (`wait(0)`) become runnable in the next evaluate phase. A delta-cycle does

not consume time and the number of delta-cycles that can be executed at each point of simulation time is not restricted. If no processes are runnable any more, simulation time advances until at least one process is runnable or the simulation is stopped. For more details on the SystemC semantics, we refer to [IEE11, GLMS02].

The simulator provides the possibility to record value changes of specified variables and signals in a Value Change Dump (vcd) trace file.

SystemC to Timed Automata Transformation

In our thesis, we use the SystemC to Timed Automata Transformation Engine (STATE) [HFG08, PHKG13, HPG15] for our formal verification. STATE is a java-based framework that automatically transforms a given SystemC design into a semantically equivalent network of UPPAAL timed automata. The resulting model has a similar structure as the original SystemC design and includes a formal model of the SystemC Scheduler to model the delta-cycle based semantics. STATE transforms processes, methods, and communication channels to single UPPAAL templates. These templates interact with the SystemC scheduler model based on the underlying event model of SystemC. STATE also includes several optimizations on the resulting model to improve the readability and to reduce the semantic state space for model checking. For a detailed explanation of the framework and the transformation rules, we refer to [HPG15].

The STATE framework imposes the following restrictions on transformable SystemC designs: variables have unique identifiers (no shadowing), only integer and boolean as base types, only arrays and structs as complex data types, no function pointers, no type casts, no unions, no void pointer, no recursion, statically determinable maximum of processes, method calls, and memory consumption, no pointer arithmetic, no direct memory access, noreferences to fields of structs, and no dynamic proccess creation.

In the following, we introduce Genetic Algorithms, which we use as learning algorithm for self-adaptation rules.

2.5 Learning with Genetic Algorithms

The main goal of self-adaptation is to optimize the performance (w.r.t. the system goals) of the underlying managed system in the current operation context. Thus, the adaptation has to choose the best control parameters for the current situation. Finding the best parameters for a given situation is a multi-dimensional optimization problem. An efficient approach to solve such problems are stochastic search algorithms, like genetic algorithms (GAs) [H$^+$92]. GAs are a subclass of evolutionary algorithms. They are easy to implement and achieve good results for a broad range of problems. The main idea behind genetic algorithms is to mimic the biologic evolution process by implementing suitable selection, combination (i.e. crossover) and mutation strategies on a set of valid solutions. By following the principle "survival of the fittest", the population of solutions

Listing 2.1: General Genetic Algorithm

```
1  P ← initialisePopulation();
2  evaluatePopulation(P);
3  while(not endCond())
4    C ← ∅
5    while(#C ≤ childsPerRun)
6      p1,p2 ← selectParents(P);
7      c1,c2 ← crossover(p1,p2);
8      c1 ← mutate(c1);
9      c2 ← mutate(c2);
10     C ← C ∪ {c1,c2}
11   end_while
12   evaluatePopulation(C);
13   P ← newGeneration(P,C);
14   best ← getFittest(P);
15 end_while
```

will evolve towards an optimal solution. The fitness of a solution describes how well the solution achieves the goals.

The general algorithm is depicted in Listing 2.1. It starts with a parent population (P) of solutions encoded as vector or string of parameter values. This representation is called the *genotype* of the solution. Usually the parent population is initialized with random solutions (cf line 1). Each solution is evaluated with a fitness function that quantifies how good the solution fulfills the criteria (line 2). This function usually operates on the *phenotype* of the solution, i.e. the original representation of possible solutions. For each iteration, the GA generates the specified amount of children (C). To this end, it selects two parent vectors from the current generation (line 6), combines their parameter values using crossover (line 7) and mutates some control parameters of the resulting child vectors (line 8). This process is repeated until the specified amount of children is reached. Afterwards, the fitness of all generated children is computed (line 12). The children compete with the individuals from the old population for a place in the new population for the next iteration (usually the size of the population is constant). This process is repeated until a termination criterion is reached, i.e. a sufficient solution is found or a computational limit is reached (e.g., the number of iterations).

In the following, we describe the main operators of GAs in detail and discuss different variants.

Selection Selection is used to increase the quality of the solutions. We differentiate between two types of selection: parent selection and survivor selection. Parent selection selects two solutions as parents of children for the next generation. Survivor selection determines which individuals will be part of the next generation. For both selection tasks, different solutions are possible. The most common are choosing the most fittest solutions or using probabilistic choice, which gives a higher chance to solutions with a high fitness and a smaller chance to those that have a lower fitness. Probabilistic choice can prevent

that the population gets stuck in a local optimum. The most popular selection algorithms are:

- In **Truncation Selection** the solutions are ordered by their fitness value and the best individuals are chosen. It is an easy method, but less sophisticated than many other selection methods, and, thus, not often used in practice.
- In **Fitness Proportional Selection (FPS)/ Roulette-Wheel Selection** a proportion of the overall probability is assigned to each individual based on their fitness value. This could be compared to a roulette wheel in a casino. The probability that a solution i is selected depends on its absolute fitness value f_i compared to the absolute fitness value of all other solutions. As the sum of probabilities over the whole population has to be one, the selection probability of solution i can be defined by $P_{FPS}(i) = f_i / \sum_{j=1}^{N} f_j$ where N is the number of solutions in the population. This method leads to a fast convergence if single good solutions dominate weak solutions. In later iterations, when fitness values are all close together, solutions have an almost uniform probability and thus, the mean population fitness increases very slowly.
- **Rank-Based Selection** was inspired by the drawbacks of FPS. It preserves a constant selection pressure, by sorting the solutions based on their fitness values and by selecting according to their rank. The mapping from rank to selection probability can be done in different ways, e.g. linear or exponential. An exponential mapping leads to a higher selection pressure.
- **Tournament Selection** does not require a global knowledge of the population, nor a quantifiable fitness measure. Instead, it is based on comparing any two solutions with each other to establish an ordering relation. The algorithm performs several *tournaments*, in which k individuals are ranked and the best one is chosen to be selected. The probability that a solution is selected depends on its rank in the population and on the tournament size k. A large k increases the selection pressure as only the fittest individual of each tournament is selected.

For survivor selection, there additionally exist strategies that also consider the age of solutions and prefer younger solutions over old solutions.

Crossover Crossover and mutation are variation operators that are used to create new solutions from existing ones. Crossover is used to combine two solutions by taking some part (some genes) from the first parent and some part from the second parent. This is usually done by selecting an index of the genotype to split the parent genotypes and by recombining the first part of each parent with the second part of the other parent. The index is usually chosen randomly. This method is called single-point or one-point crossover. It can be easily generalized to variants with more than one crossing point,

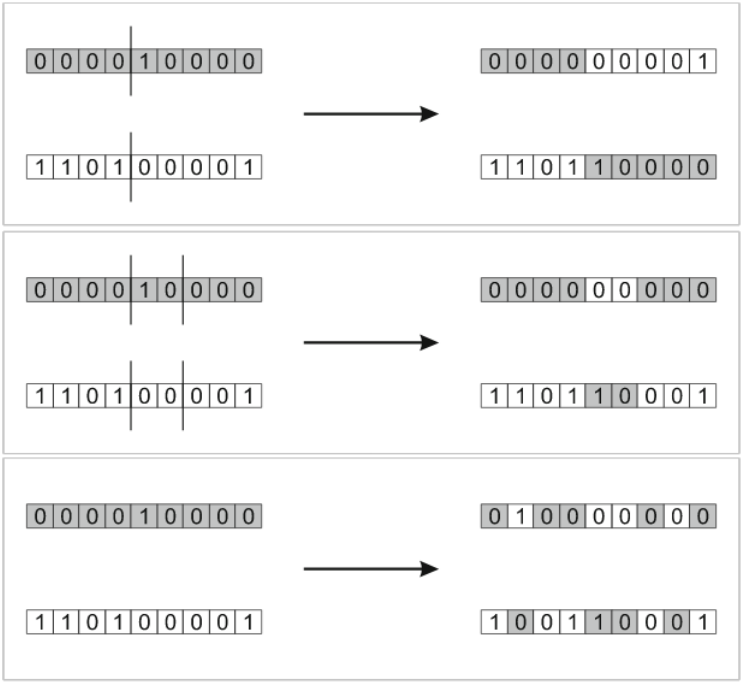


Figure 2.4: One-point crossover (top), 2-point crossover (middle) and uniform crossover (bottom) [ES+03, p. 53]

called n-point crossover. A further crossover strategy is uniform crossover, where for each parameter (gene) a random choice is performed to decide from which parent the gene should be inherited. The second child is usually created by using the inverse mapping. In Figure 2.4 these different strategies are illustrated.

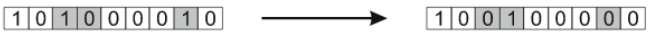


Figure 2.5: Bitwise mutation for binary encodings [ES+03, p. 52]

Mutation Mutation is a unary variation operator that mutates single parameters (genes) of a solution by applying a valid mutation operation on the selected parameters. Mutations for binary encodings are done by bit-flipping. For integer values, random resetting or adding a small (positive or negative) number can be used. Mutation of parameters is usually performed according to a probabilistic mutation rate but can also be performed by randomly deciding for each parameter whether it is mutated or not. For the probabilistic rate, each parameter is mutated with a user-defined mutation probability. While crossover can only result in solutions that have the same parameter values as their parents but in a different combination, mutation provides the population with new parameter values.

Fitness Function The fitness function forms the basis for selection. It is a function that assigns a quality measure to the genotype of a solution. Usually this is performed by transforming the genotype into the phenotype and evaluating its quality. If the problem to solve is an optimization problem, the objective function of the problem can also be used as fitness function, or a simplified version of it. Often, a weighted sum of sub-functions for each criterion is used as fitness function. If the problem is a constraint satisfaction problem or a constrained optimization problem, the satisfaction of constraints can be reflected in the fitness function by introducing penalties for constraints that are not satisfied.

For more details on genetic algorithms, we refer to [ES+03].

We use a genetic algorithm to learn adaptation rules for situations that have not been covered by the rule set before. To avoid discrepancies between the effect expectation of our adaptation rules and the actual effect, we learn context-dependent effect deviations from observed effects at runtime. In the following, we introduce classification rule learning, which we use for this task.

2.6 Classification Rule Learning

Classification rules describe the learned classification model as a set of rules of the form

IF *conditions* THEN *class*.

Ideally, this set is complete, i.e., it covers all data sets that belong to the class, and consistent, i.e., it does not cover any data that does not belong to the class. In reality, this is often not possible, and replaced by the less strict criteria coverage and accuracy, which are given as percentage of covered/ correctly classified examples.

Classification rule learning is a supervised learning task, thus all training examples are labeled with their correct class. Usually, the training data is described as a set of examples given in attribute-value representation. An example e_j then is a vector of attribute values labeled by a class label: $e_j = (v_{1,j}, ...v_{n,j}, c_j)$, where each $v_{i,j}$ is a value of attribute A_i, and $c_j \in C$ is a value of class attribute C. Attributes can either have a finite set of values (discrete or nominal) or be real numbers (numerical). As example consider the following data set that consists of three examples:

Swims	hasFourLegs	class
true	false	fish
false	false	bird
false	true	cat

From this data set, the following rule set can be learned:

- IF *swims* = *true* THEN *fish*

- IF *swims* = *false* $\wedge$ *hasFourLegs* = *false* THEN *bird*
- IF *swims* = *false* $\wedge$ *hasFourLegs* = *true* THEN *cat*

Instead of specifying *swims* = *false*, the rules can be grouped in a decision set, where rules are ordered and each subsequent rule is connected to the previous rule with *ELSE*. Thus, the first fitting rule is taken. This can also be used to resolve conflicts if several rules fit. Rules can be ordered based on the size of their conditions (rule-based), i.e. the strength of their condition, or in order of importance or frequency of the predicted class (class-based). In unordered sets, conflicts are often resolved with a voting mechanism.

Learning algorithm approaches

There exit several approaches for rule learning. The most common are the extraction of rules from a decision tree (e.g., the C4.5 algorithm) and inductive approaches using sequential covering (e.g., RIPPER). The former often suffers from subtree repetition and redundant checks. To solve this, rule pruning is used to remove conditions that are not necessary, i.e. that do not improve the accuracy of the classification rule.

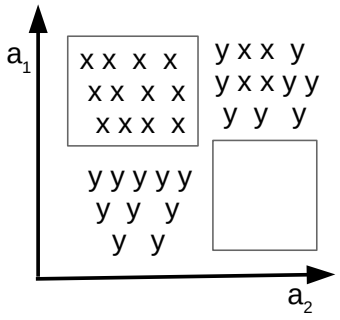


Figure 2.6: An example snapshot of the Sequential Coverage Algorithm

The inductive approaches directly extract rules from the training data by learning a set of rules for each class. They use a sequential covering algorithm that sequentially learns rules that cover many examples of the training data for one class, and ideally no examples of other classes. For each learned rule, the covered examples are removed from the training set, and the process is repeated on the remaining set until all examples are covered or the quality of learned rules is beyond a user-defined threshold. An example is given in Figure 2.6. There, the algorithm has already learned a rule for class x and deleted the covered examples. A second rule is identified (framed with a box) and some examples remain for the next iteration. With this approach, we get rules with high accuracy but not necessarily with a high coverage. Each rule is learned by using a feature construction algorithm that corresponds to a heuristic search in the space of possible rules. Most learners are top-down learners that start from the most general rule (IF true THAN class c_i) and iteratively specialize it by adding conditions as long as examples are covered

that do not belong to the predicted class. Specialization is usually guided by a heuristic quality function, such as precision or information gain, that leads to an improvement in consistency and coverage of the rule. To avoid overfitting, i.e. fitting perfectly to the training data but not generalizing well to unseen data, incremental pruning is used to generalize rules again.

Repeated Incremental Pruning to Produce Error Reduction (RIPPER) RIPPER was proposed by Cohen in 1995 [Coh95] as an inductive algorithm that does not produce rules that overfit to the data (e.g. by learning to predict noise). It combines several aspects of previous works, like incremental pruning, and adds a post-processing phase to optimize learned rules. The post-processing iteratively constructs two alternative rules for each rule and evaluates whether replacing the original rule with one of the alternatives would optimizes the overall rule set. RIPPER has a high precision and scales well on large and noisy data sets. It has been shown in several comparative studies, that RIPPER is one of the best rule learning algorithms and it is still state-of-the art in inductive rule learning [HKP12]. We have used this algorithm to identify context-dependent effect deviations of applied adaptation rules.

WEKA WEKA [WFHP16] is an open source tool for several machine learning/ data mining tasks. It provides a collection of machine learning algorithms for data mining tasks and tools for data preparation, classification, regression, clustering, association rules mining, and visualization. It provides the following classification rule learning algorithms: a decision table classifier, OneR that constructs only one rule for each target and only uses the most influencing attribute as predictor, PART, which learns rules from a partial decision tree, and JRIP, which implements RIPPER. We have used WEKA to conduct experiments for our observation-based learning.

For more details on classification rule learning, we refer to [FGL12, HKP12].

3 Related Work

In 2003, self-management was proposed as the only viable option to handle the raising complexity and dynamics of software systems [KC03]. Later this class of systems was called self-adaptive to express that these systems adapt themselves to achieve their goals. In the last two decades, researchers and engineers have developed several solutions to enable the realization of self-adaptive systems. According to Weyns [Wey17], this research can be grouped into six waves, each with a special interest in the research community. The first two waves were focused on foundations for self-adaptation: the automation of tasks, such as installing, configuring, tuning and maintaining autonomous software systems, and architecture-based solutions for self-adaptation. Afterwards, runtime models were investigated to enable reasoning about the system and its goals, and to extend the use of models from model-driven development to runtime. As goals are a key element of self-adaptive systems, goal-driven approaches became the next focal point of interest. With the increasing possibilities to build self-adaptive systems, it became conceivable to apply them in safety- or cost-critical and highly uncertain domains, such as cyber-physical systems or cloud systems. The next challenge arose: guarantees under uncertainties. This challenge is investigated in the current waves five and six, which have a different focus. In wave five, the focus lies on resolving uncertainties at runtime, whereas in wave six control-based approaches are investigated to profit from the mathematical foundation of control theory. All these research activities have contributed to the maturity of the field and further research is still necessary to reach full maturity and to support the vision of fully autonomous cyber-physical systems that we can safely trust to control several aspects of our lives. In Figure 3.1, the development of the research field of self-adaptive systems is depicted on a time line.

Our approach can be positioned in wave five (guarantees under uncertainty) and builds on ideas from waves three (runtime models) and four (goal-driven). Our combination of efficient and comprehensible adaptation-rules with a quantitative goal model that provides flexibility w.r.t. dynamic goal changes, as well as our integration of formal verification and safe runtime evolution of these rules is unique in the field of research.

In this chapter, we discuss related work to our approach. To this end, we structure our discussion according to our main contributions. First, we discuss general frameworks for self-adaptive systems with a focus on the adaptation process and knowledge models.

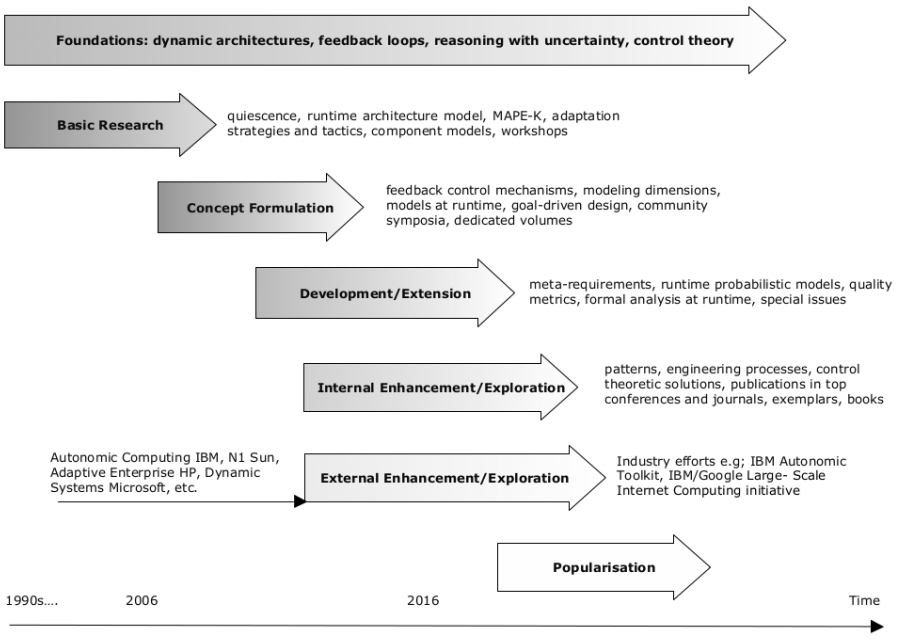


Figure 3.1: Development of the research field of self-adaptive systems. Grey shades indicate the degree of maturity in that phase. [Wey17, p. 32]

Afterwards, we focus on approaches that include online learning and optimization of adaptation operations. Then, we discuss verification approaches for self-adaptive systems. In the end of this chapter, we discuss related work to our quantitative and context-dependent goal model and the state-of-the-art in explainable CPS.

3.1 General Frameworks for Self-Adaptation

In the beginning of research on self-adaptive systems, several conceptual frameworks were developed. The most prominent framework is the MAPE-K framework, which was proposed by Kephart and Chess [KC03]. In 2007, Kramer and Magee [KM07] proposed a three-layer architecture for architecture-based self-adaptation. The bottom layer corresponds to the managed system of the MAPE-K framework. The middle-layer is responsible for adaptation of component parameters, or structural changes, such as removing or adding components or changing interconnections between components. It consists of a set of predefined plans. The top layer consists of a high-level specification of goals associated with tasks to fulfill these goals. It produces new plans if the middle layer could not handle a certain context condition with the existing plans. To this end, alternative goals are identified and suitable plans are generated. New goals can be introduced at runtime, which will also trigger plan creation.

In parallel to the research activities on self-adaptive systems, organic computing was investigated. An organic computing (OC) system is a system that adapts dynamically to changing conditions and has several self-x properties such as being self-organizing, self-configuring, self-healing, self-protecting, self-explaining, and context-aware [MSSU11].

The OC initiative has a strong focus on self-organization and emergent behavior. Emergence occurs in complex systems with interactions among large numbers of components under different operational conditions. To handle this, OC systems should also observe and adapt their own goals, plans, resources, and behaviors. A prominent architecture for OC systems is the observer/controller architecture [RMB+06]. It consists of the system under observation and control (SuOC), the observer, and the controller. This structure is closely related to the MAPE-K loop. the SuOC corresponds to the managed element, the observer is similar to Monitor and Analyze, and the controller corresponds to Plan and Execute. However, the observer/controller architecture is more detailed and also includes means for learning and optimization of the control logic. An example for the latter is [TPB+11], which we discuss in Section 3.3.

With FORMS [WMA12], a comprehensive formal reference model was developed. It consists of a small number of formally specified modeling elements and a set of relationships that guide their composition. FORMS offers a formally founded vocabulary for the most important architectural constructs of self-adaptive systems, such as the responsibilities allocated to different parts of a self-adaptive system, the processes that realise adaptation together with the models they operate on, and the coordination between feedback loops in a distributed setting. The formal representation of FORMS is specified in the formal specification language Z, thus enabling the precise description and reasoning about architectural characteristics of distributed self-adaptive software systems. FORMS can be used to define reusable architectural patterns for self-adaptive systems. It does not provide an implementation framework.

In 2013, [VTM+13] proposed DYNAMICO, a reference model that explicitly considers runtime changes in system goals and adaptation mechanisms. This framework consists of three feedback loops: One for changing goals, one for target system adaptation and one for dynamic changes of the monitoring strategies in response to changes of available sensors or changes of system goals. Changes in the goals are transferred to the adaptation feedback loop to ensure that monitoring strategies are adapted and new goals are considered during analysis and planning of adaptations.

Reference frameworks for self-adaptation that are based on runtime models are proposed in [BFT+14] and [AGJ+14]. These runtime models provide the system with self-awareness and are used for analysis and evaluation of adaptations. However, these reference models are very abstract and do not specify how the run-time models are used for adaptation and how adaptation mechanisms can be adapted. In [BFT+14], the reference model distinguishes between a ground level that basically describes a MAPE-K loop, and higher levels of abstraction that introduce various run-time models. As a result, it allows for the structured design of adaptive systems and focuses on architectural mechanisms to base adaptation on. In [AGJ+14], an abstract approach is presented where adaptation is based on system models, a feedback loop, and goal models. Their reference model possesses similarities in the structure of the system models compared to

our knowledge models and a reasoner that performs search-based learning on the system models. However, the general adaptation process within this reference model is given in a rather abstract way, especially the role of the plan models is not obvious.

None of the existing conceptual frameworks provides an integrated solution for efficient adaptation planning, runtime optimization and learning of adaptation rules and formal verification.

3.2 Planning Approaches

Planning is the most crucial part of Self-Adaptation. Two different types of planning approaches can be distinguished: online planning and offline planning. An online planner generates a plan at runtime, whereas an offline planner chooses between precomputed strategies for common cases. The latter allows for a fast response to known or predicted situations, but cannot handle unanticipated adaptation needs. In the following, we give an overview of different planning approaches.

3.2.1 Offline Planning

Many approaches focus on static adaptation using rules or configurations. Examples are [GCH+04], [SST06], [KM07] (see above), [ASSV07], [RRL+13], [FRLB15] and many more.

The most prominent framework that is based on selecting between predefined adaptation rules is Rainbow [GCH+04]. The Rainbow framework defines a reusable infrastructure for architecture-based self-adaptation. Its structure resembles the MAPE-K loop and clearly distinguishes between the reusable adaptation infrastructure and the system-specific adaptation knowledge. Adaptation knowledge is defined in terms of strategies that are defined in an *IF-THEN-ELSE* form and contain sequences of actions with their preconditions. Rainbow uses an abstract architectural model to monitor the system, evaluates the model for constraint violation, and performs adaptations if necessary. To this end, components have invariants that are associated to strategies that should be executed to restore violated invariants. Rainbow supports structural and parameter adaptation on different system levels. In [SST06], an adaptive system is abstractly modeled as a collection of communicating services that can be reconfigured based on locally available data flow information. The adaptation behavior of each service is described using configuration rules. The work in [ASSV07] provides a model-based development approach in which adaptation behavior is strictly separated from functional behavior. Adaptation is realized by switching between predefined configurations. In [RRL+13], a goal-driven approach for component-based self-adaptation is proposed. It consists of an offline phase, where a set of adaptation rules is generated, and an online phase, where rules are evaluated to choose the best adaptation strategy. The rules specify adaptation strategies for components. they consist of an event that triggers adaptation, a list of

actions that are relevant for this event and conflicting, as well as dependent adaptations. At runtime only one of the proposed actions is select. To this end, all actions are evaluated w.r.t. their influence on the key performance indicators of the system goals. This influence is specified at design time by the component designer. In [FRLB15], Event Condition Action (ECA) rules are used to specify adaptations on parameter or component level. They specify events and conditions that trigger adaptation actions. Events can be triggered by the environment or can be periodic clock events. To define more complex events and conditions, they support temporal expressions such as "during the last minute" and expressions such as "the maximum messages size received within the last 30 s". Furthermore, conditions can specify whether actions should be executed every time the condition matches, only the first time, or every time if not executed within the last t seconds. This approach facilitates the design of expressive adaptation rules that are human readable and thus support explainability. The approach also provides an approach for offline rule learning, which we discuss in Section 3.3. However, adaptation rules do not contain expectations on the effect and thus, only represent the best actions for the specified events and conditions w.r.t. the system goals at design time.

With the increasing uncertainty in CPS, such static adaptation mechanisms are not appropriate any more to deal with unanticipated changes at runtime. In our work, we combine rule-based adaptation as done in approaches with offline planning and online learning of rules. To achieve robustness w.r.t. dynamic goal changes, we provide a novel notion of timed effect adaptation rules that include effect expectations on observable environment parameters.

3.2.2 Online Planning

In online planning, different adaptation actions are evaluated for the current situation on a model of the system and the environment. Online planning can include runtime knowledge to resolve uncertainties. However, model evaluation requires sufficient time and memory, and has to be performed from scratch for every situation that requires adaptation. Thus, these approaches are typically applied to non time-critical domains or to domains with a small adaptation search space (e.g., for mode adaptation).

For online planning different techniques are used. Examples for such techniques are reinforcement learning (e.g. [KP09]), evolutionary approaches (e.g., [CGLG15]), model checking (e.g., [SHMK08]), MDP-based planning (e.g., [MCGS15]) and stochastic multi-player games (e.g., [CGSP15]).

In [KP09] and [CGLG15], adaptation plans are constructed by choosing from actions with associated learned rewards or calculated fitness values. Consequently, the impact of adaptations on the environment is only indirectly encoded and rewards have to be relearned in case of goal changes. In [SHMK08], adaptation operations are encoded as a transition system of environmental states and actions as transitions between those. In each state a set of logical propositions holds. Model checking is used to identify a

path to a state that satisfies the system goals, which are encoded in CTL. In [CGSP15] adaptation plans for architecture-based adaptation are automatically synthesized via model checking of stochastic multiplayer games (SMG). The self-adaptive system and its environment are modeled as two players of a SMG, in which the system tries to reach a goal state that maximizes a reward. By checking the maximum reward that a player can achieve independently of the strategy of the other player, an optimal strategy is synthesized. In [MCGS15], a proactive approach that explicitly considers the latency of adaptations is proposed. A probabilistic environment model is learned and combined with a nondeterministic model of the adaptive system. Bounded model checking is used to compute a strategy that maximizes the reward within a given horizon. After executing the first action of the plan, the plan is recalculated to evaluate the response of the environment. The models are encoded as markov decision procedures (MDP).

To combine the advantages of both online and offline planning, a hybrid approach was proposed in [PMCG16]. It combines deterministic planning with nondeterministic Markov Decision Process (MDP) planning. The fast deterministic planner is used to handle an immediate problem, but simultaneously the slow planner is started to provide an optimal solution. The success of hybrid planning depends on the seamless transition of execution from a deterministic plan to an MDP policy. If the predicted environment model used within the MDP are correct, the state that results from applying the first action of the deterministic planner should be included in the MDP policy. Thus, a transition is possible.

In summary, there exist various planning approaches. However, none of these approaches provide means for a fast and efficient rule-based planning that is robust to dynamic goal changes and can be adjusted at runtime to handle changes in the environmental behavior and the system topology.

3.3 Learning and Optimization of Adaptation Rules

Learning and optimization of adaptation rules has been investigated in only few approaches. Examples for investigated learning techniques are reinforcement learning (e.g. [SMQ+16]) and evolutionary algorithms (e.g., [KCW+18], [FGKV19]), and learning classifier systems (e.g., [TPB+11]) for learning of adaptation rules, and probabilistic rule learning (e.g., [SCM+13]) and model tree machine learning (e.g., [EEM13]) for online optimization of adaptation operations.

In [KCW+18], an approach for reusing knowledge of existing plans to improve the performance of replanning for unexpected changes, such as the addition or removal of adaptation tactics, changes in the goals or the environment, is presented. Replanning is based on genetic algorithms (GA). Knowledge reuse is enabled by performing the GA on candidate plans and initializing the population with individuals based on an existing plan. Due to their explicit encoding of alternatives for failures of subplans, the evaluation time

is exponential with respect to the plan size. To improve planning times, optimizations, such as trimming of long plans are proposed. In case of goal changes, this approach must reevaluate the fitness of existing plans via expensive model simulation. In contrast, our efficient calculation of the distance metric together with our separation between encoded adaptation effects and our goal model, enables a fast and efficient reevaluation of adaptation plans. Our genetic algorithm has been developed at the same time as this approach and is based on a similar idea of reusing knowledge from the last executed adaptation plan by initializing the population of our GA with individuals based on the current parameter configuration.

In [SMQ+16], adaptation operations are modeled as dynamic software product lines (DSPL). Run-time reconfigurations are driven by the DSPL feature model, which describes the possible and allowed feature combinations, and adaptation rules, which define under which circumstances a reconfiguration should take place. To deal with runtime changes in the system, its environment and goals, the authors extend the MAPE loop by learning of adaptation rules and evolution of DSPL features. Learning is based on reinforcement learning and is used to explore the configuration space of the DSPL. If none of the existing configurations is able to achieve the system goals, the need for evolving the DSPL is fed back to Evolution. However, evolution is not performed automatically, but requires human expertise. As learning is based on reinforcement learning, it suffers from the usual drawbacks, i.e. exploration in the running system which may lead to unsafe system states and the need for relearning in case of changes in the goals.

In [FGKV19], a planning approach based on optimization strategies, such as Bayesian optimization and evolutionary optimization, is proposed. In contrast to classical online planning, reuse of adaptation plans is enabled by storing optimization results. These results are associated to the situations they were learned for. Situations are dynamically identified at runtime via clustering of observed context parameters. The authors assume that for each context parameter, a discrete number of ranges for its values is provided, and that all possible system states are the Cartesian product of the ranges of context parameters. Then, clustering is used to group together states with similar impact on the output parameters. For learning of optimal configurations, a black box model is assumed. Thus, configurations are applied on the target system to evaluate their fitness. Their target system is a traffic simulation, thus this drawback is not safety critical. However, associations between learned configurations and identified situations are based on the fitness value, and thus are not valid any more after dynamic goal changes. Furthermore, similarities between situations are not exploited, thus environment changes imply that stored situations may not match anymore, and, configurations have to be relearned.

None of these approaches generalize rules to fit situations that were not used for learning. The following approach is the only approach that applies rule generalization to achieve an efficient set of rules. In [TPB+11] a multi-level observer/controller architecture for learning and self-optimizing systems is proposed. This architecture also differentiates

between a first layer, which chooses a suitable rule from a set of Learning Classifier System rules and a second layer, that is responsible for extending and optimising the set of adaptation rules using evolutionary algorithms and simulation. However, the considered adaptation rules do not include timing information and rule validation is based on simulation results only. In contrast, we introduce *timed adaptation rules* that include an effect estimation together with timing information, and we perform formal system verification before altering the knowledge base at run-time. Furthermore, we provide means for dealing with dynamic system topology changes and goal changes.

In [FRLB15], the Fossa framework for offline learning of Event Condition Action(ECA) rules is presented. These rules provide an expressive mean to specify human comprehensible adaptation rules, as discussed in Section 3.2.1. The offline learner is based on exploration strategies such as genetic algorithms to automatically create adaptation rules. These rules are evaluated offline in a test environment that uses the actual implementation code and multiple simulation settings to optimize the rule set for a multitude of environment settings and workloads simultaneously. Only rules that succeed in all test cases are kept. Fossa also applies a paired difference test for statistical significance of observed utility values. The genetic algorithm operates on the abstract syntax tree of ECA rules. This approach is similar to our approach. However, we perform online learning on the currently observed environment to deal with uncertain environment behavior that cannot be anticipated at design time. In addition to conditions and actions, we learn effect expectations on observable environment behavior to achieve robustness w.r.t. dynamic goal changes.

The following two approaches provide means for inline optimization of adaptation operations. In [SCM+13], adaptation operations are encoded as a logical program that describes available actions, their preconditions and their postconditions in terms of changes of the environment state. This model is extended to include probabilities within the expected effect. Then, probabilistic rule learning is used to update these probabilities and to change connections between actions and environment states. Learning is based on observed execution traces. In [EEM13], adaptation operations are encoded as features with a utility expectation. A learning cycle is integrated in the adaptation framework to detect inaccurate expectations on the adaptation utility. This knowledge is then used to correct the utility metrics of adaptation operations. Adaptation planning is performed by finding the optimal feature selection with integer programming. However, the utility values have to be relearned if goals change. In contrast to our work, both approaches do not consider timing of adaptations and they do not verify that applying learned rule changes does not compromise system safety.

In summary, there exist only few approaches for learning of reusable adaptation plans, and even less for online optimization of adaptation operations. However, none of the existing approaches are robust to goal changes because the fitness of learning results is evaluated once during learning and cannot be reevaluated without costly simulations.

Furthermore most approaches apply untested configurations on the target system for evaluation means, and no approach applies formal verification before introducing learning results into the system.

3.4 Verification

In the following, we give an overview of approaches that include formal models for verification of functional and safety properties in self-adaptive systems. In contrast, quantitative verification approaches, such as [CGK+11], [GCB14], and [FTG16] focus on Quality of Service requirements, such as reliability or performance. However, in this thesis we focus on safety properties.

The work in [ASSV07] provides a model-based development approach in which adaptation behavior is strictly separated from functional behavior. Adaptation is realized by switching between predefined configurations and verification properties are expressed in a temporal logic and verified using theorem provers and model checkers. In contrast, we focus on more flexible adaptation mechanisms that are adaptive themselves. Furthermore, we additionally consider timing properties of self-adaptive systems.

In [ZGC09], a modular verification approach based on Assume-Guarantee- Reasoning is presented. The adaptive system is modeled as a collection of steady-state programs and a set of adaptations as transitions between steady-states. Local properties of steady-state programs and global invariants are specified in LTL. For specifying properties that hold during the adaptation process, an extension of LTL is proposed. Adaptive systems are modeled as finite state machines that are annotated with assumptions (for adaptation) and guarantees (of stable-state programs). However, to apply this approach the set of steady-state programs has to be known in advance and the size of this set has to be manageable. Thus, it is only applicable to mode adaptation. Furthermore, timing behavior is not considered.

In [SS13] a methodology for the lightweight verification of component-based adaptive systems (called LOVER) is presented. The main idea is to enable verification for systems where some components can be replaced with a new version at runtime. At design time, these components are not known. Thus, the model is incomplete. The LOVER framework enables model checking of such incomplete models (given as labeled transition systems) and produces a set of constraints for the unspecified components, if needed for property satisfaction. At runtime, new components are verified in isolation against this set of constraints. In contrast to our approach, LOVER is only applicable for component-based self-adaptation and does not consider timing behavior.

In [NSSR13], an approach for formal modeling and verification of self-* systems based on observer/controller-architectures is presented. It is based on the Restore Invariant Approach (RIA) that separates functional behavior and adaptive behavior by defining a corridor of correct behavior (functional behavior). If this corridor is left, i.e. the invariant

is violated, the adaptation component, i.e. the observer/controller-part is assumed to set the functional components into a quiescent state (i.e., a consistent and passive state in which the system performs no actions that disrupt the reconfiguration), to adapt the reconfiguration parameters and to release the functional system from the quiescent state again, afterwards. This separation of concerns is used for compositional verification based on the assume/guarantee paradigm. For tool support, the approach has been integrated in the interactive theorem prover KIV. To support arbitrary controller implementations, a verified result checker (i.e., a result checker that has been formally verified at design time) is used at runtime. If the controller result restores the invariant, it is applied to the system. If not, it is blocked and feedback is provided to the reconfiguration algorithm. However, by relying on this approach, no liveness properties of the adaptation phase can be proven. In contrast to our approach, RIA does not support the verification of timing behavior.

In [IW14], the authors present a framework called *ActivFORMS* that uses timed automata to model and analyze self-adaptive systems. Based on a virtual machine for MAPE loops modeled in timed automata, the system implementation is directly driven by the underlying models. Furthermore, they introduce a goal management layer that adapts the MAPE models according to a goal model. The goal model itself can be updated by a system admin. The adaptation is realised as a switch between associated adaptation models for each goal. In contrast, we propose a generic framework that can be instantiated for an application by defining the abstract knowledge models and the control data, which is exchanged between the managed components and the adaptation layer. We realise the adaptation of the adaptation logic using online learning techniques on run-time models to infer new adaptation rules without manual intervention.

In [dIIW15], various formal design patterns are presented using timed automata that form components of a MAPE-K loop. Furthermore, property specification templates are given that can help to verify the correctness of adaptation behaviors. The templates have been applied to four case studies. In contrast to our work, no explicit rule-based adaptation is realized. This means that effects of adaptation and their influence on the system model cannot be analyzed. Furthermore, their approach does not cover fine-grained parameter adaptation and does not incorporate (system-level) implementations that can be analyzed with the help of formal models. Finally, although timed automata models are used, no timing properties are specified and the timing behavior of the case studies is not analyzed. Another approach that explicitly models the MAPE-K loop is presented in [ARS15]. Multi-agent abstract state machines are used to model a decentralized adaptation logic in a distributed self-adaptive system. The authors provide techniques for simulation and model checking of the adaptation logic at early design stages. In contrast to our work, they do not consider timing and they do not model the adaptation impact, i.e. the expected effect of adaptations.

[CLGS16] deals with formally modeling the impact of adaptations in architecture-based

self-adaptive systems based on discrete time Markov chains. The authors separate the adaptation impact from the environment behavior to analyze adaptations in a worst-case scenario. Uncertainty and variability are included by assigning probabilities to the outcome of adaptations and to environment actions. The proposed probabilistic models can be used together with the framework Rainbow [GCH+04] and the language Stitch [CG12]. In contrast to our work, their models do not include timing and they rely on accurate stochastic models of the adaptation impact, which assumes available field data of similar systems.

In [GKB15], we have proposed a formal architectural pattern for the construction and modular verification of distributed self-adaptive real-time systems. There, the design of adaptation components also follows the MAPE-K loop. The focus is on the separation of functional and adaptation components and how this can be exploited for analysis using the process calculus *Timed Communicating Sequential Processes* (Timed CSP) [Sch99]. In contrast, we here focus on the detailed design of the different parts of the MAPE loop and on the continuous verification at runtime.

In [SM17], a framework for model checking of adaptive systems as service is presented. As model checking is computationally expensive, the authors propose to offload this task to the cloud. The paper focuses on the dynamic allocation of required cloud resources (CPU and memory). To this end, they employ machine learning to estimate the resource usage of an actual model checking task at run time.

In summary, there are various approaches for formal verification of functional properties of self-adaptive systems. They focus on different aspects and consider models of the self-adaptive system on different levels of abstraction. However, none of the approaches provide the possibility for automatically verifying timing properties of rule-based self-adaptive systems that use parameter adaptation.

3.5 Goal Models

In this chapter, we discuss related work on goal requirement languages and goal models for autonomous systems.

Goal Requirements Languages Our goal model shares similarities with the goal requirements language (GRL) developed by [Int12]. It is based on concepts of i* introduced by [Yu97], and other models with similar concepts, e.g. TROPOS [BPG+04] (which is based on i*) or KAOS presented by [vLL00]. From these goal modeling languages, we have extracted the essence that is necessary to describe and quantitatively evaluate runtime goals. To this end, we have transferred the concept of GRL indicators to our goal model based on system and environment variables. We have generalized the provided goal decomposition types to arbitrary combinations of child distances. In both cases, we use local distance goal functions describing how far the system "is away" from achieving the respective goal. In contrast to GRL, which makes use of satisfaction levels,

we can simplify our model, because we do not require strict normalization of distances to lay in a certain interval. Moreover, we consider context-dependent precedences and context-dependent importance factors that are not part of GRL.

[PRA11] present an extension of GRL to business models. They employ logical formulas to precisely define relationships between key performance indicators. These are used in a quantitative evaluation algorithm that also includes risks. However, this approach does not address complex context-dependent relationships between parent and children goals. Moreover, modularity is limited because only ad-hoc formulas are used.

[LSYM14] present a combination of feature modeling and GRL that differentiates between optional and mandatory goals. To cope with that, the GRL evaluation strategies are adapted. Furthermore, the extended model includes special cross-tree integrity constraints in the sense of *includes* and *excludes* relations between features. The *includes* relation is similar to our precedence relation. In contrast to our work, this approach only includes quantitative evaluation in the sense of satisfaction levels of goals and thereby inherits the disadvantages of GRL as described above. Moreover, context-dependencies are not considered. With our generic distance calculation, we can capture feature modeling aspects as well, while providing more flexibility and modularity.

Context-dependent goal models are considered by [ADG10], where Tropos ([BPG$^+$04]) is extended through contextual modeling elements. Contexts are primarily used to specify how goals can be achieved in certain situations. Their semantics is similar to (context-dependent) guards in our goal model. Leaf (soft)goals can be prioritized by the user at design-time to calculate the best way to achieve the goals in the current context at runtime. However, in contrast to our work, quantitative evaluation of goal achievement and context-dependent importances/prioritization are not considered.

In the approach of [HTKS12], goal graphs are used to capture requirements, which may be subject to change during runtime. Contribution values in parent-children relationships describe priorities for the achievement of high-level goals. The basic modification operations are the deletion and addition of goals. Based on the contribution values, the impact of goal changes can be assessed w.r.t. the respective parent goals. In contrast to our work, their approach does not address gradual satisfaction of goals, context-dependencies, and precedences.

Goal Models for Self-Adaptive and Autonomous Systems [RRL$^+$13] include a goal model for self-adaptive applications. It consists of a set of goals that are based on key performance indicators (KPIs). The adaptation decision is based on information that describes the impact of available adaptations on performance indicators, and any limitations or requirements. Offline, rules are generated to specify component adaptations for a given change in the execution context. Online, these rules are evaluated whenever a change occurs to choose appropriate adaptations. We adopted their differentiation between exact and optimization goals. KPIs can be combined to composite KPIs to describe weighted combinations of subgoals. These can be used to evaluate the deviation

from a goal. In contrast to our work, the approach provides limited modularity and ad-hoc distance functions only. Priorities are only incorporated in the sense of rankings between goals.

[DM10] present a goal model for agent systems, where goals represent explicit tasks that have to be achieved. It uses precedence of goals to describe that some task needs to be executed before others. Thereby, it imposes a partial order on goals. With our goal model, we do not aim at encoding actions or tasks, but focus on an extensible structure for representing and quantifying the degree of deviation of system goals. We make use of context-dependent precedence to specify priorities on system goals.

[GBH+16] propose IRM-SA as a goal-based design approach for Cyber-Physical Systems. Goals represent invariants that have to be constantly maintained and that may be context-dependent (modeled by environment assumptions similar to our guards). Invariants are refined down to low-level obligations that can be achieved by system components. AND/OR refinements can be used to build a goal model that describes the state-space of possible system configurations to fulfill the invariants. Furthermore "requires" and "collides" dependencies can be introduced to capture dependencies and conflicts. At runtime, a SAT solver is used to select a valid configuration. To prioritize between different valid configurations, a notion of costs is introduced and combined with a total order of preferences. The costs for a system configuration depends on the selected invariants that appear in the configuration and their position in the goal graph. In contrast to our work, the goal model does not consider context dependent importance and linear precedence and the evaluation strategy only selects the best possible configuration according to the preferences but does not quantify the degree of goal satisfaction. Furthermore, runtime changes of goals are not considered.

A form of quantitative evaluation of goals at runtime is described by [CvL17]. In their model, goals are arranged in AND/OR trees. Furthermore, obstacles are included in the goal model to describe, which circumstances can lead to a violation of subgoals. Obstacles are described using LTL (linear temporal logic) formulae. At design time, rates with which obstacles can prevent goals to be satisfied are estimated. At runtime, the actual satisfaction rates of (probabilistic) obstacles are monitored over a certain period of time. They are propagated through the goal tree to obtain goal satisfaction rates. Thresholds on these satisfaction rates are used to decide whether an adaptation is necessary or not. In contrast to our work, the authors require a goal to be entirely satisfied or unsatisfied at a time. In our model, goals are assumed to be satisfiable to a certain degree at a time only, which is captured in the calculated goal's distance in a system state. Furthermore, context-dependencies are not considered, e.g. precedences and context-dependent weights.

Runtime Management of Goals in Self-Adaptive Systems [KM07] propose a three-layered reference architecture for self-adaptive systems that consists of the following layers: component control, change management, and goal management. The change manage-

ment layer is responsible for executing adaptation plans, whereas the goal management layer calculates plans to satisfy the system goals. This separation of concerns enables the introduction of new goals at runtime. In their work, they did not provide a goal model but identified the challenge of achieving a goal model that is "both comprehensible by human users and machine readable". With our goal model, we offer a solution for this challenge.

[IW14] present a framework called *ActivFORMS* that uses timed automata to model and analyze self-adaptive systems. Based on a virtual machine for MAPE loops modeled in timed automata, the system implementation is directly driven by the underlying models. Furthermore, they introduce a goal management layer that adapts the MAPE models according to a goal model. The goal model itself can be updated by a system admin. The adaptation is realized as a switch between associated adaptation models for each goal. In contrast, we propose a generic goal model that captures complex system goals together with context-dependencies and dependencies between goals. We separate goals and adaptation plans to increase flexibility and reuse of adaptation rules, which is especially important in case of uncertainties in the environment behavior, system topology, and requirements.

SimCA*([SWM17]) explicitly focuses on efficient handling of changing requirements in self-adaptive systems. There, goals describing setpoints or thresholds for parameters, or the necessity to optimize a value, are used to construct adaptation controllers with certain guarantees using control theory. Runtime changes can be handled by updating or resynthesizing those controllers. In contrast to our work, SimCA* only handles sets of requirements, which may externally be changed due to new context-situations, but that does not include context-dependencies like our guards, context-dependent importance and precedence. Furthermore, they do not consider prioritization between goals or precedences.

3.6 Explainability of intelligent CPS

Explainability has gained attention due to research projects on *Explainable AI*. Whereas these projects focus on explaining machine learning results, many CPS make context-dependent decisions that are not based on ML. To explain these decisions, some approaches focus on *explainable planning*: In [Fan18], Assumption-based Argumentation is used to model planning problems and to generate explanations for planning solutions as well as for invalid plans. [ZS19] explicitly focus on CPS. This work-in-progress aims at providing interactive explanations based on Why and Why-Not questions from end-users about specific behaviors of the system. Answers are provided in form of contrastive explanations. Explanations contain the consequences or properties of choices, and how the choices affect goals and objectives of the system. In [SSG18], verbal explanations of multi-objective probabilistic planning are automatically generated. They also use

contrastive justification as explanation for why a generated behavior is preferred to other alternatives.

In [DLFG18], the authors sketch first steps towards a conceptual framework for self-explaining CPS. They propose to add a layer for self-explanation that includes an abstract model of the system, and they propose to adjust the granularity of explanations for different user groups. They propose to construct cause-effect chains for observable actions using the abstract model. Users can access these chains to understand the cause of actions.

In [WFF19], a feedback loop approach is used to identify situations where it is valuable to ask a user for feedback about system behavior. There, the authors compare the user behavior with a goal model and ask for feedback when users achieve sub-goals or when they deviate from an expected sub-goal.

Other work has focused on rationalizing and verbalizing the behavior of autonomous agents. Rationalizations do not need to accurately reflect the true decision-making process, but give some explanations like humans would give in similar situations. In [EHCR18] an agent's actions are rationalized by using an encoder-decoder neural network to translate between state-action information and natural language. In [PSRV16] the agent's experiences on a route are verbalized by converting sensor data into natural language as answer to user queries with varying levels of abstraction, specificity and locality. Another approach to generate explanations at run-time is to use a multi-modal agent that can be queried 'on-demand' [RCGL$^+$18, CGRL$^+$18a]. There, the system behaviors are mapped into a modified version of fault trees, which the authors call *model of autonomy*, that capture the possible states of the system [CGRL$^+$18b]. The authors found that the explanations given by the agent helped improving the fidelity of the operators' mental model, increasing the operator's understanding of what the autonomous vehicles were doing and why, as well as how they work [CGRL$^+$18a].

In summary, there are some approaches towards achieving (self-) explainable CPS. However, to the best of our knowledge, none focus on explaining of self-adaptive actions and runtime learning.

3.7 Summary

There exists a broad spectrum of approaches that is related to our thesis. First of all, there is the wide area of frameworks and planning approaches for self-adaptive systems. However, none of the frameworks provides an integrated solution for efficient adaptation planning, runtime optimization and learning of adaptation rules and formal verification. The existing planning approaches do either offline planning or online planning based on model simulations. In our work, we combine rule-based adaptation as done in approaches with offline planning and online learning and optimization of those rules. To achieve robustness w.r.t. dynamic goal changes, we provide a novel notion of timed effect

adaptation rules that include effect expectations on observable environment parameters. Secondly, there are few approaches that have investigated online learning of reusable adaptation rules. None of these is robust w.r.t. dynamic goal changes because the utility of learned rules is evaluated once during learning and cannot be reevaluated without time and memory consuming simulations. The existing approaches for online optimization of adaptation operations do not consider timing of adaptations. Furthermore, none of the learning and optimization approaches apply formal verification before introducing learning results into the system. Thirdly, we have discussed existing approaches for formal verification of functional properties. There exist various approaches that focus on different aspects. However, none of the approaches enables the formal verification of timing properties of rule-based self-adaptive systems that use parameter adaptation. Fourthly, existing goal modeling languages cannot capture context-dependent weights of subgoals and none provide a modular quantitative evaluation of goal satisfaction in a given environment state. Existing goal-driven approaches for self-adaptation do not capture the gradual satisfaction of goals in different environmental contexts. Lastly, the increasing need for explanations of autonomous actions, especially for machine learning, has led to an increasing interest in explainability. There already exist some approaches towards achieving (self-) explainable CPS. However, none focus on explaining of self-adaptive actions and runtime learning. To the best of our knowledge, there is no approach available yet that allows for the design and runtime evolution of safe, intelligent and explainable self-adaptive systems.

4 Safe, Intelligent and Explainable Self-Adaptive Systems

How can we design intelligent self-adaptive systems that are flexible enough to cope with ever-changing operational contexts and how can we ensure safety and explainability of their autonomous decisions?

In the near future, intelligent cyber-physical systems (CPS), such as self-driving cars, smart homes or the internet of things, will influence our daily lives. They have to cope with uncertain and changing environments and must adhere to strict safety requirements. As an example, consider an autonomous vehicle. It is not possible to predict every situation the vehicle will face while on the roads at design-time. Thus, the control software has to be able to autonomously adapt to the current situation, while still providing required safety guarantees. The aim of this thesis is to support the design of intelligent self-adaptive systems that are flexible enough to cope with dynamically changing operational contexts, and, at the same time, provide safety assurances and explainability of their autonomous decisions. In the thesis, we focus on intelligent CPS that autonomously autonomously adapt themselves to changes in system, environment and goals. We provide a framework that enables the integrated design and formal verification of intelligent self-adaptive systems that adapt themselves to changes in system, environment and goals. To achieve our objectives, we combine a resource-efficient process for self-adaptation with dynamic evolution of the adaptation logics and continuous verification activities.

In the following, we start with an overview of our overall approach before we describe the framework architecture in detail.

4.1 Overall Approach

Our key idea for designing intelligent self-adaptive systems is to view adaptation logics as 'first-class citizens' that can evolve at runtime and that provide comprehensible access to the underlying knowledge and assumptions of adaptation and evolution decisions. In our framework, we differentiate between three layers of the system: the managed system, the adaptation layer and the evolution layer, as depicted in Figure 4.1. The managed system layer is responsible for the functional behavior of the system and interacts with the

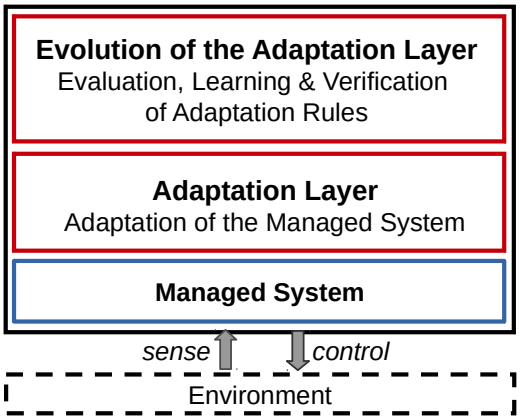


Figure 4.1: Three-layered Framework Structure

environment. It can operate independently from the other layers without self-awareness or autonomous adaptation abilities. The adaptation layer is responsible for autonomously adapting the managed system. We base adaptation decisions on a set of adaptation rules and assumptions of their effect on the environment w.r.t. the system goals. The system goals are encoded in a goal model that is shared by the adaptation and evolution layer. Managed system and adaptation layer form a self-adaptive system that is able to adapt to changes in its system, environment and goals, as long as suitable adaptation rules are available and encoded assumptions on the environment are met. The topmost layer is responsible for the runtime evolution of the adaptation rules. To this end, we continuously evaluate the correctness of the assumptions on the observable effect of adaptations and adjust incorrect assumptions based on previous observations. Furthermore, we combine simulation-based learning of new adaptation rules with heuristic rule generalization to generate new adaptation rules for situations that were not (accurately) captured by the existing rules. We perform comprehensive verification w.r.t. environment assumptions of the adapted rule set before applying it in the running system. We thereby ensure that runtime evolution does not compromise important properties, e.g. safety properties. We use executable runtime models for our simulation-based learning and comprehensive verification. We have published previous versions of this framework in [KGG15, KGG18b].

In the following, we first describe our proposed framework architecture in detail. Afterwards, we discuss general assumptions for our framework before we introduce our illustrative example. Our main contributions are explained in detail in the following chapters: We describe our adaptation layer and the components of our structured knowledge base in Chapter 5. Here, we also explain the details of our explanation base that provides the basis for explanations on the applied adaptations of the system. Our explanation base is part of our knowledge base. We present our expressive quantitative goal model and our distance evaluation algorithm in Chapter 6. In Chapter 7, we present our safe and resource-efficient evolution layer with rule-learning and verification in detail.

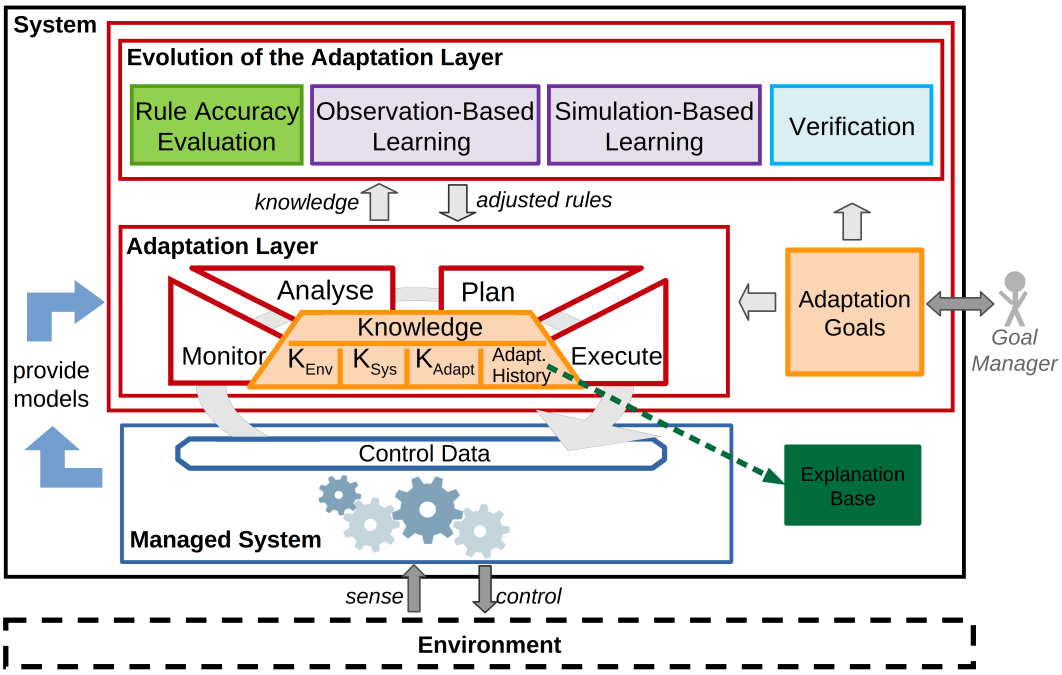


Figure 4.2: Our Framework Architecture

4.2 Framework Architecture

Our proposed framework architecture is depicted in Figure 4.2. It consists of three layers: a managed system, an adaptation layer and an evolution layer that is responsible for the runtime evolution of the adaptation logics.

Managed System The *managed system* is able to perceive and control the *environment* via sensors and actuators. It is responsible for the functional behavior of the system and may consist of several interconnected embedded systems. It can operate independently from the other layers, but without self-awareness or autonomous adaptation abilities. It is adapted by the *adaptation layer* in case of changes in system, environment and goals. Dedicated *control data* of the managed system serve as an interface between both layers. Control data are controllable system parameters including sensor data to provide information on the current environment state.

Adaptation Layer We base our adaptation layer on the MAPE-K feedback loop [KC03, IBM04], which is widely used in the design of self-adaptive systems [GVD+17]. It consists of four general phases: **M**onitoring of the managed system (by accessing or polling the control data), **A**nalysis of goal satisfaction, **P**lanning of adaptations for re-establishing violated goals or for self-optimization and **E**xecution of generated plans. All four phases share a common **K**nowledge base. In this thesis, we separate the knowledge about environment (K_{Env}), system (K_{Sys}), adaptation options (K_{Adapt}), experiences from executed adaptations (*adaptation history*), and system goals that should be maintained by

self-adaptation (*adaptation goals*) and provide models for each of them. Furthermore, we add an *explanation base* that contains structured information about adaptation decisions to enable retracing and explanation of executed adaptations. We use a comprehensible rule-based definition of adaptation logics for which we define a novel notion of *timed adaptation rules*. The key idea is to encode the expected effect of an adaptation in terms of observable changes in monitored data together with an estimation of the required time until the effect is generally observable.

To analyze goal satisfaction of the currently monitored state as well as of states that are expected to result from executing adaptation plans, we provide a hierarchical goal model together with a modular distance evaluation between a given system and environment state and the system goals. Based on this, we provide a rule- and distance-based adaptation process to enable efficient adaptation planning. By strictly separating adaptation rules and system goals, we achieve robustness of the adaptation logic w.r.t. dynamic goal changes.

Evolution Layer On top of the adaptation layer, we introduce an evolution layer. It continuously evaluates timed adaptation rules w.r.t. the accuracy of their effect expectations (*Rule Accuracy Evaluation*) and learns accurate rules for situations that were not (accurately) captured by the existing rules. Our accuracy evaluation is based on a comparison between our encoded effect expectations and the observed effects of adaptation rules. For learning, we use an observation-based correction of inaccurate effect expectations (*Observation-Based Learning*), and an evolutionary approach on model simulations to generate new rules for unmodeled situations (*Simulation-Based Learning*). Furthermore, we verify the improved adaptation logic at runtime (*Verification*). This layer can be seen as a meta-adaptation layer because it adapts the adaptation logics of the adaptation layer. To realize learning and verification, we employ executable runtime models of the system components and the environment to analyze their interaction w.r.t. the adaptation goals.

Adaptation Goals We provide a modular quantitative goal model that provides a hierarchical encoding of the *Adaptation Goals* together with their inter- and context-dependencies. We introduce a modular distance evaluation algorithm capturing the distance between a system state and the adaptation goals. The main idea of introducing such a distance is to compare the effect of different adaptation options w.r.t. multiple possibly conflicting goals. With this approach, we can formulate the problem of finding an optimal adaptation plan as an optimization problem. The objective is to maximize the fitness of the managed system, i.e. to minimize its overall distance to the goals. As goals may be contradicting, we have to solve a multi-objective optimization problem. In our rule-based setting, adaptation rules describe a final and usually small set of possible optimization steps that can be compared based on their effect expectations and the resulting distance. Thus, planning an adaptation has to operate on this small set of

solutions only. For learning of new rules, we use a genetic algorithm to deal with the optimization problem. The genetic algorithm uses our distance evaluation of the goals as fitness function.

With our modular goal structure and our modular distance evaluation, we achieve reusability of analysis and planning results for dynamic goal changes. The goal model is maintained by a goal manager who can be a person (e.g. the user in a smart home or a manager of a smart production system) or a system (e.g. an autonomous update server for legal regulations in an autonomous drone scenario).

The focus of this thesis lies on the runtime evolution of the adaptation logics and the comprehensibility of autonomous decisions. Thus, we have decided to use a straightforward planning algorithm instead of encoding probabilistic effects of adaptation rules. We include fuzziness by adding a λ_p around the effect expectations, as we assume the effect to be in the same equivalence class, i.e. deviations in this range have little impact. Effect deviations beyond this λ_p are detected by our *Rule Accuracy Evaluation* and lead to adjustments in the adaptation logic. Here, we assume that it is always possible to infer a reason for deviations from analyzing the monitored system and environment state at the time of rule execution. In future work, our framework could be combined with approaches that model uncertainty in the adaptation logic to enhance the abilities of our framework. For simplification, we assume that the system architecture consists of a central controller with a central MAPE-K loop. However, our approach principally applies to distributed MAPE-K loops as well. To this end, we plan to investigate the distribution of our knowledge base and also distributing our evolution layer in future work.

The advantages of our framework are that it enables us to integrate comprehensible timed adaptation rules, a runtime accuracy evaluation, a learning-based runtime evolution of adaptation logics, and a formal verification based on runtime models into the well-known and widely used MAPE-K framework. In combination with our explanation base, we furthermore provide the basis for automatically generating explanations on the observed adaptation behavior of the system. As a result, we enable the design of trust-worthy, safe and reusable dynamically-evolving adaptation logics.

We will now discuss our general assumptions for our framework, before we introduce our illustrative example that we use as a running example in the following chapters.

4.3 Assumptions

We make the following assumptions for our framework:

1. We assume that the managed system can be set into a safe operation mode where it does not fulfill all system goals, but safety can be guaranteed. This mode serves as a fallback option in case of failure of the adaptation layer, i.e. if no suitable adaptations can be applied.

2. We assume the adaptation goals to be quantifiable, i.e. their degree of satisfaction can be expressed quantitatively.

3. Each system component provides an interface to its control parameters and its atomic adaptation operations on them.

4. Each system component provides a relation $R \subseteq (SP \cup EP) \times EP$ describing which control parameters of the system (SP) may influence which environment parameters (EP), as well as known dependencies between environment parameters.

5. Each system component provides executable runtime models, which can be simulated and used for verification at runtime. To capture runtime behavior, we assume that they can be updated with runtime data. Furthermore, we require our runtime models to have a formal semantics, which enables us to employ them for verification purposes.

6. We assume that the designer provides environment simulation models. We require those models to have a formal semantics to employ them for verification purposes.

7. For the runtime environment of our framework, we assume an external trustworthy server to perform adaptation rule learning and verification tasks.

Note that the design of the safe operation mode of Assumption 1 depends on the criticality of violated system goals. In a non-critical environment, the system can continue execution without adaptations although, optimal performance cannot be guaranteed, for example. In a safety-critical domain like industrial production, parts of the production can be stopped until manual maintenance is finished.

Assumption 2 is substantial for our approach because analysis, planning, learning, and verification are based on quantitative distance functions that capture the distance between the current system and environment parameter values and the (current) system goals. In this thesis, we have developed a goal model from which such distance functions can be generated automatically.

Assumption 3 is necessary for applying planned adaptations to the system components, i.e. executing a sequence of adaptation operations on control parameters. Assumptions 4-6 ensure that adaptation rules can be learned and verified before being deployed to the system. Assumption 7 takes into account that learning and verification are time- and resource-consuming. Thus, these tasks should be outsourced to save resources within the actual system.

The assumptions above limit the applicability of our approach to applications where the adaptation layer does not directly control a physical environment, but adapts a controller within the managed system, and a safe operation mode can be defined. We assume that adapting the controller is usually necessary for a few times only. Thus, the potentially high effort of learning and verification is acceptable compared to the benefit gained by optimizing the system. The safe operation mode has to be safe (in the sense

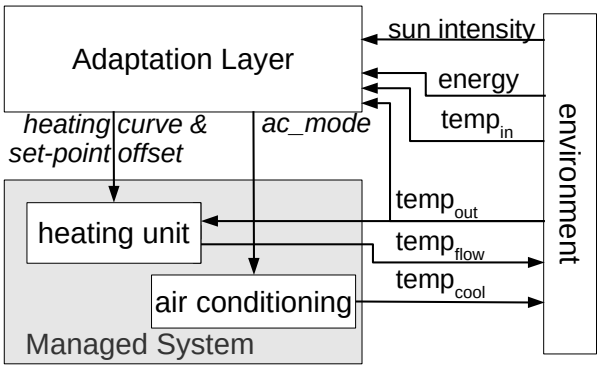


Figure 4.3: Smart Temperature Control

of satisfying the most relevant requirements) for an undefined amount of time to assure that the system operation is safe even in the case that no suitable adaptation can be learned automatically or verification fails. In highly-safety critical systems like airplanes or autonomous vehicles, this safe operation mode may consist of several fall back levels, the last option being manual operation.

4.4 Illustrating Example: Smart Temperature Control

As a running example, we introduce a smart temperature control system with a heating unit, an air conditioning and additional sensors to monitor indoor and outdoor temperature, and sun intensity (as shown in Fig. 4.3). The overall purpose of the temperature control is to keep the indoor temperature close to a desired temperature of $20°C$ at daytime (from 6 a.m. to 8 p.m.) and $16°C$ at night-time (from 9 p.m. to 5 a.m.). The heating unit uses a heating curve to determine its required flow temperature flow_temp from the current outdoor temperature. The air conditioning has three modes: off, on and power that can be used to cool down the room in summer. The power mode has more power than the normal mode on and provides a lower cooling temperature. We assume that the heating unit and the air conditioning are independent control components whose control parameters can be adjusted by an adaptation layer to cope with changes and uncertainties in the environment. This case study was created by ourselves to illustrate our concepts.

The smart temperature control system has to adhere to two main requirements: optimal temperature control and energy efficiency. Temperature control is the most important aspect. Thus, the more severely the temperature goal is violated, the less important the energy goal becomes. In the following, we describe the functionality of the managed system, i.e. heating unit and air conditioning, and sketch the responsibilities of the adaptation layer.

Managed System The managed system consists of the heating unit and the air conditioning. The sensors to monitor indoor and outdoor temperature and to measure sun intensity also belong to the managed system, but only serve as an interface to the environment, together with a sensor for the current energy consumption. The heating unit provides three control parameters: the gradient m, the offset n of the heating curve, and an offset st_offset that can be added to the flow temperature. The parameters of the heating curve are used to adjust the heating unit to the thermal properties of the building. The gradient m describes how much a change in outdoor temperature influences the flow temperature. Adjustments of the offset n lead to a uniform change in the flow temperature and can be used if it is, e.g., always slightly too warm. In contrast, adjustments of the gradient are necessary if the failure of the heating regulation is caused by the degree of changes in the outdoor temperature (e.g., it works on days with uniform temperatures, but not with cold nights and warm days). The setpoint offset is used to reduce the heating in case of additional heating from sun rays. The heating unit reacts to changes in the outdoor temperature or to notifications from the adaptation layer, and calculates the necessary flow temperature according to its heating curve. The actual heating process is abstractly modeled by a time passage of t_{heat} seconds after which the new flow temperature is passed to the environment. The air conditioning can be adjusted by switching between the three modes off, on and power.

Environment As a simplification, we use an abstract environment model that is represented by an ideal heating curve, as well as an ideal cooling curve, with the assumption that a discrepancy of $2°C$ to the ideal flow/ cooling temperature results in a deviation of $1°C$ in the room temperature. Furthermore, environmental influences like sun intensity are considered. A high intensity will lead to higher room temperatures. Heating can be reduced in these situations. This is abstractly modeled by a sun intensity factor that captures the percentage of the necessary flow temperature that will be contributed by the sun. The remaining required flow temperature is calculated as follows: requiredFlow$-=$ (sun_intensity $\times$ requiredFlow)$/100$. In the summer, we do not explicitly consider the sun intensity, as the room temperature already increases with the increasing outdoor temperatures.

The energy consumption depends on the flow temperature of the heating or the cooling temperature of the air conditioning. As we do not aim at modeling the real energy cost, but rather a trend in the cost, we simplify the calculation of the energy consumption. To this end, we assume that the energy consumption of our smart heating increases by 1kWh per additional degree of its flow temperature. Thus, the energy consumption of the heating unit is abstractly modeled by energy(flow_temp) $=$ flow_temp $-$ MIN_FLOW, where MIN_FLOW is the minimal flow temperature of the heating unit. For the energy consumption of the air conditioning, we use constant values for each mode. This abstraction is used to focus on the adaptation logic and allows for modeling environmental changes by simply changing its ideal heating curve. The environment consists of three

processes that provide the day time, the outdoor temperature (following a predefined temperature sequence of a day), and the room temperature. In our environment model, we have a coarse sample time of one hour. Thus, daytime and outdoor temperature are updated after one simulated hour. The room temperature is updated t_{room} seconds after a change in flow temperature or daytime occurred, with $t_{\text{heat}} < t_{\text{room}} \ll 1h$ to give the system sufficient time to react (as defined above, heating takes t_{heat} time units). Note that this is a simplification which could also be modeled with a (discretized) continuous change in the flow temperature during the heating process. In the remainder, we assume $t_{\text{heat}} = 30s$ and $t_{\text{room}} = 60s$.

Adaptation and Evolution Layer The adaptation layer is responsible for optimizing the managed system with respect to the temperature and energy goals, which may change at runtime. Uncertainties like the actual energy costs or the influence of incoming sun rays on the indoor temperature can be reduced at runtime by learning more accurate effect predictions based on runtime data.

We use the step-wise development of a safe and explainable adaptation logics for our temperature control system and the runtime evolution of the developed logics to illustrate our framework. We first consider a system without air conditioning, construct the adaptation logics and illustrate our observation-based learning. Afterwards, in Chapter 7.3, we add the air conditioning component and show how simulation-based learning can be used to infer suitable adaptation rules to enable effective temperature control in summer.

Summary We combine resource-efficient rule-based adaptation and on-line optimization and learning of adaptation rules to provide the necessary flexibility for uncertain environments, while reducing costly learning to a minimum that can also be moved to external servers. We can cope with dynamic goal changes due to the modular structure and distance evaluation of our goal model, as well as our separation of effect expectations of adaptation rules and the distance evaluation of the expected resulting state w.r.t. the goal model. These solutions enable our systems to maintain their system goals in ever-changing operational contexts, thus being robust w.r.t our definition. Our explicit timing information for adaptation effects further allows for latency-aware proactive adaptation and for the application in real-time systems. By embedding formal verification into the design process and into the evolution process for the adaptation logic, we ensure that our intelligent self-adaptive systems behave as intended. We provide an explanation basis for autonomous adaptation and evolution decisions to obtain trust in their correctness.

In the next chapter, we describe our resource-efficient adaptation layer and our knowledge models in more detail. Afterwards, we present our expressive quantitative goal model in Chapter 6 and our evolution layer in Chapter 7 in detail.

5 Resource-Efficient Self-Adaptation

How can we design resource-efficient and modular adaptation logics?

We base our self-adaptation on the widely used feedback-loop architecture MAPE-K [KC03, IBM04]. This general architecture abstractly describes the tasks of and the interaction between a monitoring, analysis, planning, and execution phase and includes a shared knowledge base for information transfer. To achieve a resource-efficient and explainable self-adaptation that is independent from the system goals, we provide a rule-based encoding of the adaptation logic, where planning corresponds to choosing a suitable rule from a bounded set of rules based on context-specific expectations on the effect of those rules. We base the evaluation of whether goals are sufficiently satisfied in a given system state (e.g., currently observed, predicted to be observable without adaptation, or reachable by adaptation) on a distance metric between the system state and an optimal state defined by the system goals. We define the system goals in a hierarchical goal model and provide an algorithm to calculate the distance. With this separation, we achieve the desired independence between adaptation logic and system goals. Our distance evaluation provides a flexible coupling between them. As this evaluation is fast, we perform it in every analysis and planning phase to cope with dynamic changes in the environment and system goals. We have chosen a quantitative evaluation of the degree of goal satisfaction to enable the fine-granular evaluation of adaptation effects on possibly conflicting goals to find an optimal trade-off between those goals. For continuous runtime evaluation of the rule-accuracy and for explainability, we store relevant data about selected adaptations in our adaptation history and explanation base.

In the following, we first describe our knowledge models in detail. Afterwards, we describe the different phases within our adaptation layer and their interaction with our knowledge models. In the end of the chapter, we explain how our adaptation history can be used as basis for explaining the decisions of our adaptation layer. Our evolution layer is described in detail in Chapter 7.

5.1 Knowledge Models

The knowledge models of the adaptation layer play a central role in the whole adaptation process. They capture the knowledge about the system and environment that was gained by monitoring, they describe the system goals and the available adaptation rules. Furthermore, they capture relevant data of executed adaptations for explainability. The way this knowledge is encoded influences the performance of the overall adaptation and evolution process in our framework. Thus, we have designed the knowledge models with care in order to meet our criteria for the thesis, namely continuous learning, independence between adaptation logic and system goals, continuous analysis of safety properties, explainability, and resource efficiency.

In the following, we present our knowledge models in detail and illustrate them using our illustrating example.

5.1.1 Environment Model K_{Env} and System Model K_{Sys}

The environment model and the system model capture the collected relevant information about the environment and the system. We keep this data as simple as possible, because it is continuously collected and updated in the monitoring phase within the adaptation layer. Thus, we choose a set of sensor values and system parameters as abstract representation. To represent system and environment behavior over time, we also include a history of the collected data of previous monitoring cycles. This representation is efficient and sufficient as we do not need executable models for our rule-based adaptation logic. Moreover, we also capture information about the current system topology and its changes in the abstract system model K_{Sys}. To this end, we use a set of currently available system components that is updated during monitoring.

Example In our illustrating case study, the MAPE-K adaptation layer continuously monitors indoor and outdoor temperature every *cycle* time units. To this end, temp_in and temp_out are introduced as corresponding knowledge variables together with a variable for the current daytime (time). Furthermore, the sun intensity (sun_intensity) and the energy consumption (energy) are measured. Additionally, the history knowledge variables $\text{temp\_in}_{\text{old}}$ and $\text{temp\_out}_{\text{old}}$, which represent the respective values in the last monitoring cycle, are updated accordingly. These parameters describe aspects of the environment and belong to K_{Env}. The heating parameters m (gradient), n (offset), st_offset (flow temperature offset) and the air conditioning mode airCon_mode are system parameters stored in K_{Sys}.

5.1.2 Timed Adaptation Logic K_{Adapt}

An essential model of our structured knowledge base is the adaptation logic, which is represented by a set of *timed adaptation rules*. We define a timed adaptation rule r_i to

consist of four parts: an application condition, control data manipulation commands, an expected effect, and a time after which the effect is assumed to be observable:

$$r_i : g_i \ \textbf{\&}\ c_1; c_2; \ldots; c_n \longrightarrow \textit{effect}\ \textbf{after}\ \textit{time}_{\pm[timeTolerance]}$$

The guard g_i is a condition on the system and environment parameters to describe when the adaptation rule is applicable. For proactive adaptation, the guard can also describe conditions on predicted future states or generally on computation results of the analysis phase. The commands $c_1, c_2, ..., c_n$ describe how the control data of the managed system is manipulated on applying the rule. The *effect* predicate describes the expected effect, i.e. how the environment (behavior) is expected to be influenced after applying the adaptation rule. Finally, *time* describes the smallest amount of time after which the *effect* is assumed to be generally observable. Thus, *effect* describes a relation between system and environment state before the adaptation and a state after adaptation within *time* time units. The effect prediction is used for our distance-based planning, and in the evaluation component to check whether adaptation rules are accurate. An additional acceptable delay *timeTolerance* specifies the amount of time that the effect is allowed to occur later than *time*. This delay is considered during our rule accuracy evaluation. As the focus of this thesis lies on the runtime evolution of the adaptation logic and the explainability of autonomous decisions, we have decided to focus on simple effect predicates and against encoding probabilistic effects of adaptation rules. However, our approach principally works for more complex effect predicates as well. Probabilistic adaptation effects would require some adjustments of the planning algorithm, the rule accuracy evaluation, the executable runtime models (e.g., stochastic timed automata instead of timed automata), the rule learning algorithms and the usage of a different model checker for verification (e.g., UPPAAL SMC). In future work, we plan to investigate the integration of probabilistic effect predicates to enable the explicit encoding of uncertainties in the effect predicates.

The advantage of our timed adaptation rules is that they enable us to describe an expressive, yet comprehensible, adaptation logic. They make adaptation decisions explicit and comprehensible due to their explicit application condition and timed expected effect. Their general condition-action-effect structure is a comprehensible way to specify action options and their effect for outcome evaluation. They provide a basis for increasing trust in autonomous decision-making, because of their predictability, their transparency w.r.t. their actual decisions, and their (formal) verifiability. Furthermore, adaptation rules provide a modular encoding of the adaptation options which allows for easy adjustment and exchange of single rules at runtime.

Example In our case study, an adaptation is necessary if there is a deviation from the desired room temperature refTemp (temp_in $\pm\ tolerance \neq$ refTemp). Then, the adaptation unit can adjust the heating unit by adjusting the gradient m and the offset n of the heating curve based on the knowledge variables, or adjust the setpoint offset

st_offset to compensate for incoming sun energy sun_intensity. In summer, the air conditioning can be adapted by switching between its modes. We have defined four adaptation rules for increasing/ decreasing the gradient and offset of the heating curve, and one rule for adjusting the setpoint offset of the heating. For the air conditioning we have defined two rules to switch between modes. As examples, we here present the rules for increasing the offset of the heating curve and for adjusting the setpoint offset.

increase n:

$$\begin{aligned}&\text{temp\_out} < \text{refTemp} \ \wedge \ \text{temp\_in} + tolerance < \text{refTemp}\\ &\quad \wedge \ \text{temp\_in} = \text{temp\_in}_{\text{old}}\\ &\ \textbf{\&}\ n := n + \text{stepsize}_{\text{n}}\\ &\longrightarrow (\text{temp\_in}_{\text{env}} \geq \text{temp\_in} + \text{stepsize}_{\text{n}}/2\\ &\quad \wedge \ \text{energy}_{\text{env}} = \text{energy} + \text{stepsize}_{\text{n}})\\ &\quad \textbf{after}\ k\ \text{seconds}_{\pm[timeTolerance]}\end{aligned}$$

This rule can be used if the monitored outdoor temperature is below the reference temperature (indicating a general need for using the heating unit) and the indoor temperature (temp_in) is (sufficiently) below the intended indoor temperature (refTemp) but was stable in the last monitoring cycle ($\text{temp\_in} = \text{temp\_in}_{\text{old}}$). The effect of this rule encodes the assumption that a difference of $2°C$ in the flow temperature corresponds to a difference of $1°C$ in the room temperature ($\text{stepsize}_{\text{n}}/2$), and the expected relationship between flow temperature and energy (linear for simplification). Note variables like temp_in and $\text{temp\_in}_{\text{old}}$ refer to monitored values in the knowledge base. In contrast, $\text{temp\_in}_{\text{env}}$ refers to a future indoor temperature, independent from a concrete monitoring cycle. Thus, the expected effect time describes the earliest point in time when the effect in the environment is expected to be generally observable. As the system can only evaluate this effect based on monitoring data, *timeTolerance* should be at least the monitoring sample time that is used for effect monitoring.

If the indoor temperature is above the intended temperature, the offset is decreased in a similar rule.

adjust st_offset:

$$\begin{aligned}&\text{temp\_out} < \text{refTemp}\\ &\ \textbf{\&}\ \text{st\_offset} := 0 - \text{sun\_intensity} \cdot \text{refTemp}\\ &\longrightarrow (\text{temp\_in}_{\text{env}} = \text{temp\_in} - \text{refTemp} \cdot (\text{sun\_intensity} - \text{sun\_intensity}_{\text{old}})\\ &\quad \wedge \ \text{energy}_{\text{env}} = \text{energy} - \text{energy} \cdot (\text{sun\_intensity} - \text{sun\_intensity}_{\text{old}})\\ &\quad \textbf{after}\ k\ \text{seconds}_{\pm[timeTolerance]}\end{aligned}$$

This rule can be used if the monitored outdoor temperature is below the reference temperature (indicating a general need for using the heating unit as compensation for temperature drift). The effect of this rule encodes the assumption that the sun intensity captures the percentage of the indoor temperature that will be achieved by the sun, and the expected relationship between flow temperature and energy.

5.1.3 Adaptation History

Our knowledge models as described in the previous subsections encode the current knowledge about the system and environment, about goals and available adaptation options. Additional knowledge is needed to enable the evolution of adaptation logics based on the observed performance of the adaptation layer. We encode this knowledge in our *adaptation history*. This history is a bounded set of adaptation history objects that contain relevant information concerning a single rule execution. Such an object consists of the executed rule, the execution context (in terms of K_{Sys} and K_{Env}), the expected effect values, and the execution timestamp. Additionally, we include an evaluation status (`pending, effect_missed, effect_achieved`), the actually observed effect values, the evaluation timestamp that is used to detect deviations in the effect timing, and an equivalence class that encodes the order of magnitude of the observed deviation from the expected effect. The evaluation status expresses whether the evaluation is still running because the expected effect time has not passed yet (`pending`), whether the rule missed the effect expectations (`effect_missed`) or not (`effect_achieved`). These history objects build the data base for our *Observation-Based Learning*. Furthermore, they provide a basis for explaining adaptation decisions of the adaptation layer, and evolution steps of the evolution layer, as discussed in Section 5.3 and Chapter 7. To this end, we additionally add the violated goals and the overall distance to explain the reason for adaptation. For resource-efficiency, we use a bounded adaptation history. Thus, we propose to store the adaptation history objects in an additional *Explanation Base* when they are removed from the bounded adaptation history. This explanation base can be located in the system or on an external server.

Our *adaptation history* is either stored in a knowledge base within the evolution layer or in the adaptation layer knowledge base. This architectural decision depends on whether both layers are deployed on the same hardware and use the same memory (separated knowledge bases are not necessary) or whether they are distributed and thus, access to the adaptation layer knowledge base is more costly than access to a local knowledge base. In our figure of the framework architecture, we have decided for a single knowledge base to simplify the picture. This decision does not restrict our approach.

5.1.4 Adaptation Goals

We do not include the adaptation goals in our knowledge base that is located in the adaptation layer, because they do not belong to the knowledge that is collected at runtime. Still, they are not static as they can be updated at runtime by a goal manager (i.e. human or external system). As they represent important knowledge that is the base for all adaptation decisions, we briefly sketch our goal model here. We present more details on our expressive quantitative goal model in the next chapter.

Our goal model is a hierarchical representation of the current system goals, their subgoals, and the relationship between them. Thereby, goals can represent functional or

non-functional requirements. The goal model enables the analysis whether K_{Sys} together with K_{Env} satisfies the goals, and it allows for the quantification of how close the system together with the environment is to the system goals. To this end, we use a distance function $dist(K_{Sys}, K_{Env})$ which takes as input an abstract system and environment model and provides a quantitative distance between the system and the environment state towards the goals. We provide an automatic quantitative evaluation algorithm that enables the modular calculation of this distance. We use the distance function in the analysis phase for detecting adaptation needs and for planning and learning to measure the improvement that can be achieved by applying an adaptation rule. In case of runtime goal changes, we automatically provide a new distance function.

Example The system goals in our temperature control system are to keep the indoor temperature at $20°C$ during the day and between $16°C$ and $20°C$ during the night, as well as to minimize the energy consumption. Suitable *distance functions* for the temperature goals are, for example, the absolute value of the difference between the current and the desired indoor temperature (if we want to express a linear distance for each degree deviation), or a quadratic function that results in small distances for small deviations and grows rapidly for larger deviations. For the energy consumption, we can use an exponential function that captures a decreasing impact of adaptations while the goal is approached. Such a linear distance function for the temperature goals and an exponential function for the energy goal are automatically derived from our goal model as described in the next chapter. The derived functions can be replaced by user-defined functions like the quadratic function as illustrated in the next chapter as well.

In this section, we have described our knowledge models. We have designed them to enable the realization of a further meta-adaptation layer for evolving the adaptation logics, based on evaluation, learning, and verification. In the next section, we present our resource-efficient adaptation process and describe its interaction with our knowledge models in detail.

5.2 Adaptation Layer

The adaptation layer is responsible for ensuring goal satisfaction at runtime by adapting the managed system to the current situation. Our adaptation process is an instantiation of the MAPE-K feedback loop. In the following, we describe our major design decisions for each phase of the feedback loop and the interaction with the knowledge models in detail. The architectural overview of this layer is depicted in Figure 4.2 and the process details are shown in Figure 5.1.

5.2.1 Topology-Aware Monitoring

In the *monitoring phase*, the adaptation layer regularly retrieves runtime information of the managed system and the environment and stores this information in the knowledge

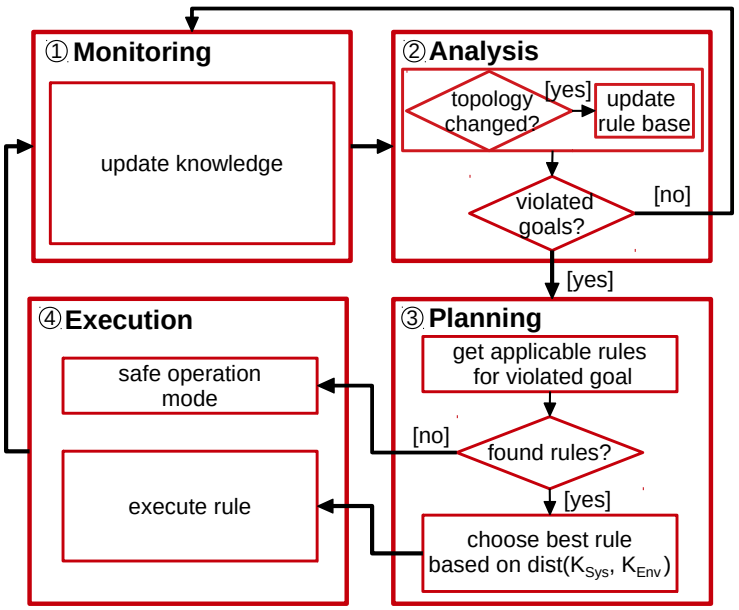


Figure 5.1: Adaptation Process in Detail

base (in the system model K_{Sys} and the environment model K_{Env}). We use polling for the cyclic monitoring as this enables us to use the monitoring answer also as heartbeat signal to detect component reachability. In the case of event-based monitoring where system components inform the monitoring component that relevant system/environment data has changed, the heartbeat algorithm can also be realized by a separate polling mechanism. If a component does not answer the monitoring request within a specified time, it is considered not alive/ not reachable any more and we remove it from the set of currently available components in K_{Sys}. Then, we update the adaptation rule base accordingly, i.e. all adaptation rules that include actions on the removed component are disabled. Thus, we are able to deal with component removal or connection failures at runtime. If the system topology is stable, monitoring could also be based on interrupts that signal that control data have changed. This would reduce the communication load and the computation load of the adaptation layer. Except for the topology management, the monitoring approach can be freely chosen in our framework. If components become available again or if new components are dynamically added to the system, they have to register and establish a connection to the monitoring component of the MAPE layer. To this end, we assume a registration component that executes a registration protocol in which required knowledge for adaptation is provided by the components. During registration, components provide their control parameters, the corresponding influence relation R and executable runtime models (RTM) that are used by our learning and verification. To avoid expensive learning in case of (current) resource restrictions, new components can also provide an initial set of adaptation rules. As these rules do not necessarily capture dependencies to other system components, they are verified in the

current system context, before being added to the set of active adaptation rules. We propose the same verification steps that we use for newly learned adaptation rules for the integration of those provided rules, namely rule effect validation and comprehensive system verification to ensure safety. If a registration component is provided, we are able to deal with dynamic topology changes at runtime.

After monitoring, the analysis phase is triggered for analyzing adaptation needs. If the rule accuracy evaluation uses the monitored data, the evaluation component is invoked to evaluate the accuracy of the current adaptation logic. Our rule accuracy evaluation can also have its own evaluation cycle time. In this case, it is not triggered by the monitoring of the adaptation layer.

5.2.2 Analysis of Goal Violations

In the *analysis phase*, we use our knowledge models to decide whether an adaptation is necessary or not. We base this decision on the quantitative "distance" of the current system state represented by K_{Sys} and K_{Env} towards the system goals represented by *Adaptation Goals* to achieve an automatic coupling between the current adaptation goals and the analysis. Thus dynamic goal changes are directly considered in the next analysis phase. The distance is calculated with our distance calculation algorithm as described in Chapter 6. If the evaluated distance exceeds a given threshold δ, i.e., some goals are severely violated, an adaptation need has been found and the planner is triggered to re-establish the goals. Our goal model also enables the specification of optimization goals, which are never fully satisfied. For these goals, we also calculate a distance that becomes exponentially smaller if those goals come close to an estimated optimal value. Here, the threshold δ can be used to specify whether further optimization may still be beneficial w.r.t. the required computation effort.

With our distance-based analysis, we support reactive, as well as proactive adaptation. Reactive adaptation reacts to observed goal violations, whereas proactive adaptation anticipates goal violations and adapts on time to prevent them. In the first case, we base our analysis on the distance between the currently monitored state and the goals. In the latter case, we use prediction models and base the analysis on the distance of this predicted state. In [KGG18a], we have presented an example for proactive production optimization based on lightweight environment profiles. Furthermore, our timed effect expectations within our adaptation rules provide a basis for latency-aware proactive adaptation. Latency-aware hereby refers to the latency between executing an adaptation and being able to observe the effect. In this thesis, we focus on reactive adaptation for two reasons: 1) the main goal of this thesis is to provide means for safe and explainable runtime evolution of adaptation logics, and 2) the design and runtime learning of accurate environment prediction models is a different and challenging research area that is out of the scope of this thesis.

5.2.3 Distance-Based Planning

```
1  chooseBestRule(){
2     minDistance ← currentDistance;
3     bestRule  ← NULL;
4     RULES  ← getCandidates (violated_goal);
5     RULES ← getApplicableRules (RULES);
6     for each r: RULES{
7        virtuallyExecuteCommands(r);
8        virtuallyApplyEffect(r);
9        dist  ← calculateDistance(virtualSystemState);
10       if (minDistance > dist){
11          minDistance  ← dist;
12          bestRule  ← r;
13       }
14       else if(minDistance = dist ∧ r.effect_time < bestRule.effect_time){
15          bestRule  ← r;
16       }
17    }
18    return bestRule;
19    }
```

Algorithm 5.1: Rule- and Distance-Based Planning Algorithm

In the *planning phase*, the best available adaptation rule for reestablishing violated subgoals is chosen if an adaptation is necessary. The problem of finding an optimal adaptation plan is an optimization problem. The objective is to maximize the fitness of the managed system, which means to minimize its overall distance to the goals. As goals may be contradicting, we actually have to solve a multi-objective optimization problem. As these problems are hard to solve, they are usually solved with heuristic techniques. In our rule-based setting, adaptation rules describe a final and usually small set of possible optimization steps that can be compared based on their effect expectations and the resulting distance. We propose to use a quantitative evaluation of the effects on the adaptation goals to capture positive and negative effects on possibly conflicting goals. As a result, we can choose the adaptation rule that minimizes the distance towards the adaptation goals the most. If the set of adaptation rules is considered too large for comparing all rules, heuristics like comparing only rules that effect the mostly violated subgoal, can be easily introduced into our planning algorithm.

In Algorithm 5.1, we provide the pseudocode of our rule- and distance-based planning algorithm. In a first step, we retrieve all rules that have an effect on currently violated adaptation goals (`getCandidates`). Then, we filter this set by checking which rules are applicable, i.e. rules with a guard that evaluates to true and that are not currently running (i.e., the time since their last execution is less then their specified effect times) (`getApplicableRules`). The subsequent rating of those rules is based on our quantitative evaluation of their expected effect on the system goals. To this end, we virtually apply adaptation rules on a copy of K_{Sys} and K_{Env} (`virtuallyExecuteCommands`) and estimate the resulting system and environment state based on the expected effect, which is encoded in the adaptation rules (`virtuallyApplyEffect`). Then we calculate the

distance of this expected system and environment state (`calculateDistance`). We apply these steps on all applicable candidate rules and compare their distance to the minimal distance that we have found so far (initially this `minDistance` is set to the current distance as we do not want to perform worse). In the end, the rule that minimizes the distance the most is chosen. If two rules achieve the same distance, we additionally compare their effect times and choose the one with the smaller effect time (cf. lines 14 -16). If no suitable rule is available, e.g. due to a situation that was not captured by the existing rules, the learning component is invoked to infer a new adaptation rule. The planning algorithm can be extended in future work to enable plans with more than one rule. To this end, independent rules could be executed at the same time. To be independent, rules should not deactivate each other (their effects should not influence the guard of the other rules) and their effects should be disjunct to enable a separate effect prediction or their effect predictions allow for combined effect prediction by building the sum of their effects. A more sophisticated planning algorithm could also examine the sequential execution of rules. However, this is computationally more expensive and is only interesting if the computational overhead for planning is less than the overhead for smaller monitoring cycle times.

The main advantage of our distance-based expected effect evaluation is its resource efficiency. With our approach, we do not need to perform regular resource-intensive simulation to evaluate the context-dependent effect of adaptation options on the adaptation goals in the current state. Instead, we use goal-independent effect expectations that may depend on the context, which is encoded in the guard of adaptation rules, and evaluate the distance from the resulting state to the goals. The notion of distance together with the explicit encoding of effect expectations in our rules enables us to re-evaluate planning results and to reuse adaptation rules in case of runtime changes of the system goals.

5.2.4 Execution

Finally, if a suitable adaptation rule was found, we apply the rule by setting the corresponding control parameters in the system components within the *execution phase*. Furthermore, we record the applied rule, the execution context and the intended effect in the adaptation history for later rule evaluation, which takes place in our evolution layer. If, however, no adaptation rule is applicable or no rule approaches the system goals sufficiently, we assume that the system can be set into a safe operation mode. This mode does not have to be a single fail-safe mode, but can also be some kind of graceful degradation that depends on the violated goals and the severity of goal violations. In [ZKGG19], we have presented an example for a degradation controller for autonomous car platoons. Similar controllers could be used to enhance the quality of operation during the safe operation mode. The system resides in this mode until a suitable adaptation rule has been learned or the environment behavior changed itself in a way that the goal is approached again or another adaptation rule is applicable.

In this section, we have described our resource-efficient adaptation process within our framework and have sketched the usage of our knowledge models within this process. The main advantages of our adaptation layer are a) flexible and comprehensible adaptation logics, b) efficient adaptation planning, and c) robustness w.r.t. dynamic goal changes.

5.3 Explainability of Adaptation Decisions

Comprehensibility of autonomous decisions enables trust in autonomous systems. To achieve this, we use comprehensible adaptation rules and particularly rely on tracing of autonomous decisions and the information that led to the decision. To this end, we argue that an explanation for an autonomous adaptation decision should contain the following aspects: a) context of the decision, b) cause and c) expectation of the effect w.r.t the cause. Furthermore, we add d) result, to enable the evaluation of the effect of the autonomous decision.

Transferred to our framework, we trace the following aspects for each adaptation decision: a) execution context (i.e. the current values of K_{Sys} and K_{Env}) and execution start time, b) the violated goals and the overall distance, c) applied rule and expected effect (i.e., the expected values after rule execution), and d) actually observed effect and evaluation timestamp, rule accuracy evaluation results (rule status (effect_ achieved, effect_missed)and deviation equivalence class). All of these information, are captured in our *adaptation history object*. Thus, we do not delete *adaptation history objects* after learning, but, instead, move the objects to our *explanation base* to keep them as basis for generating explanations. Each object in our explanation basis can already be seen as explanation for a single adaptation decision. While this explanation format is sufficiently comprehensible for self-adaptive software engineers, further processing to generate textual explanations may be beneficial for non-experts and could be part of future work. If the goal model changes at runtime, we also store the old model together with the timestamp of the change in our explanation base.

Summary In this chapter, we have described our knowledge models and the single phases of our resource-efficient self-adaptation. Our main contributions in the adaptation layer are the resource-efficient and modular encoding of the adaptation logics in our timed adaptation rules, the distance-based evaluation of the adaptation goals, and the flexible connection between adaptation rules and adaptation goals via an explicit encoding of the expected effect of each adaptation rule. By additionally storing information about adaptation decisions and their expected effect in the knowledge base and in the explanation base, we enable the continuous evaluation of the accuracy of adaptation rules and build a basis for explaining adaptation decisions at runtime. Our adaptation process is based on the widely used feedback-loop architecture MAPE-K. Other architectures like the O/C architecture used in Organic Computing [RMB$^+$06] can be mapped to

the MAPE-K architecture. Thus, this decision does not limit the applicability of our framework.

6 Goal Model

How can we quantify the effect of adaptation options with respect to context-dependent and possibly conflicting system goals with interdependencies?

We require our self-adaptive systems to autonomously cope with changing system goals at runtime. To achieve this, we propose to design *goal-aware* systems that dynamically manage their current system goals in a model and take them into account when making autonomous decisions. To this end, they evaluate their system goals at runtime. In the context of this thesis, system goals denote functional and non-functional requirements that can be evaluated in a runtime state of the system.

With the increasing complexity of self-adaptive systems also the complexity of their goal models increases. They are required to capture complex, possibly hierarchical goal structures and relations between goals such as dependencies, priorities, and conflicts. At runtime, it might be impossible or very costly to satisfy all requirements due to environment uncertainties (e.g. sun and wind energy in smart grids) or physical aspects (e.g. wear or hardware failures). To enable complex autonomous decisions that incorporate a fine-grained balancing of the cost-benefit ratio of autonomous decisions w.r.t. predictions on the future environment behavior, a qualitative judgment (e.g. satisfied, partially satisfied, not satisfied) of the satisfaction of system goals is insufficient. For example, in case of a qualitative judgment goals that are labeled with "partially satisfied" cannot be further distinguished w.r.t. their respective goal satisfaction. In contrast, a quantitative judgment precisely captures the degree of satisfaction of each goal and thus enables the choice of the most beneficial action in a systematic way.

In this chapter, we present our quantitative and context-dependent goal model as published in [KGG18d]. A former version of our goal model is presented in [KGLG17]. Our goal model captures the structure of goals and dependency relations between them. We enable the description of elementary leaf goals, parent goals, and the fine-grained description of parent-children relationships, including context-dependent importances. Our goal model combines essential modeling elements for describing and quantitatively evaluating runtime goals from existing standard goal modeling languages like, for example,

the Goal-oriented Requirements Language (GRL) [Int12], i* [Yu97] or KAOS [vLL00]. We provide additional modeling elements to describe context-dependent goal relations and importances. As a central concept, we employ local goal distance functions. They enable us to specify the dynamic calculation of the "distance" between runtime states (captured in the knowledge parts that describe the system (K_{Sys}) and environment (K_{Env}) state) and elementary (leaf) goals, and to specify how these are propagated to higher levels in the goal hierarchy in a precise, generic, context-dependent, and modular way. We provide automatically derivable local distance functions for each goal type and an efficient distance calculation algorithm in terms of linear computational complexity, which takes the specified dependencies and conflicts between goals into account. Thereby, intended conflicts between goals can be resolved by balancing their distances to achieve a Pareto optimum. In our model, goals can easily be added, removed, and adjusted without the need to change the distance calculation.

In our framework, we use our goal model to (1) represent adaptation goals, e.g. setpoints or optimization objectives, and their dependencies at runtime, and (2) to provide a modular distance function that enables us to quantify the context-dependent achievement of system goals during analysis, planning and learning of adaptations. Hence, it provides a basis to find an optimal trade-off between multiple, possibly conflicting context-dependent goals. The notion of distance together with the explicit encoding of effect expectations also enables us to re-evaluate planning results and to reuse adaptation rules in case of runtime changes of the system goals. Thus, we achieve robustness w.r.t. dynamic goal changes.

Assumptions We make the following assumptions.

1. All goals can be mapped to quantifiable system goals, i.e. goals that can be mapped to restrictions (setpoints or thresholds) or optimization of the value of key performance indicators. Those indicators have to be based on system and environment parameters that describe aspects of the environment like, e.g. *time of the day, outdoor temperature,* or *sun intensity*, or aspects of the system like, e.g. *energy consumption*. With that, we do not explicitly address the completion of tasks, as e.g. done in [DM10].

2. There is an interface that continuously provides monitored values of system and environment parameters at runtime. This enables the quantitative evaluation of goals in the current system state at runtime. To this end, we assume that all goal-relevant system and environment parameters are known at design time or are registered in the system in case of runtime changes of the goal model.

The rest of this chapter is structured as follows. We present the structure of our generic goal model and describe its use with our illustrating case study of a smart temperature control system in Section 6.1. In Section 6.2, we describe our automatic

quantitative global distance calculation. We argue that our model is particularly well-suited for runtime management of system goals in autonomous systems. We support this in Section 6.2.3 by illustrating how the distance calculation can support the autonomous choice of appropriate adaptation actions.

6.1 Generic Quantitative Goal Model

In this section, we first present the general modeling elements and the hierarchical structure of our goal model. We achieve genericity of our goal model by employing generic local goal distance functions. To provide a certain degree of context-awareness, we enable importance factors and precedences to be context-dependent. Afterwards, in Section 6.1.2, we use our illustrating case study of a smart temperature control system to illustrate the modeling capabilities of our goal model. In Section 6.1.3, we provide a formalization of our goal model. It precisely captures the modeling elements and their meaning. Moreover, it enables us to define a uniform quantitative evaluation in Section 6.2, which captures the deviation of the current system state from an ideal system state that fulfills all (currently active) goals.

6.1.1 Modeling Elements

Our goal model organizes system goals in a tree structure enhanced with dependence and conflict relations. In Figure 6.1, we give an overview of our graphical notations for goal modeling elements. In Section 6.1.2, we illustrate how they can be used to design a precise goal model from informal requirements.

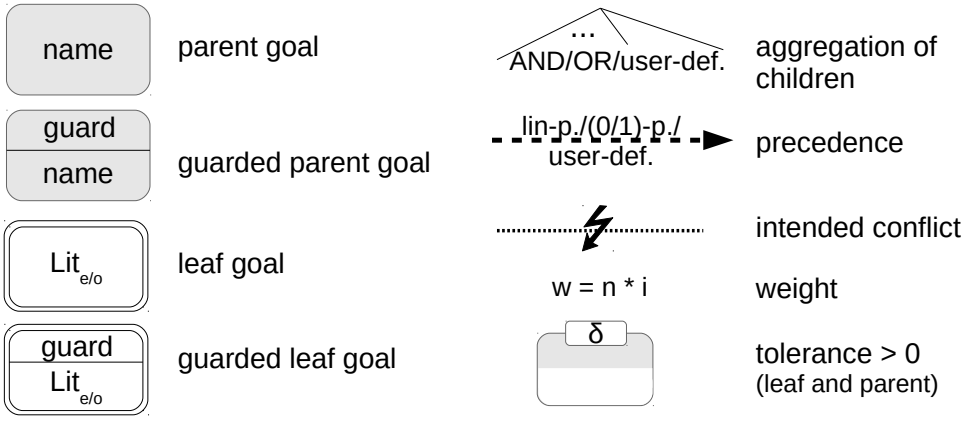
We distinguish between leaf and parent goals. Leaf goals (depicted with white, double-framed boxes) describe explicit restrictions on the system and environment variables *Var*. With these *variables*, we refer to parameters whose values can change over time and that can be monitored. They can describe environment parameters of the system like, for example, *time of the day, indoor and outdoor temperature*, or system parameters like, for example, *heating parameters*. Note that these variables can describe controllable (*heating parameters*), indirectly controllable (*indoor temperature*), or uncontrollable parameters (*outdoor temperature*). Goal restrictions are expressed with literals ($Lit_{e/o}$) that specify, for example, setpoints (e.g. indoor temperature temp_in $= 20$), thresholds for one parameter, a relation between several parameters, or an optimization, i.e. the minimization or maximization of an (arithmetic) expression (e.g. min(energy, $0, 52$)), which expresses that the energy consumption should be minimized within an expected value range of $[0,$ MAX_FLOW - MIN_FLOW$] = [0, 52]$). Leaf goals are the elementary goals that are expressed directly on system and environment variables. To capture the gradual deviation of a current system state from a leaf goal, a local goal distance function is attached to the leaf goal (see Section 6.1.3), which can, for example, be derived from the goal literal as described in Section 6.2.2. To combine subgoals, parent goals (depicted

with grey, single-framed boxes) can be introduced. They hierarchically aggregate children goals with a specified aggregation type (e.g. AND/OR, depicted at the edges between parent and children goals). To enable context-dependent activation and deactivation of goals, each goal has a guard (default is *true* and omitted in the graphical representation). A guard describes a condition that has to be fulfilled by the current system and/or environment state to activate the corresponding goal. Thus, the goal is only relevant, if the guard is fulfilled. The weight w of each subgoal describes the impact of goal deviations on the aggregated deviation in the parent goal. It is defined by the product of a normalization factor n and a context-dependent importance factor i. The normalization factor n is supposed to normalize the locally calculated distance to a suitable interval such as $[0, 100]$, and the importance i additionally describes the severity of a deviation from the goal in different contexts. A tolerance value δ (depicted on top of a goal, omitted for the default value of 0) describes to which extent a deviation is still acceptable and, thus, not propagated to higher-level goals. To express complex relations between goals, we include two kinds of dependencies. First, context-dependent weighted precedence describes the influence of the degree of satisfaction of a preceding goal on the importance of a preceded goal (depicted by a dashed arrow in the direction of influence, labeled with the type of precedence). Precedences can be used to specify that a certain goal is only relevant, if the preceding goal is satisfied ("$0/1$"-precedence) or that it becomes more relevant, the more the preceding goal is satisfied (lin-precedence). Second, conflicts (depicted by a dotted edge with a lightning symbol) describe that two goals are possibly conflicting (e.g. min vs max or different target values for the same parameter). Adding a conflict edge in the goal model expresses that the designer is aware of the (possibly intended) conflict. Such a conflict might occur due to conflicting high-level goals (e.g., energy vs performance) that can be mapped to contradicting restrictions of the same observable parameter (e.g., in our illustrative example a higher room temperature might increase comfort but also increases energy consumption). By explicitly encoding the conflict in the goal model, we enable the impact evaluation of adaptation decisions on both goals. The conflict edge does not change the distance evaluation. It is only used to connect conflicting goals. To increase flexibility, we allow the designer to provide user-defined functions for the distance of leaf goals, for aggregation types of parent goals, and for context-dependent importance factors and precedences.

In the following, we first illustrate the modeling capabilities of our goal model with our illustrating case study. Then, in Section 6.1.3, we formalize our goal model to precisely capture its semantics.

6.1.2 Illustrative Example

To model the goals of our illustrating case study, we map the requirements of our smart temperature control system (cf. Figure 6.2) to subgoals and define quantifiable leaf goals



$Lit_e := a \bowtie d \,|\, a \in [d_1, d_2]$ with a being an arithmetic expression over Var, $\bowtie \in \{\leq, =, \geq\}$, and $d, d_1, d_2 \in \mathbb{R}$ with $d_1 < d_2$.
$Lit_o := min(a, min_v, max_v) \,|\, max(a, min_v, max_v)$ with $min_v, max_v \in \mathbb{R}$ and $min_v < max_v$, defining an approximate anticipated range of a.

Figure 6.1: Modeling Elements

for each requirement. In the following, we systematically derive the goal model from the requirements specification.

Requirements

The main high-level requirements of our temperature control system are optimal temperature control and energy efficiency. Temperature control is the most important aspect. Thus, the more severe the temperature goal is violated, the less important the energy goal becomes.

R1 The indoor temperature should be $20°C$ at daytime.

R2 If the outdoor temperature is below 16, the indoor temperature should be reduced to $16°C$ at nighttime.

R3 To save energy, the indoor temperature should not be reduced in nights with an outdoor temperature greater or equal to $16°C$, but stay in an interval of $16°C$ to $20°C$.

(a) Temperature Requirements

R4 To reduce operational cost, the energy consumption should be minimized.

R4a Energy can be minimized by lowering the target temperature for the heating unit.

R4b Energy can be minimized by reducing the power of the air conditioning.

(b) Energy Requirements

Figure 6.2: Requirements for our Smart Temperature Control System

Structured Modeling of Requirements

As the requirements fall into the two categories of a) *Temperature Control* and b) *Energy Efficiency*, we split the overall system goal into respective subgoals using AND decomposition. The temperature regulations can be split into three exact leaf goals (R1 - R3), which depend on the environment context. These environment contexts can be modeled as guards. With that, we can use the AND decomposition to express that each leaf goal should be satisfied within its environment context. As the guards are exclusive, only one of these goals can be active at the same time, thus, we do not have conflicts here.

For the *Energy Efficiency* goal, we consider two modeling alternatives: Firstly, we specify a minimization goal for an observable energy consumption parameter (R4). This modeling decision requires an explicit modeling of the energy consumption and has the advantage of also capturing other unmentioned influences on the energy consumption. Secondly, we split the *Energy Efficiency* goal into two minimization goals according to the specifications R4a AND R4b. This approach does not include other influences on the energy consumption, but does not require additional monitoring capabilities. This example shows that our goal model enables a fine-granular modeling of requirements and requires an early consideration of design decisions. In our running example, we use the first modeling alternative of capturing the energy goal to enable online learning of the actual influence of adaptations on the energy consumption.

In Figure 6.3a, we present the resulting goal model for the first alternative of modeling the energy goal. Here, we assume that the energy consumption parameter depends on the flow temperature of the heating or the cooling temperature of the air conditioning. The energy consumption of the heating is abstractly modeled by energy(flow_temp) $=$ flow_temp - MIN_FLOW, and the energy consumption of the air conditioning is assumed to be constant for each mode with 0 for mode off, 20 for mode on and 25 for mode power. The resulting values can range from 0 to 52. In Figure 6.3b, we present the goal model for the second alternative. We capture the applied power of the air conditioning by its mode and use the mode range as approximate interval: airCon_mode $\in [0, 2]$. For the indoor temperature, we assume an approximate interval of: temp_in $\in [15, 30]$.

For simplicity, we set all tolerance values to 0. If they are greater than 0, they only reduce the observed distances because deviations within the tolerance are not propagated to higher-level goals.

Dependencies and Conflicts Our requirements include one dependency, which is captured at the end of the requirements description above. It can be modeled as a linear proportional precedence (lin-precedence) that describes a linear influence on the importance of a goal. This means that the contribution of *Energy Efficiency* to the overall goal depends on the quantitative degree of satisfaction of *Temperature Control*.

Thus, the more severe temperature regulations are violated, the less important energy efficiency get.

The requirements include a conflict between the exact restrictions of requirements R1, R2 and R3 and the minimization goal of requirement R4a. To indicate that this conflict is on purpose to enable a balancing between temperature and energy, we add conflict edges in our alternative version (Fig 6.3b).

The requirements of our temperature control system do no include all of our modeling elements. In Chapter 9, we evaluate our distance calculation on the goal model using the example of an autonomous drone delivery system. There, we also use "0/1"-precedence, an OR-aggregation, and context-dependent importance values.

Priorities and Weights To capture the priorities of goals, we use subgoal specific weights in our goal model that consist of a possibly context-dependent importance factor and a normalization factor to obtain distance values in the interval $[0, 100]$ as described in detail in Section 6.1.3. Deriving the context-dependent importance factors of subgoals from requirement specifications is hard. In our temperature case study, we manually specify the following qualitative priority: *Temperature Control* $>$ *Energy Efficiency*. For the remaining leaf goals in AND-aggregations, we use an equal importance as they are exclusive. This prioritization is captured in our weights in Figure 6.3a and Figure 6.3b. We have manually chosen the exact value of importances for our example in a way that the resulting distances match our expectations.

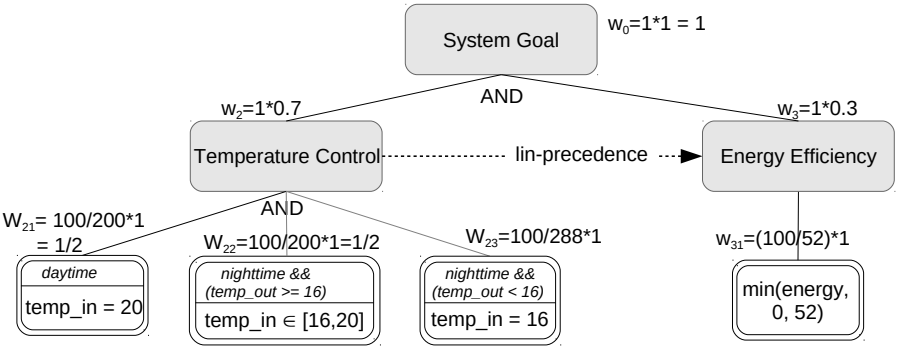
In this subsection, we have shown how our modeling elements can be used to construct a modular and quantitative goal model from a set of structured high-level requirements. In general, the complexity of goal modeling depends on the amount and on the format (structure, clearness) of the given requirements. This complexity can limit the practical applicability of goal-based approaches if not supported by graphical modeling tools and guidelines. Furthermore, deriving the importance of subgoals is a challenging task, that should be supported by systematic prioritization approaches. In future work, we aim at integrating our modeling approach with systematic context-dependent prioritization approaches like, e.g. iterative pairwise comparison of subgoals ([SSDD17]). Identifying precedence relations between system goals is also challenging if those are not explicitly encoded in the requirements specification ("the more..", "unless").

In the following, we formally define all modeling elements and illustrate their precise meaning with our smart temperature control example.

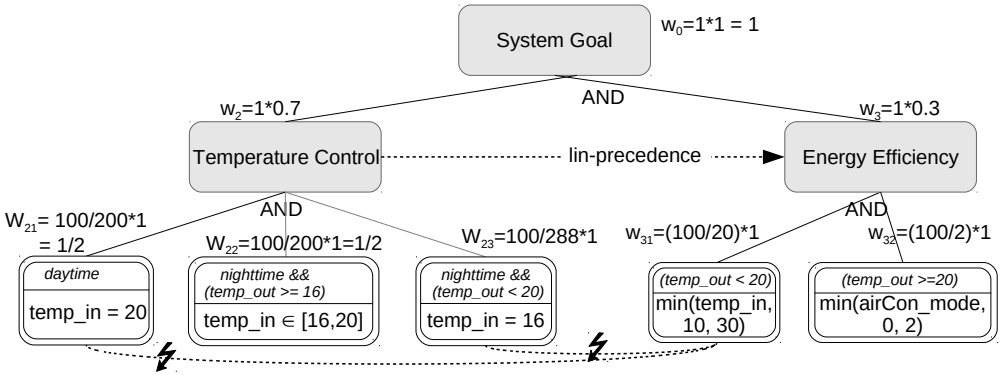
6.1.3 Formalization of the Goal Model

In this section, we formalize our goal model. This enables an unambiguous understanding and its systematic and precise evaluation in a system state (see Section 6.2).

To ensure a high expressiveness of our goal model, we design some of the modeling elements to be context-dependent, i.e. they may depend on the system state.



(a) Goal Model of the Smart Temperature Control System



(b) Alternative Way of Modeling the Energy Efficiency Goal

Figure 6.3: Alternative Goal Models for the Smart Temperature Control System

Definition (System States): A system state σ w.r.t. system and environment variables *Var* is defined to be a variable valuation, i.e. $\sigma : \mathit{Var} \rightarrow \mathbb{R}$. We denote the set of all such states as *State*.

Atomic Leaf Goals

Each leaf goal $l \in L$ carries a literal goal formula

$$\mathit{form} : L \rightarrow (Lit_e \cup Lit_o)$$

that describes the (boolean) satisfaction of the respective leaf goal and that can be evaluated in the current system state.

We distinguish exact and optimization goals. Exact goals are described using the following syntax.

$$\mathit{Lit}_e := a \bowtie d \,|\, a \in [d_1, d_2]$$

with a being an arithmetic expression over Var, $\bowtie \in \{\leq, =, \geq\}$, and $d, d_1, d_2 \in \mathbb{R}$ with $d_1 < d_2$. They enable the descriptions of restrictions on system and environment parameters. As optimization goals, we here focus on minimization and maximization of an arithmetic expression a.

$$\mathit{Lit}_o := min(a, min_v, max_v) \,|\, max(a, min_v, max_v)$$

In both cases, we assume that an approximate anticipated range of the observed value can be given via $min_v, max_v \in \mathbb{R}$ with $min_v < max_v$. This assumption is realistic, as boundaries can usually be estimated, e.g, based on physical constraints. For example, the energy consumption of our heating system cannot be minimized beyond zero and the maximal energy consumption of the system can be estimated. In the general case of $max(a, min_v, max_v)$, for example, a should evaluate to values in the interval $[min_v, max_v]$ and, then, the distance for this optimization goal should be small (close to 0) if it is close to min_v or smaller, and large for values close to max_v or larger.

Note that the leaf goal formula *form* is not directly used to evaluate a goal model. However, it can be used to derive basic goal distance functions automatically as described in the next section. If provided manually, it should be ensured that the distance is 0 if the literal formula evaluates to $true$ in the current system state. Generally, the distance function $\mathit{dist} : L \times \mathit{State} \rightarrow \mathbb{R}^+$ enables the quantification of the distance between a given system state $\sigma : \mathit{Var} \rightarrow \mathbb{R}$ and the respective leaf goal. While a distance value of 0 in a current system state can be achieved for exact goals, for optimization goals this is not possible. In the case of an exact interval goal $x \in [d_1, d_2]$, for example, the distance is 0 if the current variable valuation of x lies in this interval. However, if a goal specifies that a certain system variable should, for example, be maximized, the corresponding distance should never reach 0. This is because the value max_v only captures an approximate

reference point. Beyond this point, we aim at capturing that further optimization still has an effect on the distance, i.e. the distance decreases if a gets larger beyond max_v. This can be modeled, for example, using exponential curves. Note however, that the relaxed distance that is based on the goal's tolerance value can be 0. We give more details about general distance functions and present some particularly useful ones in Section 6.2.2.

Decomposition of Goals

A parent (i.e. non-leaf) goal $p \in P$ is decomposed into several subgoals. The parent-child relationship is given by a function *children* : $P \to \mathbb{P}(L \cup P)$ establishing the actual tree structure. The distance function of a parent goal *Dist* : $P \times \textit{multiset}_{\textit{fin}}([0,1] \times \mathbb{R}^+) \to \mathbb{R}^+$ describes the propagation of weighted calculated children distances to the level of the respective parent goal[1]. It operates on a finite multiset[2] ($\textit{multiset}_{\textit{fin}}$) of evaluated child distances (current (context-dependent) importance factors of all active children $\times$ their calculated distances) to achieve modularity and flexibility w.r.t. structural goal changes, i.e. new goals can easily be added to the children of a parent node. In many cases, AND and OR decompositions suffice to describe the parent-children relationship. However, also other parent-children relationships can be defined.

Guards and Context-Dependent Weights

Leaf goals L and parent goals P each have a guard *guard* : $(L \cup P) \to \textit{Formula}$[3], a distance normalization factor *norm* : $(L \cup P) \to \mathbb{R}^+$, a context-dependent importance factor *importance* : $(L \cup P) \times \textit{State} \to [0,1]$, and a tolerance value $\delta : (L \cup P) \to \mathbb{R}^+$. The context-dependent weight of a goal is given by the derived function
weight : $(L \cup P) \times \textit{State} \to \mathbb{R}^+$ with $\textit{weight}(g,\sigma) = \textit{norm}(g) \cdot \textit{importance}(g,\sigma)$.
Thus, weights capture both, the normalization and the importance.

We usually assume that the normalization factors normalize the calculated distances of goals to values between 0 and 100. This facilitates the comparison of their distances. However, we do not restrict our model to this. Thus, a designer has the freedom of normalizing distances to other ranges. Importance factors describe to which degree the respective goal contributes to its parent goal. Note that the importance factor depends on a system state. This enables us to dynamically adjust the importance of a goal in different system contexts. Explicit guards are therefore unnecessary, because they could be handled using context-dependent importance factors (returning 0 if the guard is not satisfied). However, for clarity, we have decided in favor of explicit guards.

Example As an example that illustrates the interplay of normalization and importance, consider the subgoals of the *Energy Efficiency* goal in our alternative model (Fig. 6.3b). One is speaking about the indoor temperature (values ranging from 10 to 30) and

[1] $[0,1]$ denotes the set of real values between 0 and 1.
[2] A multiset is a set that allows for multiple instances for each of its elements.
[3] *Formula* represents any predicate on the system and environment variables.

the other about the mode of the air conditioning (values ranging from 0 to 2). If the maximal deviation *max_dev* from these goals can be estimated, all distance values can be normalized to values $\in [0, 100]$ by assigning a normalization factor of $n = 100/$*max_dev*. As the importance i is given as percentage indication, the normalization only has to be performed for leaf goals as the combination of child distances then sums up to at most 100. In our example, we assume full knowledge of maximal deviations gathered from hardware specifications and experiences: For the indoor temperature, we assume temp_in $\in [0, 40]°C$, the energy consumption can reach a maximum of 52, and the air conditioning mode can have values between 0 and 2. As we use an individual quadratic distance function for our temperature leaf goals (as described in Section 6.2.3), we have to use the result of applying this function to the maximal values of temp_in. Thus *max_dev* for "temp_in $= 20$" is 200, and for "temp_in $= 16$", we get *max_dev* $= 288$. For optimization goals, the maximal deviation is given by the specified expected value range, thus the maximal deviation from these goals is $max_v - min_v$.
Note that for AND-goals dynamically changing importance factors do not introduce a problem for the suitable calculation of the parent goal even if the sum of (dynamic) importances of all subgoals is not exactly 1. As we base the distance function *Dist* of a parent goal on a multiset of child distances together with their individual importances, we can define a normalization on the children importances as described in Section 6.2.2.

Context-Dependent Weighted Precedence

With context-dependent weighted precedences given by *precBy* : $(L \cup P) \to \mathbb{P}((\mathit{State} \times \mathbb{R} \to [0, 1]) \times (L \cup P))$, we can describe that the importance of a goal g depends on the current distances of the preceding goals. For $(cdwp, p) \in$ *precBy*(g), we use an arrow from p to g as modeling element in our visual representation of goal models (e.g. the precedence between *Temperature Control* and *Energy Efficiency* in Figure 6.3a). The function $cdwp$: *State* $\times \mathbb{R} \to [0, 1]$ describes the absorption on the importance of the preceded goal, whose value $cdwp(\sigma, d)$ depends on the current state σ and the calculated distance d (see Section 6.2) of the preceding goal. Precedence can, for example, be used to describe priority of goals in an AND decomposition. This is captured by "0/1"-precedences. Then, the function $cdwp$ would return 1 if the preceding goal distance is 0 and 0 otherwise. For the definition of, for example, linear precedences that influences the importance of the preceded goal proportionally based on the distance of the preceding goal, we define that $cdwp(\sigma, d) = (100 - d)/100$. We require that precedences do not conflict with implicit dependencies of the tree itself. Thus, the intersection of the transitive closure of tree arrows and the transitive closure of precedence arrows is required to be empty. Furthermore, we require that there are no cyclic dependencies between precedences. This is important for our quantitative evaluation (see Section 6.2) to be well-defined.

Example In our smart temperature example, we use a linear precedence between *Temperature Control* and *Energy Efficiency* as described in Section 6.1.2.

Conflicts

Another important dependency between goals are conflicts given by the symmetric relation *conflict* : $(L \cup P) \rightarrow \mathbb{P}(L \cup P)$. It applies to goals that restrict the same system or environment variables in possibly conflicting ways. While such goals can be desirable for expressing an envisioned balance between parameters, unintended goal conflicts need to be detected and resolved. Based on the given set of system and environment variables *Var*, a straight-forward analysis could automatically detect conflicts between goals restricting the same variable. As a result, a warning to the system designer or administrator could be thrown. Unintended conflicts currently have to be resolved manually.

Example In our example, all subgoals of *Temperature control* specify exact values for the indoor temperature temp_in, while the subgoal of *Energy Efficiency* in Version 2 (Fig. 6.3b) specifies to minimize the indoor temperature in order to reduce energy consumption. Here, the conflict is on purpose. Both goals contribute to a common parent goal (*System Goal*) such that its optimal distance is balanced between both according to the given importance values. This is similar to a Pareto optimum. If, however, the system should guarantee that the exact temperature value requested by the user of the heating is usually achieved, the conflict is probably unintended. To resolve this conflict, the corresponding energy subgoal could be deleted to ensure that energy reduction will not collide with heating settings. If unresolved, our distance evaluation can, however, still be used to choose a temperature that optimizes the overall goal distance at runtime.

In this section, we have presented our hierarchical and generic goal model, its use in an example, and its formalization. In the next section, we present our modular quantitative distance calculation algorithm, which enables the efficient quantitative evaluation of a goal model in the current system state.

6.2 Global Distance Calculation

Our formalization enables the definition of an unambiguous semantics for the goal model in terms of a global distance calculation that can automatically be performed. In this section, we first present our distance calculation for our goal model (Section 6.2.1). Second, we present suitable local distance functions for all leaf goal types, which can be derived automatically (Section 6.2.2). Moreover, we provide example distance functions for parent goals. Finally, we present and discuss the calculated global distance for our smart temperature control system in example system states (Section 6.2.3).

6.2.1 Distance Calculation Algorithm

Our distance calculation is realized as a bottom-up algorithm that starts at the leaf goals and propagates the results along the edges in the tree to the parents. Hereby, guards and precedence relations, tolerance values, and the type of decomposition are taken into account. In the following, we describe the general distance calculation algorithm of leaf and parent goals in terms of recursive functions. By applying the evaluation algorithm on the root node of a goal model, the goal tree is traversed until the leaf goals are reached for which the individual distances are calculated. These results are then propagated upwards in the tree again until the distance of the root is finally calculated.

Auxiliary Definitions Guards and (context-dependent) precedences can activate or deactivate goals if guards or preceding goals are not satisfied. In our distance calculation, we only consider goals that are currently *active*.

Definition (Active Goal): A subgoal is called active if a) its guard evaluates to *true* in the current system state $\sigma : Var \to \mathbb{R}$, b) if it is a parent goal, it has at least one active child, and c) the current goal importance and precedences do not render the goal irrelevant as captured by the derived importance *DImportance* defined below. Formally, this is captured as follows.

$$\begin{aligned} \mathit{active}(g,\sigma) = \sigma \vDash \mathit{guard}(g) \wedge \\ g \in P \to \exists c \in \mathit{children}(g).\, \mathit{active}(c,\sigma) \wedge \\ \mathit{DImportance}(g,\sigma) > 0 \end{aligned}$$

We define the derived importance *DImportance* to consist of the current importance factor of the considered goal and each of the precedence absorption factors.

Definition (Derived Importance):

$$\begin{aligned} \mathit{DImportance}(g,\sigma) = \mathit{importance}(g,\sigma) \cdot \\ mult(\{\!|\, cdwp(\sigma, eval(p,\sigma)) \mid (cdwp,p) \in \mathit{precBy}(g) \,|\!\}) \end{aligned}$$

Distance Calculation for Leaf Goals In general, the distance calculation of a leaf goal g_l is based on the value given by its local distance function $dist(g_l, \sigma)$, where σ is the current system state. First, it is checked whether the goal is *active* and whether the distance is still acceptable w.r.t. the tolerance value $\delta(g_l)$. If this is the case, the distance is set to 0. Otherwise, it is relaxed using $\delta(g_l)$ and weighted using $\mathit{weight}(g_l)$. Let g_l be a leaf goal and $\sigma : Var \to \mathbb{R}$ be a system state. Then, the distance $eval(g_l, \sigma)$ is calculated formally as follows:

$$\mathit{eval}(g_l,\sigma) = \begin{cases} 0 & \text{if } \delta(g_l) > d \vee \neg \mathit{active}(g_l,\sigma) \\ (d - \delta(g_l)) \cdot n & \text{otherwise} \end{cases}$$

where $d = \mathit{dist}(g_l, \sigma)$ and $n = \mathit{norm}(g_l)$.

Distance Calculation for Parent Goals For calculating the distance of a parent goal, we first filter its children to get all *active* children. Then, we collect the current

importance factors and current individual calculated distances of each active subgoal in a multiset and feed them into the parent distance function $Dist(g_p, \mathit{ChildDistances})$. This function combines the weighted distance values of its active children according to the decomposition type. The result is then further relaxed according to its tolerance value and weighted. Let g_p be a parent goal and $\sigma : Var \to \mathbb{R}$ be a system state. Then, the distance $\mathit{eval}(g_p, \sigma)$ is calculated formally as follows.

$$\mathit{eval}(g_p, \sigma) = \begin{cases} 0 & \text{if } \delta(g_p) > d \vee \neg \mathit{active}(g_p, \sigma) \\ (d - \delta(g_p)) \cdot n & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathit{Children} &= \mathit{children}(g_p) \\ \mathit{ActiveChildren} &= \{c \in \mathit{Children} \mid \mathit{active}(c, \sigma)\} \\ \mathit{ChildDistances} &= \{\!| \, (\mathit{DImportance}(c, \sigma), \mathit{eval}(c, \sigma)) \\ &\qquad \mid c \in \mathit{ActiveChildren} \, |\!\} \\ d &= \mathit{Dist}(g_p, \mathit{ChildDistances}) \\ n &= \mathit{norm}(g_p) \end{aligned}$$

Note that this calculation is only well-defined if precedence dependencies do not conflict with implicit dependencies of the tree itself. This means that the intersection of the transitive closure of tree arrows and the transitive closure of precedence arrows is empty. Furthermore, we require that there are no cyclic dependencies between precedences.

6.2.2 Extraction of Local Goal Distance Functions

Our distance calculation is based on individual local distance functions of leaf and parent goals. For leaf goals, we differentiate between exact and optimization goals. The distance of exact goals is 0 if the goal is satisfied, whereas a distance of 0 is never reached for optimization goals as they are never satisfied completely (assuming that the given target values are only approximations of the reachable minimum or maximum value). This has to be considered when defining a distance function for leaf goals.

Often, a basic local distance function is sufficient that is directly derived from the literal formula that captures the main intent of a leaf goal. For exact goals (literals from Lit_e), we derive the following local linear distance functions. They capture a linear distance between the variable in the current system state σ and the lower respectively upper bounds.

a) For literals of the form $a \bowtie d$ with a being some arithmetic expression over *Var*, $\bowtie \in \{\leq, =, \geq\}$, and $d \in \mathbb{R}$, we get:

$$\mathit{dist}(g_l, \sigma) = \begin{cases} 0 & \text{if } \sigma \vDash \mathit{form}(g_l) \\ |\sigma(a) - d| & \text{otherwise} \end{cases}$$

Example The derived local goal distance function of the "temp_in $= 20$" goal in our temperature example is the following.

$$\textit{dist}(\text{"temp\_in} = 20\text{"}, \sigma) = \begin{cases} 0 & \text{if } \sigma(\text{temp\_in}) = 20 \\ |\sigma(\text{temp\_in}) - 20| & \text{otherwise} \end{cases}$$

This can be simplified to $\textit{dist}(\text{"temp\_in} = 20\text{"}, \sigma) = |\sigma(\text{temp\_in}) - 20|$. However, we decided to use an individual distance function for the temperature regulations as described in 6.2.3.

b) For literals of the form $a \in [d_1, d_2]$ with $d_1, d_2 \in \mathbb{R}$ and $d_1 < d2$, we get:

$$\textit{dist}(g_l, \sigma) = \begin{cases} 0 & \text{if } \sigma \vDash \textit{form}(g_l) \\ min(|\sigma(a) - d_1|, |\sigma(a) - d_2|) & \text{otherwise} \end{cases}$$

To regulate the distance values for optimization goals, i.e. $\textit{form}(g_l) \in Lit_o$, such that the maximal distance can be determined for normalization, we expect the optimization goals to carry an approximate range for the result of its arithmetic expression. We include those interval bounds in our local distance functions for minimizing and maximizing an arithmetic expression. We require that the impact of changes towards the target value is high if the observed value is far from the approximate target value and small if it is close to the approximate target value. As described in the previous section, we aim at capturing that further optimization still has a positive effect on the distance. Thus, the distance value should never reach 0. In the case of a maximization goal, for example, this means that that the distance should still decrease if a gets larger beyond max_v. To achieve this, we use an exponential function. As resulting distance functions, we derive the following.

a) For $\textit{form}(g_l) = min(a, min_v, max_v)$:
$\textit{dist}(g_l, \sigma) := e^{(\sigma(a) - min_v) \cdot ln(100)/(max_v - min_v)}$

b) For $\textit{form}(g_l) = max(a, min_v, max_v)$:
$\textit{dist}(g_l, \sigma) := e^{(max_v - \sigma(a)) \cdot ln(100)/(max_v - min_v)}$

Both these functions are illustrated in Figure 6.4. In the case of a maximization goal (curve b), for example, the distance is 100 if the expression a evaluates to the minimal value min_v and it is 1 if it equals max_v. Thus, further optimization positively affects the distance. The intention of curve a) for expression minimization is analogue.

The local distance function of parent goals combines the weighted distance values of all active children according to its decomposition type. In most cases, AND- and OR-decompositions are sufficient. For AND-decomposition, all child distances influence the overall distance. Thus, we propose to follow a weighted sum approach. To this end, we multiply each individual distance (d) with its current importance factor (i). To

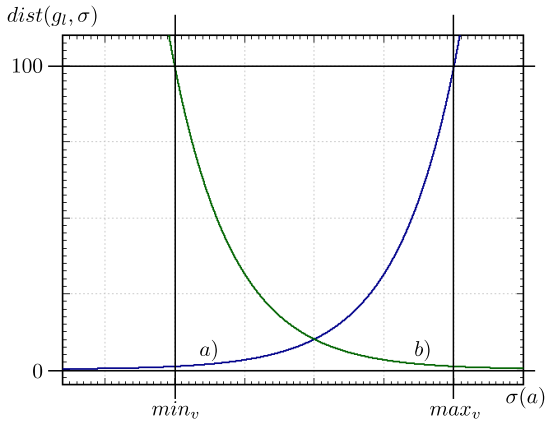


Figure 6.4: Local Distance Functions for Optimization Goals

additionally normalize the importances among all children to 1, we divide each individual importance by s, the sum of all current children importance factors.

Dist$(g_p,$ *ChildDistances*$) = sum(\{\!|\, i/s \cdot d \,|\, (i,d) \in$ *ChildDistances* $|\!\})$

where $s = sum(\{\!|\, i \,|\, (i,d) \in$ *ChildDistances* $|\!\})$

For OR-decomposition, only one of the subgoals has to be fulfilled. Thus, we take the minimal child distance as the result.

$Dist(g_p,$ *ChildDistances*$) = min(\{\!|\, i \cdot d \,|\, (i,d) \in$ *ChildDistances* $|\!\})$.

Example In our temperature example, the system goal has the AND aggregation type. Using the above distance function for AND, the resulting distance calculation of the system goal is as follows.

eval(*System Goal*, σ) $=$ sum$(\{\!|\, 0.7 \cdot eval($*Temperature Control*$, \sigma),$
$0.3 \cdot$ *eval*(*Energy Efficiency*, σ) $|\!\})$

6.2.3 Example

In this section, we use our smart temperature example to illustrate our distance calculation. We use our goal model of Figure 6.3a.

For the leaf nodes, we use the derived local distance functions as proposed above with the exception of the exact goals restricting the temperature. For these goals, we use an individual distance function that takes into account that moderate deviations from the target temperature (*TARGET*) have a small effect on the distance and larger deviations have a significantly higher effect. To achieve this, we use a quadratic function that fulfills the following requirements: the vertex is at $(TARGET|0)$ and a deviation of $2°C$ results in a distance of 2.

dist("temp_in $= TARGET$", σ) $= 0.5 \cdot (\sigma($temp_in$) - TARGET)^2$

For the parent goals, we use the proposed local distance functions for AND-decompositions as described above.

To illustrate how our distance calculation can be used to evaluate different adaptation options, we consider the following situation (system state): It is daytime (time $= 13$) and the outdoor temperature is $10°C$, the indoor temperature is $25°C$, the air conditioning is off (airCon_mode $= 0$), and the current energy consumption is 24. Thus, we have the following (incomplete[4]) system state.

$\sigma = \{\text{time} = 13, \text{temp\_out} = 10, \text{temp\_in} = 25, \text{airCon\_mode} = 0, \text{energy} = 24\}$

In this situation, we get an overall distance of ≈ 9.21 as weighted sum of the following subdistances.

$eval(\textit{Temperature Control}, \sigma) = 6.25$
$eval(\textit{Energy Consumption}, \sigma) \approx 16.11$

Now the temperature system has two options to decrease the indoor temperature: a) reduce heating or b) turn on the air conditioning. While turning the air conditioning on is explicitly encoded in our adaptation rules, reduction of the heating has to be performed by adjusting the heating parameters. For simplicity, we here omit the detailed adaptation steps and just compare the expected result of both adaptation options:

a) $\sigma = \{\text{time} = 13, \text{temp\_out} = 10, \text{temp\_in} = 20, \text{airCon\_mode} = 0, \text{energy} = 14\}$

$eval(\textit{System Goal}, \sigma) \approx 1.99$
$eval(\textit{Temperature Control}, \sigma) = 0$
$eval(\textit{Energy Consumption}, \sigma) \approx 6.64$

b) $\sigma = \{\text{time} = 13, \text{temp\_out} = 10, \text{temp\_in} = 20, \text{airCon\_mode} = 1, \text{energy} = 34\}$

$eval(\textit{System Goal}, \sigma) \approx 11.7$
$eval(\textit{Temperature Control}, \sigma) = 0$
$eval(\textit{Energy Consumption}, \sigma) \approx 39.06$

As we can see, reducing the heating has a positive effect on both, the indoor temperature and the energy consumption. Whereas using the air conditioning has a positive effect on the indoor temperature, but also increases the energy consumption, which leads to a significantly higher overall distance. Based on these values, the planning algorithm can choose strategy a) over strategy b).

In this section, we have presented our modular distance calculation algorithm with which we can quantify the deviation between a runtime state and the system goals. We have provided automatically derivable local distance functions for all considered kinds of leaf goals. To illustrate our distance calculation, we have shown how changes in the system behavior influence the distance values in the context of our smart temperature control example. In summary, the global distance calculation can serve as the basis for runtime evaluation of current and possible future system states within analysis and planning phases of self-adaptive systems, as well as for fitness evaluation during learning of new adaptation options.

[4] heating parameters and environment parameters not relevant for the calculation are omitted

Practical Applicability and Limitations of our Goal Model To enable goal-aware autonomous decisions the first crucial step is to create a corresponding quantitative goal model. This task includes several challenges and (currently) limits the practical applicability of our approach. The main challenges are a) to identify a quantitative key-performance indicator for each system goal (which may be difficult for aspects like security or safety) and provide a corresponding interface that gives access to runtime values of these indicators, b) to derive the structure of the goal model from explicit and implicit relations between informal requirements (as shown for a small amount of requirements in our illustrative example in Section 6.1.2), c) to derive precedences and context-dependencies from the requirements specification, and d) to identify suitable (context-dependent) importances for subgoals.

Runtime updates of our goal model require a manual mapping from new requirements to the existing goal-structure, which is straight forward in case of changes on existing goal elements (e.g. priorities, thresholds) and more challenging in case of introducing new goals as described for the initial creation of the goal model. The technical aspect of runtime updates is easily implementable due to the modular (object) structure of the goal model, as illustrated in Chapter 9. To further assess the practical applicability of our approach, further evaluations are necessary.

Summary In this chapter, we have presented our modular and generic goal model that provides an automatic quantitative goal evaluation capturing the "distance" between a runtime system state and the system goals. Our goal model comprises elementary quantifiable goals, hierarchic goals based on generic goal decomposition types, and complex relations between subgoals, which we model using individual goal guards and precedence relations. We have applied our goal model to our illustrating case study to illustrate its applicability and its suitability as a basis for autonomous decision-making. As described in Chapter 4, we use our distance evaluation to analyze whether autonomous decisions are currently necessary, to evaluate the cost-benefit ratio of given decision options, and as a basis for learning of new decision options. In the next chapter, we describe our runtime evolution of the adaptation logic in detail.

7 Runtime Evolution of Adaptation Logics

How can we enable continuous learning under safety, comprehensibility and resource restrictions?

With our adaptation layer, we enable a light-weight resource efficient self-adaptation that autonomously evaluates different adaptation options in the current situation, as described in the previous chapter. However, its success highly depends on the accuracy of the expected effect of the adaptation rules, and, thus, on the designers degree of uncertainty about the environment behavior that the system may face at runtime. Furthermore, the accuracy may decrease over time with changes in the environment behavior. We distinguish between two kinds of uncertainty:

1. the environment situations that can occur at runtime are known, but the effect of adaptations may depend on unknown aspects, i.e. the designer is uncertain about the quantification of the effect and about context-dependencies that may influence the effect

2. uncertainty about situations that may lead to goal violations and require adaptation.

In the first case, the expected effect of adaptations may not be accurate, which might lead to over- or undershooting while trying to re-establish system goals. In the second case, unexpected situations were not considered during the design of adaptation rules, which leads to the lack of applicable adaptation rules at runtime. In both cases, runtime observations can be used to adapt the adaptation logic to the actual operational context. As the environment behavior, the system structure and the system goals may evolve over time, a continuous adjustment is necessary to achieve a co-evolution of the adaptation logic.

While self-adaptivity realized through MAPE-K feedback loops [IBM04] has become a prominent approach to autonomously cope with changing environment behavior in complex dynamically evolving systems, the safe and explainable co-evolution of the adaptation logic itself is not sufficiently addressed in existing approaches. To overcome this limitation, our framework provides an evolution layer that detects and safely replaces inaccurate adaptation rules, learns missing rules, and verifies the evolved adaptation logic.

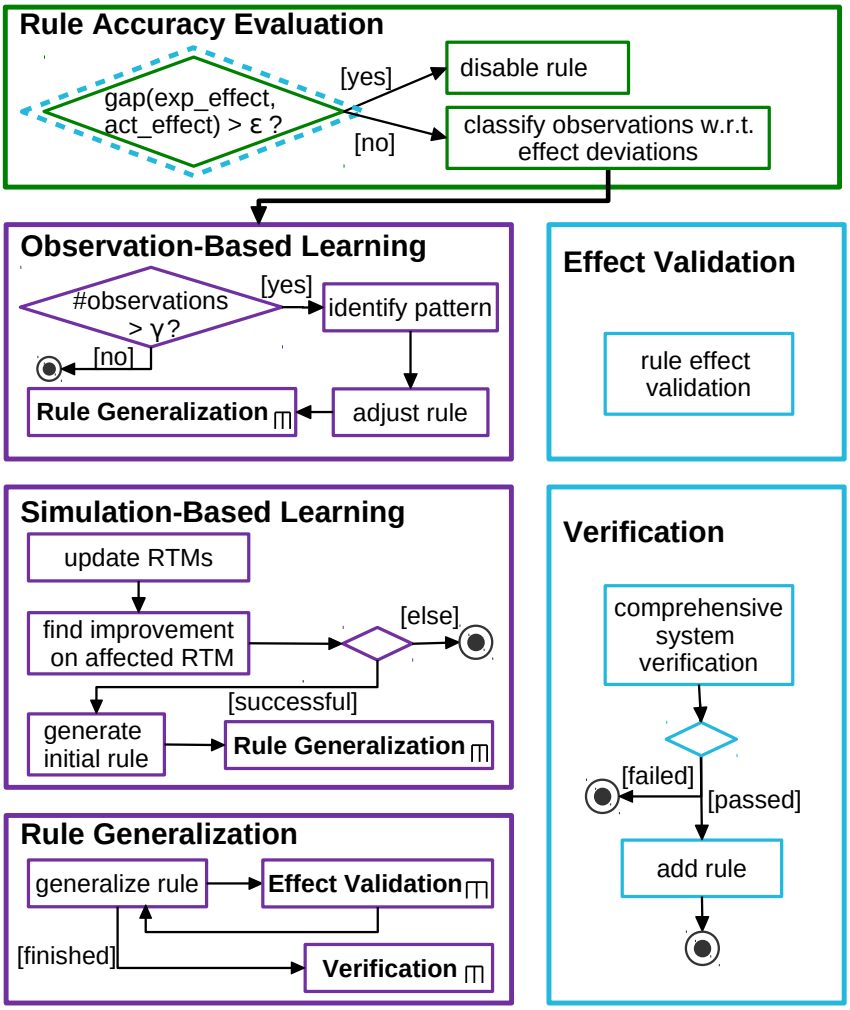


Figure 7.1: Evolution Process in Detail

To this end, we propose a *Rule Accuracy Evaluation*, an *Observation-Based Learning*, a *Simulation-Based Learning*, and a *Verification* approach as shown in Figure 7.1. In our *Rule Accuracy Evaluation*, we continuously compare the expected effect of previously applied rules and the actual values of system and environment parameters until the expected effect time (plus time tolerance) is expired. If the effect did not occur, we calculate the gap between the expected and observed effect. If the gap is too large (greater than a rule-specific accuracy tolerance ε_r), this rule is disabled to ensure the correctness of planning results. If, however, a smaller deviation is observed, we propose to keep the rule in the rule base, and to store and use previous evaluation results to learn from observations instead of using costly learning via model simulation. Our accuracy evaluation can also be seen as a runtime monitoring approach to ensure the correctness of the adaptation layer. Additionally, the retracing of adaptation decisions and their actual effect also help to gain a deeper understanding of autonomous decisions and thus, supports the explainability of the adaptation layer. In our *Observation-Based Learning*, we analyze whether some recurring pattern can be detected in the observed effect deviations.

If, for example, the effect deviation of a certain rule is always the same, or if this deviation depends on some external environment conditions, this knowledge can be used to *adjust* the existing adaptation rules. To this end, we propose to split the original adaptation rule into more specific rules that capture context-dependent effects by refining the guards of those rules according to the observed context conditions and by adding the observed deviations to the expected effect of the respective context-specific rule. Adjusting rules reduces discrepancies between future model predictions and the actual system behavior. However, observation based learning cannot be used to discover suitable adaptations for situation that were not considered during the design of adaptation rules. This is addressed in our *Simulation-Based Learning*. We propose to use learning on model simulations to determine promising adaptations for the currently observed environments. To minimize the resource consumption that is necessary for such a search-based learning, we iteratively generalize the learned solutions to achieve more generic rules that can be reused in similar situations. As a basis for the explainability of runtime changes on the adaptation rules, we store relevant evaluation and learning results in our explanation base.

In the following, we first describe our *Rule Accuracy Evaluation*, and then our learning approaches *Observation-Based Learning*, and *Simulation-Based Learning*. With the combination of these three parts, we achieve an explainable runtime evolution of the adaptation logic. In the end of this chapter, we discuss the stability of our rule evolution approach. To ensure correctness of learned rules, we perform verification of the learned rules before applying them on the running system. Our verification approach is described in the next chapter.

7.1 Rule Accuracy Evaluation

To detect inaccurate effect expectations, we continuously observe and evaluate the actual effect of applied adaptations. To this end, we evaluate whether the expected effect occurs in time or within a rule-specific acceptable delay $timeTolerance$. This tolerance is motivated by the fact that the effect is evaluated on monitored data and the values depend on the monitoring cycle time. If there is a gap between the expected and the actual effect, we check whether the *gap* is smaller than a rule-specific tolerance value ε_r. We assume that ε is the maximal deviation that is tolerable for planning. If this threshold is exceeded, we disable the corresponding rule to ensure correctness. If, however, a smaller deviation is observed, the rule can still remain in the rule base and observed effects of several executions can be compared to learn recurring patterns, e.g. context-dependent deviations. This status tracking for applied adaptations is described in Section 7.1.1. Note that the tolerance ε_r on the one hand has to be large enough to cope with the fact that there is a certain gap between the actual behavior and the model level, and, on the other hand, has to be small enough to allow for precise re-establishing of system goals.

In preparation of our *Observation-Based Learning*, we classify the observed effects of safe rule executions (with gap $\leq \varepsilon_r$) w.r.t. the observed deviation from the expected effect (in Section 7.1.2). To this end, we require the designer to provide a fuzziness value λ_p for each monitored parameter that specifies the deviation from the expected effect that is still considered as equivalent. This classification enables a fast detection of recurring effect deviations.

Example For our running example of a smart temperature control system, we assume that the concrete environment behavior is not entirely known at design-time, but can be abstractly modeled with a heating curve that describes the relation between the flow temperature of the heater and the current outdoor temperature on the indoor temperature. As this is a very simple model, it is likely that this relation is not accurate in every situation (e.g., high sun intensity) or that it may change due to changes in the construction of the building (e.g., thermal insulation) or in the characteristics of the radiator (e.g., installation of a new radiator). The former situation leads to the observation that the adaptation rules for adjusting the parameters of the heating curve always miss their effect if the sun intensity was high. The latter situation leads to the observation of a general shift in the actual effect of these rules. We use the scenario of an radiator exchange to illustrate our *Rule Accuracy Evaluation* and *Observation-Based Learning*.

7.1.1 Status Tracking

```
1  input: adaptation history
2  begin
3    foreach pending adaptation rule
4      DEVS ← calculateDeviation(expectedEffect,currentState);
5      add DEVS to history object;
6      effectOccurred ← checkFuzzyness(DEVS);
7      dueDate ← executionStartTime + effectTime;
8      if(effectOccurred && currentTime <= dueDate + timeTolerance)
9        status ← effect_achieved;
10     else if(currentTime > dueDate + timeTolerance)
11       status ← effect_missed;
12     if(status != pending)
13       evalTime ← current time;
14       saveCurrentState();
15     end_if
16     if(status = effect_missed ∧ ExceedsGap(DEVS)) disableRules();
17   end_for
18   moveChangedObjectsToEvaluationDataBase();
19 end
```

Algorithm 7.1: Rule Status Evaluation Algorithm

To enable rule accuracy evaluation, we create a new *adaptation history object* in our knowledge base for each execution of an adaptation rule. This object contains the

applied rule, the execution context (i.e. the current values of K_{Sys} and K_{Env}), the expected resulting effect (i.e., the expected values after rule execution), the execution timestamp, and a rule status. Its status is initialized with `pending`. In our `rule status evaluation`, which is given in Algorithm 7.1, we continuously (with a cycle time that can be independent from the monitoring cycle time) check for all `pending` rule executions in our *adaptation history* whether the expected effect has occurred in time. Based on this, we update the status of each history object, which expresses whether the evaluation is still running because the expected effect time has not passed yet and the effect has not occurred (pending), whether the rule missed the effect expectations (`effect_missed`) or not (`effect_achieved`). To this end, we first calculate the deviation between the expected and the current values of each effected parameter and store the result for later analysis in `DEVs` (lines 4, 5). Then, we compare these deviations to the parameter-specific fuzziness value λ_p and check whether all deviations are within the allowed range, i.e. the effect occurred (line 6). If this is the case within the expected effect time, we set the status to `effect_achieved` (lines 7-9). If the expected effect time passed, we allow for a given additional acceptable delay `timeTolerance` for the effect to be visible. If the effect has not occurred yet, we check whether the expected effect time has already passed. In this case, the status is set to `effect_missed` (lines 10-11) and the rule is no longer observed. Otherwise, the rule should be re-evaluated later (still pending). For all rules for which the status was changed, we save the current values of K_{Sys} and K_{Env} (current state) and the current timestamp (lines 12-14). For rules that have missed their effect, we check whether the *gap* between the expected and the actual effect of applied adaptation rules is smaller than the tolerance value ε_r. If this *gap* is greater than the maximal deviation ε_r, the corresponding rules are disabled from the rule set in K_{Adapt} to ensure correctness of planning results (line 16). In the end, all history objects with a status change are moved into a separate evaluation data base that is used for the classification of rule effects, which is described in the following subsection and which builds the basis for our *Observation-Based Learning*.

The size of this data base is restricted and the oldest entries are moved into our explanation archive following the FIFO principle. With that, we restrict the amount of data, and thus, the computation time, of our classification. Furthermore, our restriction to the latest observations also supports the exploration of recent changes in the environment. As more data may lead to more accurate results, the size of the data base has to be chosen with care and should be tailored to the application and the specific dynamics of the operational environment.

Example As illustrating example, we assume that the effect of flow temperature adjustments has changed after the installation of a new radiator. In this case, the assumption that a change of $2°C$ in the flow temperature leads to a change of $1°C$ in the room temperature is not valid anymore. Instead, a constant shift of the effect can

be observed. In the following, we evaluate the accuracy of the rule *increase n* after the radiator exchange.

- execution state:
  $\sigma_{\text{execution}} = \{\text{time} = 17, \text{temp\_out} = 15, \text{temp\_in} = 18,$
  $\text{airCon\_mode} = 0, \text{energy} = 6\}$
- adaptation rule (shortened):
  $\text{guard}_{\text{increase\_n}}$ **&** $n := n + \text{stepsize}_{\text{n}}$
  $\longrightarrow (\text{temp\_in}_{\text{env}} \geq \text{temp\_in} + \text{stepsize}_{\text{n}}/2$
  $\wedge \ \text{energy}_{\text{env}} = \text{energy} + \text{stepsize}_{\text{n}})$
- expected effect:
  $\sigma = \{\text{time} = 17, \text{temp\_out} = 15, \text{temp\_in} = 19, \text{airCon\_mode} = 0,$
  $\text{energy} = 8\}$
- observed effect:
  $\sigma_{\text{effect}} = \{\text{time} = 17, \text{temp\_out} = 15, \text{temp\_in} = 20, \text{airCon\_mode} = 0,$
  $\text{energy} = 8\}$
- calculated deviation: $\{(\text{temp\_in}, 1), (\text{energy}, 0)\}$
- rule status: `effect_missed`

The deviation of $1°C$ is less than the tolerance value, thus the rule is kept in the rule set and the history object is moved into the evaluation data base for further processing within our *Observation-Based Learning*.

With this knowledge, we can learn the new relationship between flow temperature and room temperature (assuming a temporarily constant outdoor temperature) with our observation based learning.

7.1.2 Deviation Classification

The main idea of our classification algorithm is to divide all observed rule executions from the evaluation data base into equivalence classes that are calculated based on the observed deviation from the expected effect. To this end, we require the designer to provide a fuzziness value λ_p (for each effected parameter p) that specifies the deviation from the expected effect that is still considered as equivalent. Based on the classification result, we can have two cases: a) all executions have a uniform deviation (are in the same equivalence class), b) deviations are non-uniform (more than one equivalence class). In case a), we have observed an inaccuracy in the effect quantification and in case b), the effect may depend on the execution context or on some incidents that happened between execution and evaluation. For both cases, we propose to refine the adaptation rules to more specific rules that provide adjusted effect expectations for the observed contexts in our *Observation-Based Learning* as described in the next section.

In the following, we explain our equivalence class construction. Based on the fuzziness value λ_p, we define that a rule is accurate if the observed deviation from its expected effect is smaller than λ_p for each effected parameter p, i.e. $dev_p < \lambda_p$. Thus, we first classify the deviation for each effected parameter. Our parameter equivalence classes c_p have the following form for $\lambda_p > 0$:

$$..,[-5\lambda_p,-3\lambda_p),[-3\lambda_p,-\lambda_p),[-\lambda_p,\lambda_p],(\lambda_p,3\lambda_p],(3\lambda_p,5\lambda_p],...$$

This can be formally defined as:

$E := \{[-\lambda_p, \lambda_p]\} \cup E_{neg} \cup E_{pos}$

with $E_{neg} := \{[-(2k+3)\cdot\lambda_p, -(2k+1)\cdot\lambda_p) \mid k \in \mathbb{N}_0\}$

and $E_{pos} := \{\ ((2k+3)\cdot\lambda_p, (2k+1)\cdot\lambda_p] \mid k \in \mathbb{N}_0\}$

For $\lambda_p = 0$, each deviation forms a new equivalence class as no deviations can be considered equivalent.

Example For our example, we assume $\lambda_{temp\_in} = \lambda_{energy} = 0.5$. Thus, the equivalence classes for our rule execution of *increase n* with deviation {(temp_in, 1), (energy, 0)} are $(0.5, 1.5]$ and $[-0.5, 0.5]$.

For our classification, we use class identifier (IDs) to refer to these equivalence classes. As class IDs, we use the distance to the equivalence class of an accurate rule with respect to parameter p. Thus, for accurate effect expectations (i.e., deviation $dev_p \in [-\lambda_p, \lambda_p]$), we get classID$_p(dev_p, \lambda_p) = 0$, for $dev_p \in [-3\lambda_p, -\lambda_p)$, we get classID$_p(dev_p, \lambda_p) = -1$, and classID$_p(dev_p, \lambda_p) = 1$ for $dev_p \in (\lambda_p, 3\lambda_p]$, etc. The classID of the class for a given deviation dev_p can be formally defined as follows:

$$\text{classID}_p(dev_p, \lambda_p) = \begin{cases} 0 & \text{if } dev_p \in [-\lambda_p, \lambda_p] \\ -(k+1) & \text{if } dev_p \in [-(2k+3)\cdot\lambda_p, -(2k+1)\cdot\lambda_p) \\ k+1 & \text{if } dev_p \in\ ((2k+3)\cdot\lambda_p, (2k+1)\cdot\lambda_p] \end{cases}$$

We can calculate this identifier for a given deviation dev_p with the following function:

$$f_{\text{classID}_p}(dev_p, \lambda_p) = \begin{cases} dev_p & \text{if } \lambda_p = 0 \text{ or } dev_p = 0 \\ c\_id(dev_p, \lambda_p) & \text{if } dev_p > 0 \\ c\_id(|dev_p|, \lambda_p)\cdot(-1) & \text{if } dev_p < 0 \end{cases}$$

$$c\_id(dev_p, \lambda_p) = \begin{cases} ((dev_p/\lambda_p) - 1)\%2) & \text{if } dev_p\%\lambda_p = 0 \\ ((dev_p/\lambda_p) - 1)\%2 + 1 & \text{otherwise} \end{cases}$$

Within our classification algorithm, we classify the overall accuracy of an effect expectation for each rule execution that is stored in our evaluation data base (unless it is already classified). To this end, we first calculate the classIDs for each affected parameter of this rule observation. Then, we combine these classes into an overall equivalence class c_{rule}, which we represent as vector of parameter classIDs in the order in which the parameters occur in the effect formula: classID$_{rule} = \langle$classID$_{p1}, ...,$ classID$_{pn}\rangle$. As we only compare executions of the same rule with each other, this order is fixed. The calculated classID$_{rule}$ is stored in the adaptation history object of the corresponding rule

execution. The classified set of rule effect observations builds the basis for our subsequent *Observation-Based Learning*.

Example For our example, we get

$$\text{classID}_{\text{increase n}} = \langle \text{classID}_{temp\_in}, \text{classID}_{energy} \rangle = \langle 1, 0 \rangle.$$

In the subsequent learning step, we check whether this deviation can be observed several times to avoid adjustments for outliers. Due to the change of the radiator, we can observe a permanent shift. Thus, the rule is adjusted as described in the next section.

With our *Rule Accuracy Evaluation*, we provide a systematic evaluation of observed deviations from the encoded expectations that also considers the timing behavior of adaptation effects. The evaluation results are used to adjust existing adaptation rules within our *Observation-Based Learning* as described in the next section.

7.2 Observation-Based Learning

With our accuracy evaluation, we classify executions of adaptation rules based on whether they achieve the expected effect in time or within an acceptable time span, and if not, to which degree the effect differs. Based on this, we can detect systematic deviations from the expected effect and adjust our adaptation rules. To this end, we rely on our effect classification to identify equivalent deviations and context conditions that are characteristic for each observed equivalence class. The main idea of our rule adjustment is to split the original adaptation rule into more specific rules that fit the observed deviation classes and specify the context in which these effects are expected.

Example In our smart temperature example, a correction may be necessary for an adaptation rule that describes the effect of adjusting heating parameters on the indoor temperature. In our adaptation rules, we have assumed that this effect is independent from the outdoor temperature. If for example, the effect is different for outdoor temperatures greater than $15°C$, we will observe a different deviation in this situations.

7.2.1 Learning Process

The main steps of our *Observation-Based Learning* are depicted in Algorithm 7.2. For each evaluated rule execution in our evaluation data base that did not achieve its effect in time (i.e. with status = effect_missed), we first check whether we have sufficient observations for this rule to allow for generalization of observations, i.e. the amount of observed executions within the evaluation horizon is larger than a rule specific threshold γ. If this is the case, we proceed with our correction of rule effect expectations as given in Algorithm 7.3. Otherwise, this rule will be further observed as long as it is safe (the deviations are within the safety threshold ε_r). Note that the necessary amount of observations γ is application dependent and has to be chosen with care, as more data

```
1  input: evaluation data base
2  output: NEW_RULES
3  begin
4    NEW_RULES ← ∅;
5    foreach r ∈ evaluated rules with r.status != effect_achieved
6      if(count(r, E) > γ)
7        NEW_RULES ← NEW_RULES ∪ ruleEffectCorrection(r, E);
8      end_if
9    end_for
10   generalisationAndVerification(NEW_RULES);
11 end
```

Algorithm 7.2: Observation-Based Learning

may increase the accuracy but lead to more inaccurate adaptations of the system before adjusting the rule set. The latter decreases the quality of the adaptation process but due to the definition of the maximal tolerance ε_r, we ensure that a severe deviation only occurs once. After constructing the set of specific rules, we perform an iterative rule generalization and verification as described in Section 7.4. With that, we aim at reducing the total amount of a) adaptation rules to speed-up planning, and b) rule learning steps that are triggered by observing deviating effects in new context conditions that can be covered by already learned rules. In the verification step, we verify the newly learned set of adaptation rules and only insert rules into the rule base K_{Adapt} that have been verified successfully. Our verification is described in Chapter 8. Note that although we rely on runtime model simulation for our generalization and verification, we still need less simulation steps for observation-based learning than for simulation-based learning, where simulation is also used for creating initial adaptation rules.

7.2.2 Correction of Rule Effect Expectations

To adjust our rule effect expectations, we identify context-dependent deviations by analyzing the execution context to extract context conditions that can be used to refine the guard of the adaptation rule. We propose to split the original adaptation rule into context-specific rules. To this end, we refine the guard of the original rule based on extracted context-conditions. For each specific rule, we adjust the effect expectation by adding a calculated offset (which could be negative as well). For the identification of context conditions, we rely on the relation $R \subseteq (SP \cup EP) \times EP$ that specifies which control or environment parameters may possibly influence which environment parameters. We assume this relationship to be specified at design time and to be updated at runtime if new components provide new sensors or actuators. The intent of this relation is to use expert knowledge to reduce false implications from spurious correlations. In the following, we describe our `ruleEffectCorrection` algorithm, which is given in Algorithm 7.3.

Preparation In a first step of our `ruleEffectCorrection` algorithm we filter the identified equivalence classes of our *Rule Accuracy Evaluation* (Line 6) to get a set

```
1  input: rule r, evaluation data base E
2  output: NEW_RULES
3  begin
4    CLASSES ← getClasses(E);
5    //filter significant deviations
6    CLASSES ← filter(CLASSES, λ class.elemCount(class) > γ);
7    NEW_RULES, COND ← ∅;
8    COND ← extractConditions(CLASSES);
9    //create specific rules
10   foreach class_j ∈ CLASSES \ {class_0}
11     effect_j ← addOffset(class_j, r);
12     NEW_RULES ← NEW_RULES ∪ createRule(class_j,r,COND,effect_j,0);
13   end_for
14   NEW_RULES ← NEW_RULES ∪ createRule(class_0,r,COND,r.effect,0);
15   return NEW_RULES;
16 end
```

Algorithm 7.3: ruleEffectCorrection

of significant evaluation results, i.e. the amount of observed executions within each remaining equivalence class is larger than the rule specific threshold γ. Thus, for rules with non-uniform deviations, i.e. more than one identified equivalence class, we only consider those classes that contain enough observations. Note that λ in Line 6 of our algorithm is a λ-term that specifies an anonymous function, and should not be confused with λ_p. The main idea of our correction is to split the rule into more specific rules. By only using significant evaluation results, we avoid creating rules for random effect deviations or for very specific and rare context conditions. Disregarded deviation classes will still be considered later if enough observations are collected that fall into this class.

Extraction of Context Conditions In a next step, we try to identify context conditions that are characteristic for each rule equivalence class (Line 8). These conditions are conjunctions of value ranges for system and environment parameters and are inferred by comparing the execution contexts of adaptation rules within the same class and between classes within `extractConditions`. To identify responsible context parameters, we rely on classification rule learning algorithms such as RIPPER. This rule learning algorithm infers rules of the form *IF condition THEN class*. As classes, we use our calculated equivalence classes. The condition contains all parameters and value ranges for which the class could be observed in the data. These ranges get more precise if more data is observed. However, even for small datasets, we get good results if the data does not contain too much noise. The extracted conditions are used to create rules with guards that describe specific context conditions (Lines 12 and 14) because the observed behavior might be context-specific and should not be generalized beyond available runtime observations. If, for example, an autonomous robot vehicle is driving on a sandy surface it might observe that the covered distance is smaller than expected. However, this effect is only valid for sandy surfaces and will be different on e.g. asphalt. Thus, even if all

```
1  input: equivalence class c_i, rule r
2  //set of tuples (parameter name p, offset o_p)
3  output: formula effect_i
4  begin
5    set<(p,o_p)> OFFSET ← ∅;
6    foreach p ∈ effectedParameters(r)
7      DEVIATIONS_p ← filter(r.DEVIATIONS, hasParamName(p));
8      o_p ← avg(DEVIATIONS_p)
9      OFFSET ← OFFSET ∪ (p,o_p);
10   end_for
11   //adds each o_p to its respective subformula effect_p
12   //and returns the modified effect formula
13   effect_i ← addOffsetToEffect(r, OFFSET);
14   return effect_i;
15 end
```

Algorithm 7.4: addOffset

observed deviations are uniform, we split our rule into a rule that captures observed contexts and deviations and the original rule that is refined to the unobserved contexts.

Example For our example, where the effect is different for outdoor temperatures greater than $15°C$, we observe the following deviations for our rule increase_n: a deviation of $1°C$ for eight rule executions with $\text{temp\_out} = 15$ and a deviation of $0°C$ for four executions with $\text{temp\_out} = 6$.

Effect Correction For inaccurate rules (Line 10), we adjust the expected effect to the observed effect in these contexts by adding the average observed deviation (which could be negative as well) of all rule executions within this specific equivalence class to the expected effect (Line 11). This is done for each affected parameter and parameter-specific effect subformula formula_p within our `addOffset` algorithm as depicted in Algorithm 7.4. We have decided to take the average instead of a class-specific representative number to be even closer to the actual behavior.

Example For our example, we add the constant shift of $1°C$ to the subformula for the indoor temperature: $\texttt{effect}_{\text{temp\_in}} = \texttt{effect}_{\text{temp\_in}} + 1$.

Creation of new Adaptation Rules The extracted context conditions and the class-specific effect formulas are now used to create a set of specific rules that follow the following construction scheme:

$r_{i1} : (g_i \wedge\ cond_1 \wedge\ \neg(\bigvee cond \in \text{COND} \setminus cond_1))$ **&** $c_1; c_2; \ldots; c_n$
$\longrightarrow (\textit{effect} \oplus \textit{OFFSET}_1)$ **after** $time$

$r_{i2} : (g_i \wedge\ cond_2 \wedge\ \neg(\bigvee cond \in \text{COND} \setminus cond_2))$ **&** $c_1; c_2; \ldots; c_n$
$\longrightarrow (\textit{effect} \oplus \textit{OFFSET}_2)$ **after** $time$

...

$r_i : (g_i \wedge \neg(\bigvee cond \in \text{COND}))$ **&** $c_1; c_2; \ldots; c_n \longrightarrow \textit{effect}$ **after** $time$

```
1  input: classID class_j, rule r, context conditions COND,
2     formula effect_j, timing offset t
3  output: rule specific_r
4  begin
5    if (class_j != 0)
6      cond_j ← getCond(c_j, COND);
7      r.guard ← r.guard ∧ cond_j ∧ ¬(⋁ cond ∈ COND \ cond_j);
8      r.effect ← effect_j
9    else
10     r.guard ← r.guard ∧ ¬(⋁ cond ∈ COND);
11   end_if
12   return r;
13 end
```

Algorithm 7.5: createRule

with COND and OFFSET from Algorithm 7.3 and 7.4, respectively. The corresponding algorithm is given in Algorithm 7.5. Lines 4 - 7 construct rules with adjusted effect, whereas Line 9 refines the guard of the original rule such that it fits all other, possibly non-observed contexts.

Example For our example, we split our rule into one for temp_out $= 15$ with the corrected effect, and two rules without effect correction: one for temp_out $= 6$ and one for temp_out $\neq 6 \wedge$ temp_out $\neq 15$. Over time, when more conditions for both classes are observed, we create more rules and rely on rule generalization to merge similar rules as described in Section 7.4.

In summary, the *Rule Accuracy Evaluation* detects inaccurate adaptation rules and classifies the observed effects of safe rule executions (gap $\leq \varepsilon_r$) w.r.t. the observed deviation from the expected effect. The evaluation results are used within our *Observation-Based Learning* to refine deviating adaptation rules into more specific context-dependent rules that capture the observed effects. This enables the accurate co-evolution of the adaptation rules. In the next section, we describe our *Simulation-Based Learning* that is used to learn new rules in the case that no fitting rule exists (e.g. because of situations that are not captured by the existing rules, or due to rule disabling caused by severe deviations form the expected effect) in detail. First, we introduce the runtime models that we use for learning and verification. Subsequently, we describe our learning algorithm in detail. Afterwards, we describe our rule generalization approach that enables reuse of learning results in similar contexts, and thus reduces the learning effort. To enable the traceability of autonomous adaptation and evolution decisions for comprehensibility and explainability, we store relevant information in an explanation archive as described in Section 7.5. Note that the runtime evolution of the adaptation logic has to be done with care to avoid compromising the stability of adaptations, and thus, of the system. In Section 7.6, we discuss how the stability of meta-layer adaptations (adaptations of the adaptation logic) can be addressed.

7.3 Simulation-Based Learning on Runtime Models

With our *Observation-Based Learning*, we are able to correct the effect expectations of existing adaptation rules. To cope with situations that are not covered by the existing rules, e.g. due to unexpected behavior of the environment, or due to changes in the system goals or in the system topology, we propose a *Simulation-Based Learning* on executable runtime models of the system and its environment. The main idea is to apply changes to the system runtime models and to observe their effect on the simulated environment, which is considered a black box. Based on these simulations, we extract an initial adaptation rule and generalize this rule to increase the applicability of newly learned rules. Each generalization step is validated with model simulation. In the end, we perform a comprehensive system verification to verify that adding the new rule does not compromise any important system properties. In the following, we present our *Simulation-Based Learning* approach in detail. A previous version has been presented in [KGG18b].

Example To illustrate the *Simulation-Based Learning* of adaptation rules within our running case study of the smart temperature control system, we consider the following scenarios:

1) We assume that the initial control system is a smart heating system that only controls the heating unit. Due to climate change, summers are getting hotter and air conditioning is necessary to achieve acceptable room temperatures in summer. After having installed such an air conditioning component, it has to be integrated into the smart temperature control system.

2) The old heating unit is replaced by a floor heating.

To integrate the new components into our self-adaptive system, we rely on our topology-aware monitoring and assume that the air conditioning and the floor heating provide runtime models that can be used to learn the effect of tuning its control parameters with regard to the room temperature and the overall energy consumption. We integrate these models into our simulation environment (assuming that the necessary interfaces (e.g., channels and ports) already exist or are added by some kind of installation routine) and apply our learning process.

7.3.1 Learning Process

In our simulation-based learning process (as depicted in Figure 7.1), we first update all runtime models (RTM) with the current system parameters K_{Sys} and environment parameters K_{Env}. This update includes at least the parameter initialization for the current situation. To improve the predictions of model simulation, we propose to use the previously observed system and environment behavior (recorded in the histories) to update the relation between system actions and environment states in the RTMs, e.g., by adjusting probabilities of transitions, or by changing transitions in the models. This

update mechanism is RTM specific and, thus, is assumed to be managed by the update interface of the runtime models. In Section 7.3.2, we present two exemplary formalisms for RTMs and briefly discuss how the update can be realized for the formalisms.

After updating the RTMs, we apply a learning algorithm on the runtime models to find an adaptation that leads to an improvement of the overall distance to the goals (cf. Chapter 6 for distance calculation), when applied in the current situation. The learning algorithm uses atomic operations, which are specific to each runtime model, to mutate system control parameters in an appropriate way. We base the evaluation of newly learned model parameters w.r.t. the system goals on our distance function. To achieve stability of adaptations, we require the quality evaluation to include stability aspects, e.g., by evaluating the distance of several future states, and by only choosing states that are stable for some specified time. The learning algorithm can be freely chosen as long as it operates on simulation results of RTMs. In this thesis, we exemplarily use a genetic algorithm (GA) for fine-granular parameter adaptation (as e.g., for the heating curve parameters in our running example) and a "scenario-based mode-evaluation" on characteristic environment scenarios for coarse-grained mode adaptation. We have chosen to use a GA for parameter adaptation because it is easy to implement and achieves good results for a wide range of problems. The application of GAs for self-adaptive systems in general was also recommended by Coker et al. [CGLG15]. They argue that stochastic search-based algorithms like GAs are well-suited to handle multi-dimensional search spaces and complex optimization problems, which are often present in self-adaptive systems. For mode adaptation, the search-space is small and all available modes can be evaluated in different scenarios to learn the best mode for characteristic scenarios. We describe our genetic algorithm and its interaction with the RTMs in detail in Section 7.3.3 and our *scenario-based mode-evaluation* in Section 7.3.4. As the runtime models may may focus on different aspects (e.g. communication, energy consumption, ...), and represent different components (e.g. heating, air conditioning, ...), not all models have to be mutated. The relevant models are chosen according to heuristics w.r.t. the violated subgoals. These heuristics are a parameter of the learning algorithm.

If learning was successful, i.e., a configuration was learned that achieves an improvement, we use the sequence of mutations to generate an initial adaptation rule that consists of a specific guard describing the current situation and parts of the history that led to this situation, commands capturing the necessary changes to the control data of the system, and the effect of applying these changes in the current situation. The latter can be extracted from a model simulation after learning as described in Section 7.3.5. This also provides us with the timing information when the effect is assumed to be observable. After constructing the initial rule, we perform an iterative rule generalization and verification as described in Section 7.4. With that, we aim at reducing the total amount of a) adaptation rules to speed-up planning, and b) rule learning steps that are triggered by observing new context conditions that can be covered by already learned

rules.We only add the generalized rule if verification was successful. If no suitable parameters could be learned or if the final system verification fails, the learning/verification component terminates and waits until it is triggered again by subsequent executions of the evaluation component in future monitoring cycles. In this case, we rely on the existence of a suitable safe operation mode in which the system can be set.

In the following, we first describe our requirements on runtime models and two exemplary modeling formalisms, which we use in this thesis to illustrate and evaluate our framework. Then, we present our genetic algorithm and its interaction with the RTMs in detail. Afterwards, we describe how we construct an initial adaptation rule from the learning result. We present our subsequent rule generalization that is used for observation- and simulation-based learning in Section 7.4.

7.3.2 Runtime Models for Learning and Verification

In our framework, we apply online learning on runtime models (RTMs) to learn new adaptation rules. These models capture the system and environment behavior in an executable model. They have to a) capture the interplay between the modeled system component and the environment, b) be executable in a way that allows for a prediction of future system and environment behavior, and c) provide an interface for instantiation of environment and system state and integration of observed environment behavior based on current and history data. The latter allows us to consider the environment as black box during learning, where we apply changes to the system models and observe their effect on the environment state. Furthermore, we require our runtime models to have a formal semantics, which enables us to employ them also for verification purposes. The runtime models may focus on different aspects of the actual system implementation, which enables a separate analysis of, e.g., timing or energy consumption.

Example In our running example, we have modeled the influence of our temperature control system on the indoor temperature and on the energy consumption in separate processes in our runtime model of the environment.

The runtime update mechanism for RTMs allows for (i) instantiating the RTM for a specific situation for which a rule should be learned and (ii) for integrating observed behavior to enable co-evolution of RTMs and actual environment. For the first purpose, the RTM should provide some parameters that can directly be set to situation specific values or value vectors, or that allow for switching between predefined context modes. For the second purpose, mode switches provide a solution for environments that follow predefined modes or for systems that show a finite amount of actuator failures, which can be represented by different models as done for continuous self-modeling of a four-legged robot in [BZL06]. For unguided runtime evolution of RTMs, architectural parameters like delays between actions (as e.g. done within [MNBB17]) or probabilities of showing different behavior (as e.g. done in [EGMT09, MJJ+17]), or even complete models (as done in (timed) automata learning, e.g. [dMPCdS12, CdL12, MNBB17, TALL18]) could

be inferred from the observed behavior. Behavioral parameters, like e.g. underlying functions, could also be constructed based on the observed history of parameter values.

Example In our running case study, we have, for example, encoded the possibility to switch between scenarios that consist of representative temperature and sun intensity trends for different seasons. Based on the observed outdoor temperature and sun intensity values, it is possible to identify the best fitting scenario or to add an observed scenario. Moreover, the ideal heating curve or the effect of the air conditioning are encoded as (linear) parameterized functions and, thus, can be easily adjusted in the environment model.

In this thesis, we use two exemplary modeling mechanism: UPPAAL timed automata (UTA) and the system description language SystemC. Both enable the modular design (i.e. with UTA templates or SystemC modules) of different communicating components on different levels of abstraction. Thus, we can capture the interplay between system components and environment. Both provide simulation and verification capabilities and can be parameterized to provide an interface for runtime updates. Thus, both are well suited as runtime models in our framework. The main reason for using both modeling mechanisms is that UTAs provide us with build-in model checking possibilities, whereas SystemC models enable fast and efficient simulation, which is beneficial for simulation-based learning. However, other formalisms like e.g. domain specific languages (e.g. Simulink) or formal languages (e.g. CSP [Sch99]) are suitable as well as long as they are executable (for learning) and have a formal semantics (for verification).

Correct runtime model creation is a crucial task for our simulation-based learning. However, formal modeling is hard and error prone, and requires expert knowledge. To support this process and, thus, to increase the acceptance of our framework, we propose to reuse models from mode-based design, and/or to automatically extract RTMs from a system level implementation during the design process. In this thesis, we use an existing SystemC to Timed Automata Transformation Engine (STATE) [HFG08, HPG15] to extract UPPAAL Timed Automata (UTA) models from a SystemC system-level implementation to enable formal verification on SystemC runtime models. With that, we can define a verification strategy for UPPAAL timed automata and reuse it for SystemC RTMs.

7.3.3 Rule Learning for Parameter Adaptation based on a Genetic Algorithm

Finding the best parameters for a given situation is a multi-dimensional optimization problem. An efficient approach to solve such problems are stochastic search algorithms, like genetic algorithms (GAs). GAs are easy to implement and achieve good results for a broad range of problems. Thus, we instantiate the learning algorithm in our framework with a basic genetic algorithm that operates on parameters of our runtime models. The process is depicted in Figure 7.2. It starts with a parent population of vectors of control

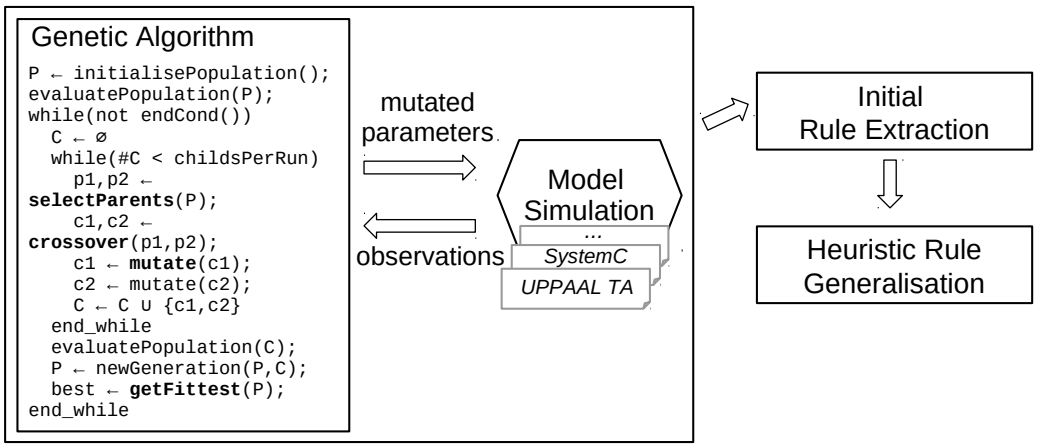


Figure 7.2: Rule Learning using a Genetic Algorithm

parameter values (i.e. (name, value) tuples). Usually, this parent population is initialized with random solutions. However, we only want to learn solutions that can be reached by applying a series of valid mutation operations in the current situation. Thus, we modify the random initialization such that each parameter of a new individual can be reached from the current parameter by adding a random multiple (which could be negative as well) of the valid step size for this parameter. For each iteration, the GA generates the specified amount of children by selecting two random parent vectors from the current generation, combining their parameter values using single-point crossover and mutating some control parameters of the resulting child vector. Single-point crossover is the most common crossover strategy for genetic algorithms and has delivered good results for our case studies. Mutation of parameters is performed according to a probabilistic mutation rate and corresponds to applying a valid mutation operation on the selected parameter. To decide which system parameters should changed during learning, we rely on the expertise of the designer. In future work, this decision can be based on a heuristic that considers history information and currently violated system goals.

To compute the fitness of new individuals, we use a combined simulation of the mutated system and an environment model that is instantiated with the actual observed environment behavior. The simulation starts at some time in the past with the old parameters to enable the simulation to reach the current situation, or is directly set to a state that corresponds to the current situation. Then, the mutations are applied by an abstract adaptation process that we add to the system model, and the simulation proceeds for some specified time (based on predictions of the future environment behavior) to observe the effect. During simulation, changes to monitored parameters are recorded in some RTM-specific trace file to observe the effect of mutations. The resulting data set is fed into a fitness function to calculate the fitness of the mutation series. To this end, we use a provided fitness function or extract a fitness function from the distance function of our goal model. The fitness function should consider additional aspects such as stability

of the adaptation effect. This can be achieved by applying the fitness function not only on the state where the effect could be observed first, but on several well chosen samples from the simulation (e.g., using the average fitness of three states that are observable after the effect time).

Example To illustrate the learning component, we consider an example scenario for our illustrating case study of an adaptive temperature controller. For presentation purposes, we assume that the knowledge base currently has no adaptation rules at all. Furthermore, we initialize the system with an assumed heating curve with $(m = 1, n = 30)$ (while the ideal one is $(m = -1.2, n = 44)$. For the environment, we consider a spring environment scenario (outdoor temperature values between 5 and $20°C$) in which the current parameters lead to a situation where the indoor temperature reaches $24°C$ at 11 a.m. This situation is depicted in Figure 7.4 left to the vertical line. At time point $40200s$, the MAPE-K layer recognizes the deviation of the indoor temperature to the reference temperature of $20°C$. As no adaptation is applicable, the learning component is invoked to learn the parameters m and n with which the correct flow temperature can be calculated such that the reference temperature can be achieved. In this example, our learning algorithm returns with the "correct" parameters $m = -1.2$ and $n = 44$.
To evaluate the fitness within our genetic algorithm, we use the distance function of our goal model:

$$dist = 0.7 \cdot dist_{temp} + 0.3 \cdot dist_{energy} \text{ with}$$

$$dist_{temp} = (0.5 \cdot (\sigma(temp\_in) - 20)^2) \cdot 100/200 \text{ and}$$

$$dist_{energy} = e^{(\sigma(energy) - min_v) \cdot ln(100)/(max_v - min_v)} \cdot 100/52 = e^{(\sigma(energy)) \cdot ln(100)/(52))} \cdot 100/52.$$

To achieve a stable result, we apply this function on each value change of room temperature and energy within the simulation interval and used the weighted mean (w.r.t. the timespan until the next value change) of these fitness values. As the GA tries to maximize the fitness, and our distance-based planning aims to minimize the distance, we multiply the distance with -1. Thus, the resulting fitness function is

$$f = (\sum(-1) \cdot dist(sample) \cdot duration(sample))/\#duration.$$

Choosing an appropriate fitness function is one of the key aspects of applying the GA. By using the distance functions that are encoded in our goal model, we can be sure that the effect of adaptations is evaluated according to the designers intents. However, as we are not interested in the absolute value of the fitness during learning, but rather in the trend, i.e. whether the distance is improved or not, often a simple fitness function is sufficient.

The fitness of all children w.r.t. the system goals is evaluated to decide which parameter vectors are kept for the next iteration. As an optimization, we do not simulate all children, but compare their parameter values with individuals from the parent population. If an individual with the same parameter values exists, we copy the

fitness value from the parent to the child. We use the elitist selection to select the best individuals to build the next generation. For our prototype,we have chosen single-point crossover and elitist selection as these are basic crossover and selection algorithms that give good results on many optimization problems. These functions can be easily replaced by more problem specific solutions. Thus, this decision is no limitation for our framework.

The learning algorithm stops if a) a target fitness is reached or b) if it has performed a specified maximum amount of iterations. The target fitness, the maximum number of rounds and the population size are parameters of the algorithm, which can be chosen by the designer or calculated at runtime w.r.t. necessary precision, available resources and time criticality of learning. As a result, we get a series of adaptation commands and the best parameter values for the control parameters that are found with this heuristic optimization algorithm. After learning, a rule can be extracted and generalized as described in Section 7.4.

7.3.4 Rule Learning for Mode Adaptation based on Scenario-Based Mode-Evaluation

In mode adaptation, the adaptation layer chooses the best system mode, i.e. a predefined configuration for a specific behavior, for the current situation. There exist only a small amount of different modes between which the system can change. Even, if there are several independent modes for different functionalities, the amount of valid mode combinations that are relevant for a certain system goal usually is small enough to allow for a comprehensive evaluation of all these modes in a concrete scenario. Thus, we do not use a GA for mode-based adaptation. Instead, we differentiate between two cases: simulation-based mode-evaluation for a concrete situation, e.g. because of newly detected environment behavior, and scenario-based mode-evaluation for several characteristic environment scenarios, e.g. to learn adaptation-rules for a new system component.

To identify the best mode for one concrete situation, we perform a simulation of this situation for each mode. To this end, we instantiate the system with a mode and simulate the interplay between system and environment within this situation. We evaluate the fitness of each mode by using the same *simulation-based* evaluation as for the fitness evaluation within our genetic algorithm. By comparing the fitness of all mode combinations, we can select the best one for the current situation. The result of this simulation-based mode-evaluation is used to extract an initial adaptation rule as described in Section 7.4.

The main idea of our *scenario-based* evaluation is to simulate several scenarios for each mode to learn adaptation rules for different situations in advance. If the environment shows some recurring characteristic behaviors, like e.g. characteristic temperature trends in different seasons and in the course of a day, we can use this knowledge to select the most valuable scenarios for our evaluation. To achieve this, we identify characteristic

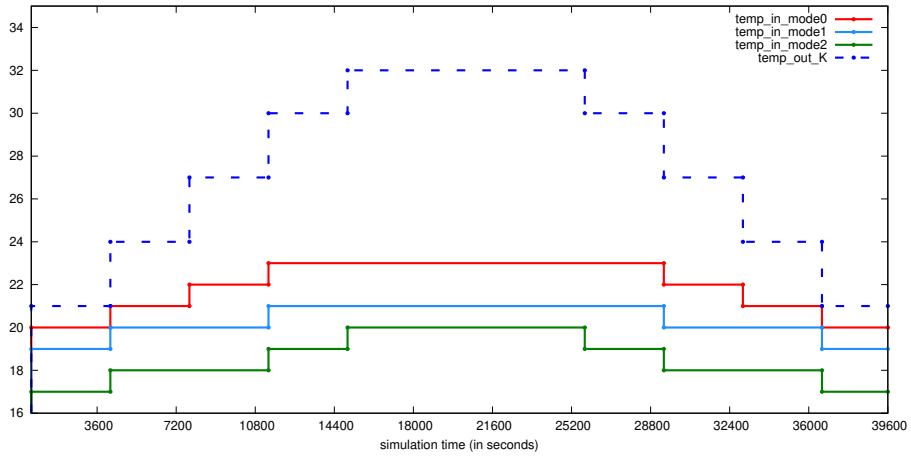


Figure 7.3: Resulting Indoor temperatures for different modes of the air conditioning

scenarios that describe a typical sequence of environment values over time (e.g., typical temperature trends in spring, summer or hot summer). Then, we simulate each scenario (or selected scenarios that are likely to occur in near future) with each mode combination and compute the fitness value for each point in time. By comparing the result of all simulations, we can identify the best mode for each time span and extract adaptation rules for each mode switch. To extract adaptation rules that describe the effect of switching from any non-optimal mode to the optimal mode at a given point of time, we iterate through all simulation traces and construct one rule for each non-optimal mode as described in Section 7.4. With this approach, we simulate each scenario for each mode only once, and identify all points of interest within these traces. The result is a comprehensive set of initial adaptation rules for the evaluated scenarios. We further generalize these adaptation rules to achieve a broader applicability, e.g. for similar scenarios.

Example To learn suitable adaptation rules for the newly installed air conditioning, we evaluate the different modes of the air conditioning (off, on or power) in one suitable scenario that contains all relevant outdoor temperature values (summer time: outdoor temperatures that start at around $18°C$ in the night and increase up to $32°C$ in the early afternoon) with our scenario-based evaluation. The result is shown in Figure 7.3. The simulation starts at 10 a.m. and ends at 8 p.m. In our scenario, the difference between the outdoor temperature in the night or in the early afternoon is quite high. Thus, we have to adapt the mode several times during such a typical summer day to achieve best results. In our simulation, the best results can be achieved with mode off for outdoor temperatures below $24°C$, with mode on for outdoor temperatures between $24°C$ and $30°C$, and mode power for outdoor temperatures above $30°C$.

In the following, we first describe our generation of an initial adaptation rule from the simulation of the learning result of our genetic algorithm. Afterwards, we show how our GA can be applied on both RTM languages.

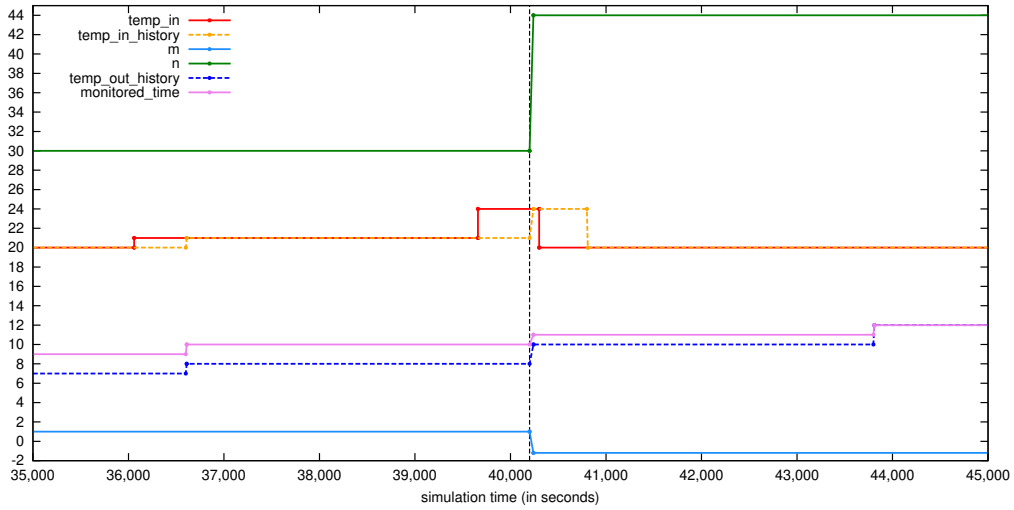


Figure 7.4: Effect of Learned Adaptation

7.3.5 Initial Rule Generation

If learning was successful, i.e., a configuration was learned that achieves an improvement, we use the sequence of applied mutations and a simulation of the mutated RTM to generate an initial adaptation rule as follows. First, we add an abstract monitoring process that periodically stores system and environment data to mimic the history information of the knowledge base to the mutated RTM. With this history information we are able to formulate history-aware guards. This data is also traced in the respective trace file. Then, we simulate the resulting RTM, parse the trace file and extract the relevant values for the guard and the effect of the new adaptation rule. Here, we differentiate between monitored values that are stored in the history (indicated by index K) and current environment values. History values are used for guard extraction, while environment values are used for effect extraction. In the following, we describe these extraction steps in detail.

Guard Extraction

The guard describes the situation, i.e. the parameter values for system and environment parameters as captured in K_{Sys} and K_{Env}, at the start of the adaptation execution and parts of the history of the same parameters that lead to this situation. The amount of considered history data and the selection of potentially relevant parameters is application specific and can be specified in the rule generation algorithm. In a subsequent rule generalization step, we, furthermore, perform an iterative reduction and relaxation of guard conditions.

Example We illustrate the rule extraction with our example from Section 7.3.3. We initialize the system with an assumed heating curve with $(m = 1, n = 30)$ (while the ideal one is $(m = -1.2, n = 44)$. For the environment, we consider a spring environment scenario (outdoor temperature values between 5 and $20°C$) in which the current parameters lead to a situation where the indoor temperature reaches $24°C$ at

11 a.m. This situation is depicted in Figure 7.4 left to the vertical line. At time point $40200s$, the MAPE-K layer recognizes the deviation of the indoor temperature to the reference temperature of $20°C$. As no adaptation is applicable, the learning component is invoked to learn the parameters m and n with which the correct flow temperature can be calculated such that the reference temperature can be achieved. In this example, our learning algorithm returns with the "correct" parameters $m = -1.2$ and $n = 44$. Note that we have omitted the effect on the energy to concentrate on the temperature.

To extract an initial guard we not only include the temperature values for the current situation, but also the last values of the history. Here, we use t to refer to the current monitoring cycle and $t-1$ to refer to the last cycle, etc. Thus, we get the following guard.

$$(temp\_in_{K(t-3)} = 20 \wedge temp\_out_{K(t-3)} = 7 \wedge time_{K(t-3)} = 9) \wedge (temp\_in_{K(t-2)} = 20 \wedge temp\_out_{K(t-2)} = 7 \wedge time_{K(t-2)} = 9) \wedge (temp\_in_{K(t-1)} = 21 \wedge temp\_out_{K(t-1)} = 8 \wedge time_{K(t-1)} = 9) \wedge (temp\_in_{K(t)} = 24 \wedge temp\_out_{K(t)} = 10 \wedge time_{K(t)} = 10)$$

Commands

The commands should capture the necessary changes to the control data of the system. In our adaptation rules, we assume that all commands are executed at the same time. Thus, we can directly set the mutated control parameters to the respective learned values by generating commands of the following form: $p1 = p1\_learned$.

Effect Extraction

To extract the effect of an adaptation, we use a simulation trace that results from applying the learned model parameters in the initial situation (where the system goals are violated) and simulate their effect on the environment model for some application specific time span that was also used to evaluate the quality during learning. We analyze this simulation trace to identify the point in time where the adaptation effect occurs. To this end, we use our distance function to identify a state in which the distance has improved and stays stable for some time. If the trace contains several candidates, a heuristic has to specify which point should be taken, e.g., the first one, or another one within a given time range after the adaptation. This point also provides us with the timing information when the effect is assumed to be observable. Note that we have based our effect time estimation on the time point where the effect is generally observable, instead of when the adaptive system is able to observe the effect in the knowledge base. This results in adaptation effects that are independent on concrete monitoring cycle times.

The system and environment state that is reached after this time span can be used to extract all changes in the environment parameters to describe the effect of those changes. To increase the precision of this initial adaptation rule, we refine the effect prediction based on this reference state and a relation $R \subseteq (SP \cup EP) \times EP$. R

specifies which control parameters of the system (SP) may possibly influence which environment parameters (EP), as well as known dependencies between environment parameters. The relation is used to remove all changes in environment parameters that were not influenced by the adaptation (e.g., passage of time). We assume this relation to be provided together with the runtime model.

Example After learning of (optimal) parameter values for our illustrating case study, we have (manually) performed our proposed rule extraction and generalization process. To this end, we have simulated the execution of the learned adaptation commands ($m = -1.2, n = 44$) directly after the detection of the undesired situation. The results are shown in the part right to the vertical line in Figure 7.4. At time point $40200s$, an indoor temperature (*temp_in*) of $24°C$ is monitored. Then, the adaptation is executed and the parameter values *m* and *n* are modified. At time point $40300s$, the indoor temperature decreases to $20°C$ as reaction to the adaptation in our simple model. This effect is monitored at time point $40800s$. From this simulation trace, we have extracted the following initial effect.

$temp\_in = 20 \wedge temp\_out = 12 \wedge time = 11$ **after** 100 seconds

In this subsection, we have described our initial rule generation. This initial rule is only applicable in a specific situation as described in the guard. In the following, we first describe how our rule learning algorithm can be applied to runtime models that are modeled in UPPAAL timed automata or SystemC. Afterwards, in Section 7.4, we describe our rule generalization and show how the initial adaptation rule for our air conditioning example can be further generalized.

7.3.6 Rule Learning on UPPAAL and SystemC Models

Rule Learning on UPPAAL Timed Automata

In this subsection, we describe how we apply our exemplary learning process on UPPAAL Timed Automata (UTAs) runtime models. The following description is valid for the usual UTAs as well as for the stochastic extension UPPAAL SMC that can be used to integrate dynamic behavior (ODEs) or stochastic behavior. In the following we use UTA to refer to any kind of UPPAAL timed automata.

Our basic genetic algorithm operates on parameters of UTAs. To enable the evaluation of the effect of adaptations in the current situation, we integrate an adaptation automaton (see Figure 7.6) into our system model that executes a series of parameter changes (*p1, p2*) at the specified point in time (`startAdaptTime`). The GA mutates the parameter values (`p1_learned, p2_learned`) of these changes. The mutated UTA model is simulated with the *UPPAAL simulator* or with the *UPPAAL SMC Model Checker* [DLL+15], depending on the type of the UTA, to observe the effect of mutations. It is also possible to combine dynamic or stochastic UPPAAL SMC models with our automatically extracted UPPAAL runtime models, e.g. to model a more realistic environment. However,

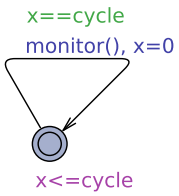


Figure 7.5: Abstract Monitoring Automaton for Rule Generation

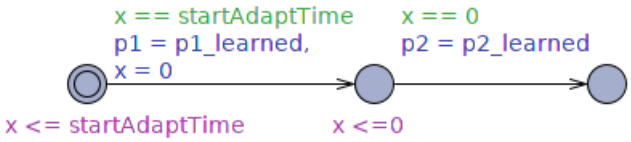


Figure 7.6: Adaptation Automaton for Learning

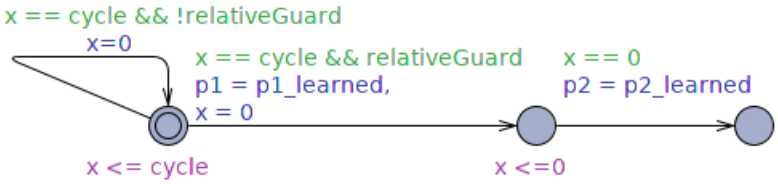


Figure 7.7: Modified Adaptation Automaton for Guard Generalization

to this end, the extracted models have to be manually adjusted, e.g. by expressing binary communication in terms of broadcast communication because UPPAAL SMC only supports this subset of UTAs. Within the simulation, the mutations are applied by the adaptation automaton, and the simulation proceeds for some specified time. The resulting data set (encoded in a text file) is fed into a fitness function to calculate the fitness of the mutation series. For the extraction of an initial adaptation rule, we add an abstract monitoring automaton as shown in Figure 7.5. It periodically (every *cycle* seconds) stores system and environment data to mimic the history information of the knowledge base.

Rule Learning on SystemC models

In this subsection, we describe how we apply our exemplary learning process on SystemC runtime models. To this end, our basic genetic algorithm operates on parameters of SystemC modules. To enable the evaluation of the effect of adaptations in the current situation, we integrate an adaptation process into our SystemC model that executes a series of parameter changes at a specified point in time. In Listing 7.7, we give a code snippet for our adaptation process with a default time resolution of seconds. This can be

```
1  void adaptation_process(int p1_learned, int p2_learned){
2    wait(startAdaptTime, SC_SEC);
3    p1 = p1_learned;
4    p2 = p2_learned;
5  }
```

Listing 7.7: Adaptation Process for Learning

```
1  void adaptation_process_withGuard(int p1_learned, int p2_learned){
2    bool waiting := false;
3    while(waiting){
4      wait(cycle, SC_SEC);
5      if(relativeGuard){
6        p1 = p1_learned;
7        p2 = p2_learned;
8        waiting = true;
9      }
10   }
11 }
```

Listing 7.8: Modified Adaptation Process for Guard Generalization

adjusted to the actual time resolution that is used in the adaptation layer. The code is equivalent to the timed automata presented above (cf. Figure 7.6). The GA mutates the parameter values of these changes. All variables that are used to calculate the fitness of the resulting model are traced in a Value Change Dump (vcd) trace file during the simulation of the mutated model with the SystemC simulation engine. For the extraction of an initial adaptation rule, we, furthermore, add an abstract monitoring process as shown in Listing 7.6. It periodically (every *cycle* seconds) stores system and environment data to mimic the history information of the knowledge base.

```
1  void monitoring_process(){
2    while(true){
3      wait(cycle, SC_SEC);
4      monitor();
5    }
6  }
```

Listing 7.6: Abstract Monitoring Process for Rule Generation

7.4 Generalization of Adaptation Rules

Our Rule Learning, as described in the last section, learns an initial adaptation rule that describes a concrete context and a concrete effect that can be assumed to occur after applying concrete control parameter values to the system. In a next step, we further generalize this initial rule to achieve a rule that is also applicable in other similar contexts. To this end, we perform a step-wise generalization process that includes *rule effect validation* on executable runtime models. Our step-wise generalization also makes use of the influence relation R between monitored parameters. At the end of the learning

```
1  input: rule set NEW_RULES, bool onlyGuardAbstraction
2  output: –
3  begin
4    if(¬ onlyGuardAbstraction)
5      NEW_RULES ← relativeEffect(NEW_RULES);
6      NEW_RULES ← relativeGuards(NEW_RULES);
7    end_if
8      NEW_RULES ← abstractGuards(NEW_RULES);
9    if(¬ onlyGuardAbstraction)
10     NEW_RULES ← relativeCommands(NEW_RULES);
11   mergeRules(NEW_RULES, K_Adapt);
12   comprehensiveVerification(NEW_RULES);
13 end
```

Algorithm 7.6: Rule Generalization

process, we perform a *comprehensive system verification* considering the interplay of all the adaptation rules in the adaptation logic K_{Adapt} to verify that adding the new rule does not compromise any important system properties. We only add the generalized rule if the verification was successful. If no suitable parameters could be learned to generate an initial rule or if the final system verification fails, the learning/verification component terminates and waits until it is triggered again by subsequent executions of the evaluation component in future monitoring cycles. In this case, we rely on the existence of a suitable safe operation mode in which the system can be set.

Our generalization is given in Algorithm 7.9. We differentiate between complete generalization as required for simulation-based learning and pure guard abstraction as performed after observation-based learning (indicated by the boolean `onlyGuardAbstraction`). We generalize initial rules stemming from simulation-based learning as follows:

1. **Relative Effects:** First, we generalize the effect by encoding the relative change of effected parameters. If an adaptation is able to restore some system goals, we also reflect this in the effect prediction, e.g., by referring to relevant reference values from the goals.

   Example As we can see in the simulation (see Figure 7.4) for our illustrating example, the adaptation restores our system goal. Thus, we generalize the effect by referring to the reference value of our temperature goal:

temp_in $=$ refTemp **after** 100 seconds.

2. **Relative History of Environment Parameters in Guards:** In a similar way, we, then, generalize the guard by using relative values w.r.t. previous history values. We use relative indexes that describe how many values we consider in the history, i.e., previous monitoring cycles. We refer to the current point in time with i, the previous monitoring cycle with $i-1$, etc. This allows us to describe relations in a general form that may apply to several concrete situations. Note that we assume a constant monitoring cycle time here. For variable cycle times, the time span

between history values additionally has to be captured. Whether this generalization might be valid, or whether at least some concrete values are necessary, can be checked by simulating the runtime model with the modified adaptation process. If this check is positive, we still have to perform a more thorough *rule effect validation* to check whether this also holds in other environment scenarios.

Example For our example scenario, we formulate the history of environment parameters in the guard using relative values:

$$
\begin{aligned}
&(\mathsf{temp\_in}_{K(t-2)} = \mathsf{temp\_in}_{K(t-3)} \wedge \\
&\mathsf{temp\_out}_{K(t-2)} = \mathsf{temp\_out}_{K(t-3)} \wedge \\
&\mathsf{time}_{K(t-2)} = \mathsf{time}_{K(t-3)}) \\
&\wedge \\
&(\mathsf{temp\_in}_{K(t-1)} = \mathsf{temp\_in}_{K(t-2)} \wedge \\
&\mathsf{temp\_out}_{K(t-1)} = \mathsf{temp\_out}_{K(t-2)} \wedge \\
&\mathsf{time}_{K(t-1)} = \mathsf{time}_{K(t-2)}) \\
&\wedge \\
&(\mathsf{temp\_in}_{K(t)} = \mathsf{temp\_in}_{K(t-1)} + 3 \wedge \\
&\mathsf{temp\_out}_{K(t)} = \mathsf{temp\_out}_{K(t-1)} + 2 \wedge \\
&\mathsf{time}_{K(t)} = \mathsf{time}_{K(t-1)} + 1).
\end{aligned}
$$

3. **Stepwise Abstraction of Guards:** The amount of included history is following heuristics and may be too restrictive. Thus, in a following step, we do a stepwise abstraction of the guard by removing conditions, and by relaxing conditions. To ensure that the resulting guard is still specific enough to guarantee the achievement of the effect, we include simulation-based rule effect validation after each step. Here, a validation that is based on a small amount of simulation runs is acceptable, as learning is followed by a stronger comprehensive system verification. We propose to start with the strongest abstraction, i.e. removing an entire condition, to get fast results. This is especially helpful for reducing the amount of history information in the guard. If removing some condition results in an inaccurate rule, it is included again and relaxed gradually by analyzing for which additional "neighboring" values the rule effect is still observable. This results in guards that describe intervals of values for some parameters. As there may exist dependencies between environment parameters, this should not only be done with single parameters, but also for combinations of parameters. Choosing appropriate candidates for stepwise abstraction should be based on the expertise of domain experts.

   Example For our example scenario, we have used our stepwise abstraction of the guard to weaken the guard and to reduce it to only consider the relevant conditions. Note that we added a clock constraint to our adaptation execution process to respect the timing of the MAPE-K loop, as this timing was included in the initial effect time prediction. First, we have omitted the daytime information because

it has no direct influence on the indoor temperature. After validating that this abstraction still leads to the same simulation results, we have reduced the amount of considered history information. To this end, we first have omitted the conditions for the history values *K(t-2)*, and after validation also the conditions for *K(t-1)*. Thus, as a result, we get the following adaptation rule:

$$\begin{aligned}&(\text{temp\_in}_{K(t)} = \text{temp\_in}_{K(t-1)} + 3 \wedge \\ &\text{temp\_out}_{K(t)} = \text{temp\_out}_{K(t-1)} + 2 \\ &\textbf{\&}\ n := 44; m := -1.2 \\ &\longrightarrow \text{temp\_in} = \text{refTemp}\ \textbf{after}\ 100\ \text{seconds}\end{aligned}$$

This rule is always applicable if the environment follows the ideal heating curve $f(x) = -1.2x+44$ and if the outdoor temperature increases by $2°C$ during one hour. Then the adaptation directly sets the optimal parameter values. To achieve more general adaptation rules (like our manually constructed rules) further generalization steps on the commands and the guard conditions have to be performed.

4. **Relative Commands:** After abstraction, the rule is applicable in other, similar contexts. In the last step, we replace the commands by relative commands to capture gradual changes on parameters. Note that in some cases, concrete values are necessary to describe a specific mode that should be achieved by adaptation. To check whether the abstraction is valid, we, again, perform simulation-based rule effect validation. Note that command generalization has to be validated after successful guard abstraction as it only makes sense if the original values of learned parameters may differ from the initial scenario.

   Example For our example, we get m := m − 2.2 and n := n + 14.

5. **Merging of Similar Rules:** In the end, we merge the resulting rule set with the current rule set K_{Adapt} to keep the number of rules low. To this end, we compare the resulting new adaptation rule with existing rules in the knowledge base. If a similar rule that includes the same commands and has the same effect, but a slightly different guard exists, these two rules can be merged by combining their guards.

As we assume the parameter names in the runtime models and in K_{Sys} to be identical, the generated rule can be inserted into the knowledge model K_{Adapt} after a final comprehensive system verification step. For pure guard abstraction, we only perform our iterative guard abstraction, followed by merging and verification.

To enable a quick check on the validity of our generalization steps, we replace the adaptation process that executed the adaptation commands at a specified time with a modified version of it. The new version checks every *cycle* time units whether the rule guard evaluates to true and executes the adaptations as soon as the guard is true. Note that we added a clock constraint to our adaptation execution process to respect

the timing of the MAPE-K loop, as this timing was included in the initial effect time prediction. The corresponding timed automaton is given in Figure 7.7, and the SystemC process is given in Listing 7.8.

The advantage of employing adaptation rules, even if they are learned at runtime, is that their comprehensibility and transparency of adaptation decisions is maintained.

7.5 Traceability of Decisions for Explainability

As our intelligent self-adaptive systems not only decide to adapt but also autonomously evolve their adaptation logic, we also aim at providing valuable information to make these decisions explainable as well. To achieve this, we create a log of learning results that consists of *explanation objects* for each newly learned rule that passed our verification. These objects are stored in our explanation basis and contain the original adaptation rule (if existent), the kind of learning (observation- or simulation-based learning), the learning result (refined context-specific rules, or newly learned rule), and, for observation-based learning, the underlying evaluation results (status, set of equivalence classes with considered rule executions, context conditions), and the timestamp of adding the rule to the knowledge base. To explain the results of simulation-based learning, we propose to log the updated RTMs as these will further evolve over time, and the influence relation R, which can, in principle, also change, e.g. due to topology changes or new insights on the actual influence relation that could be inferred from runtime observations. Note that runtime evolution of the influence relation has not been investigated in this thesis. Furthermore, we add the fitness function of the genetic algorithm to explain the evaluation of found solutions. Our *explanation objects*, thus, form an explanation basis for autonomous evolution decisions. With our adaptation and evolution explanation bases, we enable the tracing and explanation of autonomous decision from both layers.

7.6 Stability of Learned Adaptations

Disabling of rules in *Rule Accuracy Evaluation* and learning of new rules in *Simulation-Based Learning* can result in cyclic on-off adaptations of adaptation rules in unstable environments or system topologies. To prevent this, we propose to

a) only disable rules if a severe gap ($\geq \varepsilon_r$) between expected and actual effect was observed for this rule.

b) disable inaccurate rules but do not delete them immediately. Instead, keep a certain amount of old rules in memory for re-evaluation in case of learning requests.

Removing Rules The amount of old rules that are kept in the knowledge base, and the strategy of ultimately removing old rules when the knowledge base reaches its capacity limit, depends on the application and its environment characteristics. One possibility

would be to use a FIFO strategy for implementing forgetting in the memory. Another one would be to always forget the rule that was the least useful one in the past, where usefulness can be measured by the ratio between successful and unsuccessful applications of the rule (w.r.t. its expected effect), or by the average improvement in the distance that was achieved by applying the rule.

Re-Evaluation of Disabled Rules For re-evaluation, we propose to evaluate the accuracy and fitness of applicable disabled rules in the learning component after updating the RTMs before starting extensive learning. For each applicable disabled rule, first, the rule commands are applied to the RTMs. Then, the fitness is evaluated and, if the fitness is greater than a given threshold, rule verification is performed to check whether the effect prediction is accurate. In this case, the rule is reactivated. Otherwise, an initial rule for the current situation, with the commands of the old rule, and the actual effect is extracted and generalized. If the fitness of this rule is below the threshold, the rule is considered not suitable and is not reactivated. If no rule was found this way, normal learning starts.

While this re-evaluation is still costly, it can save time and resources compared to the time needed for exploration in the learning algorithm. In the best case, an existing old rule can be reactivated. In the worst case, several old rules have been re-evaluated, but due to the fact that no rule can be reactivated, learning still has to take place afterwards. The performance of re-evaluation can be increased by using some heuristics for deciding whether to re-evaluate or not. Basing re-evaluation only on the effect predictions encoded in the old rules is likely to lead to false results, as the rules were disabled because of their inaccuracy. Even, if they were only disabled because of topology changes, effect predictions may be outdated.

In our *Observation-Based Learning*, we refine adaptation rules to capture context-dependent effects. To provide a certain stability, we only refine rules if the effect has been observed several times (specified by the design parameter γ). However, if the effect deviates because of events that cannot be observed, e.g. due to a lack of sensors, the system will recognize a general effect shift, and adapt accordingly. As example consider an autonomous robot that drives against a low obstacle but is not able to detect the obstacle. As a consequence, it will assume a different effect of motor actions. If someone removes the obstacle, the robot has to refine the affected rules again. Detecting unobservable events it out of the scope of this thesis. Thus, we assume that the observed effect shift is stable if no causal context conditions can be inferred.

Summary In this chapter, we have described our evolution layer that enables the safe evolution of adaptation logics at runtime. Our *Rule Accuracy Evaluation* continuously evaluates and classifies the accuracy of timed adaptation rule effect expectations. If the observed deviation is severe, the respective adaptation rule is disabled to ensure the

correctness of planning results. In case of small deviations (below a given threshold), the rule is kept in the rule base and evaluation results are used to refine the effect expectations or the expected effect time in our *Observation-Based Learning*. The main idea of this learning mechanism is to extract context conditions from the equivalence classification results of our rule accuracy evaluation. Based on those, we split adaptation rules into context-specific rules with corrected effect or effect time expectations. With that, we enable the evolution of existing rules. To furthermore enable the definition of new adaptation rules for situations that are not covered by the existing rule base, we perform *Simulation-Based Learning*. The main idea of this learning mechanism is to perform heuristic search on executable runtime models that provide means to update them with observed environment behavior. We evaluate the fitness of learned adaptations based on the quantitative distance towards the adaptation goals that is provided by our goal model. Thus, the evaluation also considers runtime changes of the adaptation goals. The learning result is a specific adaptation rule for the uncovered situation. Each learning step (observation- and simulation-based) is followed by a rule generalization and verification step. With our generalization we enable the reduction of necessary learning steps and, thus, the frequency of applying inaccurate rules (for observation-based learning results) and the overall resource consumption (for simulation-based results). To ensure the correctness of learned rules, we continuously validate that the effect still holds during our rule generalization. In the final verification step, we perform comprehensive system verification to ensure that new rules do not compromise important system properties. In the next chapter, we describe our verification processes in detail.

8 Continuous Verification

How can we give guarantees for an evolving adaptation logic?

Intelligent cyber-physical systems, such as autonomous cars, autonomous drones, surgery robots, or smart grids, gain more and more influence in safety-critical domains. Thus, it becomes particularly important to ensure that these systems satisfy their requirements. In the presence of continuous learning in an ever-changing operation environment, design-time verification is not sufficient. Instead, continuous (i.e. design-time and runtime) verification is critical for the success of our intelligent self-adaptive systems. This need is, e.g., stressed by [Ghe10, DLGM+13, RCR+18, WBC+17]. In case of self-adaptive systems that are based on feedback loops, continuous verification should be explicitly addressed by integrating verification into the feedback loops [TVM+13].

In this thesis, we continuously ensure the correctness of the adaptation logics even in the presence of several uncertainties, autonomous learning, and adjustment of adaptation rules. To achieve this, we integrate different verification phases into our framework as illustrated in Figure 7.1 in Chapter 7. We rely on two different types of verification: light-weight *safety monitoring* (during *Analysis* and *Rule Accuracy Evaluation*) to detect invalid assumptions on the operational environment (and system topology), and *formal verification* to ensure the initial correctness at design time and the correctness of runtime changes of the adaptation rules w.r.t. the current knowledge on the environment.

Safety monitoring does not operate on models but, instead, gets information from the running system and detects and possibly prevents property violating behavior of the actual system. Thus, it does not rely on the accuracy of models and it avoids the complexity of comprehensive system verification with the cost of less coverage and, probably, the execution of faulty behavior before this is detected. To ensure the correctness of the evolution of adaptation rules, we integrate formal verification into our meta-adaptation layer: We perform *rule effect validation* under the current environment model to ensure that the effect prediction of the rule holds after rule application, and *comprehensive system verification*, where overall system properties are verified considering the interaction of adaptation layer and managed system. Both formal verification phases are based on runtime models that are built at design time and continuously updated at runtime.

Thus, those models get more accurate at runtime but there is still uncertainty about the actual behavior of the environment. As a consequence, our formal verification is only comprehensive with regard to the current knowledge about the environment. To deal with this, we, again, rely on our safety monitoring. In our framework instantiation in Chapter 8, we use model checking as formal verification technique. However, other techniques, like e.g. theorem solving, can also be used.

In summary, we combine both techniques to get the best of both worlds. Our formal verification is always executed before learning results are integrated into the actual adaptation logic. We thereby avoid faulty behavior that can be anticipated based on our models. For unanticipated behavior, we can rely on our safety monitoring and its ability to prevent major damages.

In the following, we first discuss how safety monitoring is embedded in our framework and afterwards, we describe our comprehensive verification in detail.

8.1 Safety Monitoring at Runtime

The main idea of embedding safety monitoring into our adaptation and evolution process is to detect unanticipated behavior that may lead to property violations. Unanticipated behavior may be caused by dynamic changes of the environment or system topology. Thus, we use safety monitoring to continuously observe whether a) important properties/ goals are satisfied, b) adaptation rules still fit to the current environment, and c) adaptation rules still fit to the current system topology.

The main idea of self-adaptive systems is to autonomously ensure that system goals are always satisfied, or only violated for a short amount of time before being re-established. In this sense, we are able to embed *safety monitoring at runtime* into our adaptation layer by inserting general system properties, e.g. safety properties, into the goal model (cf. Chapter 6). At runtime, their satisfaction will continuously be analyzed in the analysis phase of the adaptation layer (cf. Chapter 5). If appropriate counter measures are encoded in the adaptation rules, the adaptation layer will not only detect property violations but take actions to re-establish violated properties or, in case of proactive adaptation, anticipate possible violations and avoid them. Otherwise, the system will be set into a safe operation mode that guarantees safety but only with a subset of the desired functionality. Safe operation modes can also be defined as a degradation cascade of operation modes with decreasing functionality. In this case, either a degradation controller or our adaptation process can select the best available operation mode. In [ZKGG19], we have presented a systematic design method for such degradation controllers targeted for autonomous car platoons. A similar approach can be used to define countermeasures for violated goals within our adaptation layer.

To ensure that those counter measures will actually achieve their expected effect, i.e. that the underlying assumptions are still valid in the current environment, we continuously

observe the effect of executed rules in our *Rule Accuracy Evaluation* (cf. Section 7.1). If violations are detected, faulty rules are corrected based on past observations in *Observation-Based Learning* (cf. Section 7.2). To this end, our *Rule Accuracy Evaluation* also evaluates the severity of deviations and evaluates whether rules are still accurate enough to be executed without compromising correctness. Thereby, we balance protection and collecting more observations from further rule executions to enable more precise corrections.

A further threat to goal satisfaction are dynamic topology changes. If a component that is responsible for maintaining a certain system goal fails, this failure cannot be resolved by choosing an adaptation rule that relies on the failed component (e.g. setting a target temperature for a floor heating unit in a smart home will only be possible if such a heating unit is available). To avoid such incorrect adaptation plans, we continuously monitor the system topology and adjust the set of adaptation rules to reflect topology changes. If no alternative component is available, planning will fail and thus, the system will be set into a safe operation mode. Our topology evaluation also detects newly integrated components and triggers learning of new rules for these components. Afterwards, we are able to use new capabilities and thus, avoid underperformance of the adaptation logics.

Our safety monitoring provides a lightweight mechanism to detect behavior that may lead to property violations and to prevent (further) violations by taking appropriate counter measures. To minimize the amount of faulty behavior that is shown by our system, we exploit the collected knowledge of the system and perform comprehensive verification on our runtime models for each change of the adaptation logics. Thus, we find a balance between resource consuming model-based verification with a high coverage of possible runtime behavior and continuous safety monitoring to enable a fast detection of property violations due to unmodeled environment behavior. In the following, we describe our formal verification in detail.

8.2 Rule Effect Validation and Comprehensive System Verification

In this section, we describe our formal verification in detail. It is performed at design time, and also at runtime within our evolution layer. We first describe our general verification process and our generic tool chain. To model the interaction between adaptation layer and managed system, we propose to use an abstract model of the MAPE loop that can be varied in its degree of abstraction. We describe such a model in Section 8.2.2. Afterwards, we introduce general adaptation properties for self-adaptive systems and formalize them in (T)CTL.

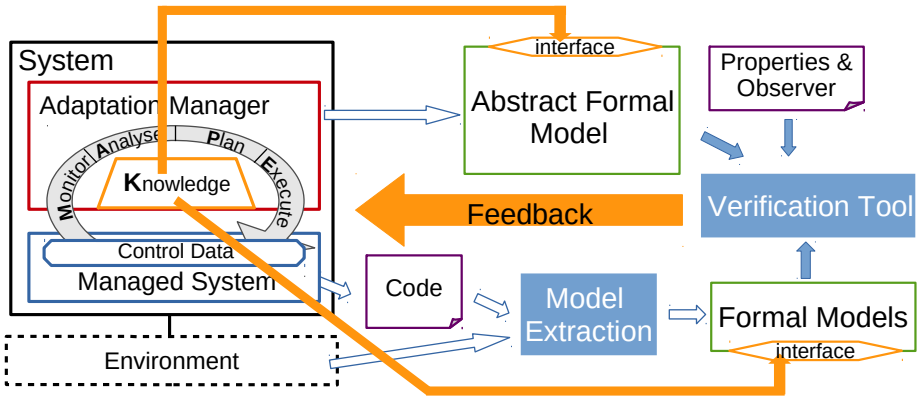


Figure 8.1: General Verification Tool Chain

8.2.1 General Verification Process

Our comprehensive verification operates on formal runtime models capturing the managed system, the environment, and the adaptation layer. In Figure 8.1, our general verification tool chain is depicted. It consists of an offline extraction of formal models (depicted with white arrows) and an online instantiation and update of those models (depicted with orange filled arrows), followed by the verification process. At design-time, we extract formal runtime models from the managed system, from an environment simulation, and from the adaptation layer (MAPE-K). The latter is abstracted during the extraction to increase verification performance. With these, our verification tool chain can be used to ensure correctness during the design process of the initial self-adaptive system. At runtime, those models are updated with the information of the knowledge base (depicted with orange filled arrows). The formal models and the system properties of interest are given to a verification tool. The verification result is used as feedback concerning the correctness of the system, e.g., during learning of new rules. The abstract models of the adaptation layer can range from a complete model of the whole MAPE-K loop, including the whole set of adaptation rules (necessary for comprehensive verification of system properties) to a model of one concrete rule execution (sufficient for rule effect validation). Finding the appropriate level of abstraction for verification in general is hard. Due to the explicit structure of our adaptation layer and the knowledge, we are able to provide an abstract MAPE process that includes all relevant aspects (e.g. timing) and may be varied in its level of abstraction, e.g. concerning the planning algorithm as we illustrate in Section 8.2.2.

8.2.2 Abstract MAPE Process

We use an abstract MAPE process that can be varied in its level of abstraction to increase verification performance. Our abstract MAPE process with variations of knowledge models (*K1, K2, ..*) and planning algorithms is depicted in Figure 8.2. It contains the

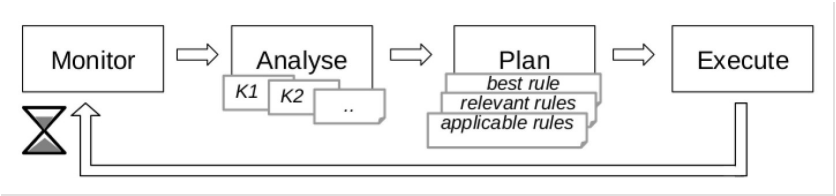


Figure 8.2: Abstract MAPE Process

four phases from the MAPE-K feedback loop: It periodically *monitors* the managed system and the environment. Then, it *analyzes* whether an adaptation is necessary. If an adaptation need was detected, it searches for the best applicable rule and applies it to the system. After that, the process waits for the next monitoring cycle. Within this general process, we abstract from complex knowledge models, and use simpler analysis and planning methods by introducing some nondeterminism. The degree of nondeterminism can range from nondeterministically choosing an applicable rule (i.e. any rule with a guard that evaluates to true), over choosing from relevant rules (i.e. any applicable rule that influences a violated goal), to a more concrete planner that takes dependencies between rules into account or even some distance-based planning that more or less corresponds to the actual planner but uses a simplified goal model. This degree of nondeterminism can be varied to balance between resource consumption and result accuracy. Although nondeterminism generally increases the state space for model checking, it requires less computation and variables that would be necessary to model concrete planning algorithms.

If the formal modeling mechanism supports timing, we propose to abstract from any timing behavior inside the MAPE-cycle and model timing only between cycles.

In the following, we introduce our (T)CTL formalization of important adaptation properties that we have analyzed in this thesis. Afterwards, we describe an instantiation of our general tool chain with SystemC and UPPAAL Timed Automata and show an abstract MAPE process in UPPAAL Timed Automata in Section 8.3.

8.2.3 Adaptation Properties

In the following, we first introduce important adaptation properties and give a brief explanation for each of them. Afterwards, we show how they can be formally expressed in (T)CTL to enable model checking.

In our *rule effect validation*, we verify that the encoded effect expectation of an adaptation rule holds based on the current environment knowledge. We call the corresponding property **successful execution of a rule**, which means that after each rule execution its encoded *effect* is always achieved in *time*. However, this local view is not sufficient to guarantee that the adaptation process is able to successfully re-establish violated system goals. To this end, we perform *comprehensive system verification* and define each system goal as a **weak invariant** that may be violated only for a limited

amount of time. In contrast, strict safety requirements that always have to be ensured, if necessary by switching to a safe operation mode, are represented by **strong invariant**s. An important property of self-adaptation is **system stability** in the sense of avoiding oscillating behavior by continuously performing adaptations. Hence, the system should always eventually resides in a stable operating system state for some time.

To enable model checking of these properties, they have to be formalized. To precisely capture the meaning of these properties and to enable model checking, we formalize them in (T)CTL:

1) **Successful Execution of a Rule:** To show that the *effect* of a rule r_i is always achieved in time, we introduce a corresponding observer. If the effect could not be observed within the specified expected effect time, the observer encodes this fact in a boolean variable or in a similar accessible encoding, e.g. a process location, *failed_effect*. The corresponding CTL reachability formula is $AG\ \neg r_i.\mathit{failed\_effect}$. Note that this alone does not ensure that the effect is always satisfied eventually. For this, it is necessary to additionally verify the formula $AG(r_i.\mathit{active} \rightarrow AF\ r_i.\mathit{effect})$, which encodes that each activation of rule i is eventually followed by a system state where the effect holds.

2) **Weak Invariant:** A weak invariant inv_w may only be invalid for a limited amount of time. It can be expressed with the following TCTL formula $AG(\neg inv_w \rightarrow AF_{\leq t}\ inv_w)$.

3) **Strong Invariant:** $AG\ inv_s$ expresses that a strong invariant inv_s is globally satisfied.

4) **Stability:** We define that the system always eventually resides in a stable operating system state (where no adaptation takes place) for some time over the reachability of such a state. To this end, we require the encoding of rules to provide a boolean *active* that is set to true during rule execution. Then, stability can be expressed as $AG(true \rightarrow AF\ \forall i \in \mathit{rules}.\neg r_i\mathit{active})$. This untimed property does not specify how long the system stays in such a stable state. If we want to express that the system is stable for at least *min_stable* time units, we have to introduce a clock *stable* that is set to 0 the moment the lastly executed rule becomes inactive. Now, we can express this property with: $AGAF\ (\forall i.\neg r_i.active \wedge stable \geq min\_stable)$.

In the next section, we exemplarily instantiate our tool chain with SystemC and Timed Automata and explain how these adaptation properties can be verified using the UPPAAL model checker.

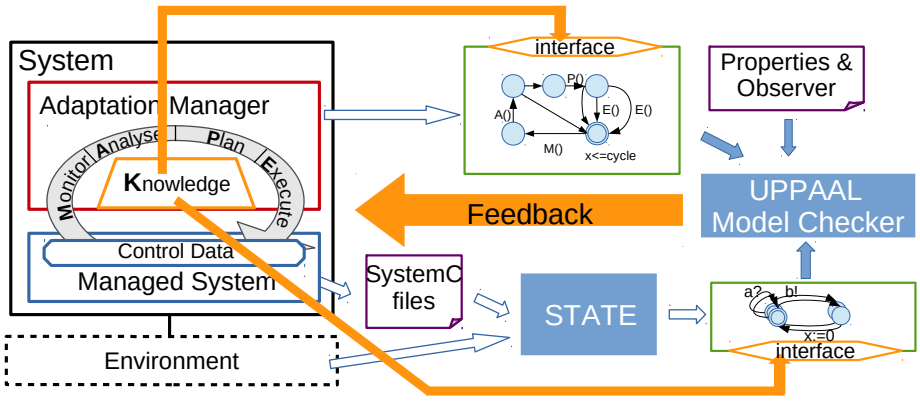


Figure 8.3: Instantiated Tool Chain with UPPAAL TA

8.3 Tool-Chain Instantiation with SystemC and UPPAAL Timed Automata

To show the applicability of our verification tool chain, we exemplarily instantiate it with SystemC as design language for the managed system and UPPAAL timed automata as formal modeling language for our runtime models. In Figure 8.3, we show our instantiated tool chain. We transform the functional components of the managed system and environment simulation modules from the SystemC implementation into equivalent UPPAAL timed automata (UTA) models using the existing transformation engine STATE [HFG08, HPG15] as described in Section 7.3.2. The MAPE-K adaptation layer is not transformed directly, but mapped to abstract UTA models of the MAPE-K layer to reduce the resulting state space and to reduce verification time. Our abstract UTA models consist of a generic MAPE-K template following our abstract MAPE process as described in Section 8.2.2 and a rule automaton together with a corresponding observer automaton for each adaptation rule. In the following, we describe our MAPE-K template and our generic rule automaton.

8.3.1 Abstract MAPE-K Template and Formal Rule Automata

Abstract MAPE-K Template

Our generic MAPE-K template is depicted in Figure 8.4. As prescribed by our abstract MAPE process, it periodically (every *cycle* time units) *monitors* the managed system and the environment. Then, it *analyzes* whether an adaptation is necessary. If an *adaptation need* was detected, it searches for the best applicable rule (*findBestRules()*) and applies it to the system. After that, the template waits for the next monitoring cycle.

After planning, one of the applicable rules is immediately chosen non-deterministically (the rule events r_i are urgent). If, however, no rule is applicable, the planner leaves that location just after recognizing this. Thereby, the guard $x > 0$ is technically used

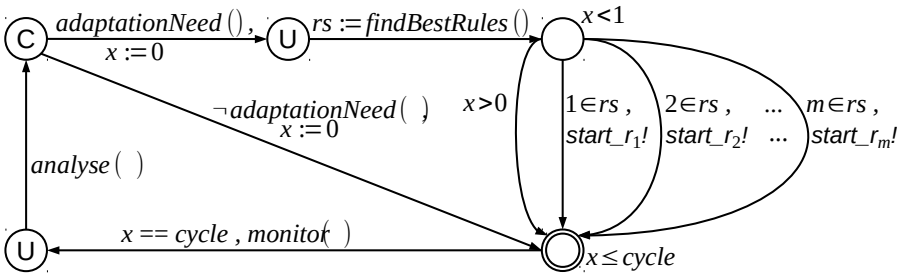


Figure 8.4: Our generic MAPE-K template from [KGG16]

to give applicable rules priority over the implicit observation that no rule is applicable. The invariant $x < 1$ ensures that the automaton reaches the lower location and the execution phase is finished. Then, the planner waits for *cycle* time units before the system parameters are monitored again. The timing behavior inside the MAPE-cycle is abstracted and is only modeled between cycles.

The most crucial part is calculating the set *rs* of best fitting rules within *findBestRules()*, which allows us to precisely adjust the abstraction of the actual planning component. In its most abstract form, the entire set of rules can be returned of which one is non-deterministically chosen subsequently. When refining the adaptation logic, the selection of rules can be adjusted to take, e.g., dependencies between rules into account. A detailed adaptation logic can be achieved by transforming the planning logic implemented in SystemC to our abstract planner by mapping the rules to our rule automata, copying the distance functions of the subgoals, and adding methods that compute the expected effect of each rule (as encoded in the rules) to the planner. These are then used in `findBestRules()` to determine the best adaptation action of all *relevant* rules (effecting a violated subgoal). To illustrate this, we show a generic excerpt of the function `findBestRules()` of the resulting UPPAAL model in Listing 8.1.

```
1 if(relevant[1]){
2    if(guard_rule1() ){
3     //calculate distance of estimated effect
4     estDist = estDist_rule1();
5     if(estDist < dist){ //rule is current best
6       dist = estDist; currentBest = 1; }}}
```

Listing 8.1: Excerpt of `findBestRules()`

Such a transformation could be automatised (for basic distance functions that are restricted to language elements of UPPAAL) by using, e.g., some static analyses.

Example For our rule *increase n*, presented in Section 5.1.2, we get the code that is depicted in Listing 8.2

In our generic planner, we abstract from polling of sensors or time-consuming planning to focus on the interplay of adaptation rules forming the essence of the adaptation logic. However, these phases could be enriched by more sophisticated components based on, e.g., formal templates for self-adaptive systems [dIIW15].

```
1 if(relevant[1]){
2     if(temp_out < refTemp && temp_in + tolerance < refTemp
3         && temp_in = temp_in_old){
4      //estimated effect with quadratic distance function
5      estDist = 0,5·(temp_in − refTemp)·(temp_in − refTemp);
6     . . .}}
```

Listing 8.2: Excerpt of `findBestRules()`

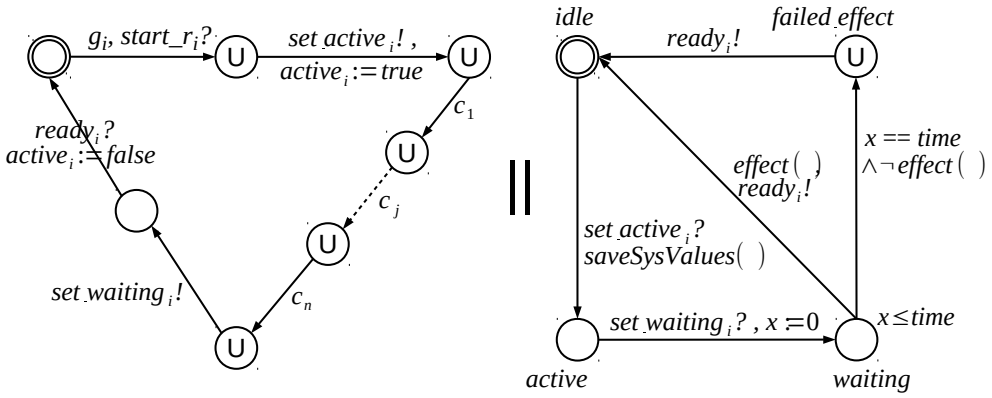


Figure 8.5: Generic Rule Automaton

Formal Rule Automata

As described in Section 5.1.2, the general structure of our timed adaptation rules consisting of a guard g_i, commands $c_1, c_2, ..., c_n$, an *effect* formula describing the expected effect, and the expected *time* after which the effect can be generally observed:

$$r_i : g_i \ \textbf{\&} \ c_1; c_2; \ldots; c_n \longrightarrow \textit{effect} \ \textbf{after} \ time_{\pm[timeTolerance]}$$

timeTolerance is used to allow for a certain delay of the effect due to a more coarse grained monitoring. In our generic rule automaton, we omit the *timeTolerance* and, instead, evaluate the effect on the actual environment values.

On the left side of Figure 8.5, we give the definition of a generic rule automaton for a rule r_i. When the rule is triggered using the event $start\_r_i$, it is checked whether the rule is applicable by checking the guard on the shared knowledge variables. The activation of the rule is followed by executing the several commands of the rule manipulating the corresponding control data. As soon as all commands have been executed, the rule automaton waits for *time* to pass by. The observer automaton on the right side of the picture is explained in the next subsection.

Interface between Adaptation Layer and Managed System To ensure that changes in the control parameters are detected quickly after the execution of adaptation steps, we use an observer thread inside the SystemC design that continuously observes the control parameters and, in case of changes, notifies affected functional components. This thread

is also included in the generated UTA model and ensures that the control parameter can serve as interface between the generated model of the managed system and our abstract model of the adaptation layer.

Our formalization of our adaptation layer and our adaptation rules in UPPAAL timed automata preserves their structure and leads to models that are human-readable and comprehensible. Thus, results on those models can be easily retraced to the system. As *verification tool* in our instantiated tool chain, we use the UPPAAL model checker to verify our adaptation properties as defined above. In the following, we explain our verification of adaptation properties with the UPPAAL model checker.

8.3.2 Analysis of Adaptation Properties in Timed Automata

To analyze the adaptation behavior of self-adaptive timed automata models w.r.t. our adaptation properties (cf. Section 8.2.3), we define effect observer for each rule, and we reformulate the TCTL properties because of the limited support for TCTL of the UPPAAL model checker.

Observing Rule Automata On the right side of Figure 8.5, we define an observer pattern for each rule. Before a rule automaton executes its commands, the values of relevant knowledge variables are saved to local variables based on which the effect can be checked later on. After executing the commands, the observer waits in location *waiting* for at most *time* time units. If *effect* holds within this time, the observer returns to its *idle* location as soon as possible because we define $ready_i$ to be urgent. If the effect does not hold after *time* time units, the observer passes the location *failed_effect* before going back to the *idle* location. In both cases, the *ready* event is emitted setting the rule automaton back to its initial state. This means that, here, only one instance of a rule can be active at a time. This is necessary for keeping track of the duration until the *effect* of a rule takes place. If, however, it is necessary that several instances (with a fixed maximum number) of the same rule are active at the same time, it can be copied to form new rules.

Property Formulation for the UPPAAL Model Checker Our generic rule automata allow us to use the UPPAAL model checker for verifying several adaptation properties introduced above. To this end, we show how each property can be equivalently expressed in the (T)CTL dialect of UPPAAL. In Table 8.1, we compare both formalizations.

1) **Successful Execution of a Rule:** Here, we can use the same formula as in CTL. We only have to replace G by [], which is the UPPAAL symbol for $\Box$. In our UPPAAL model, *failed_effect* is a location in the corresponding rule observer (see Figure 8.5). Whenever a rule is applied, it will eventually have to take the *effect()* edge or the $\neg$*effect()* edge. To show that the effect is always satisfied eventually,

Table 8.1: Adaptation Properties formalized in (T)CTL and UPPAAL

	(T)CTL	UPPAAL
1)	$AG\ \neg r_i.$*failed_effect* $AG\ (r_i.$*active* $\rightarrow AF\ r_i.$*effect*$)$	$A[]\ \neg r_i.$*failed_effect* $r_i.$*active* - -> $r_i.$*effect*.
2)	$AG\ (\neg inv_w \rightarrow AF_{\leq t}\ inv_w)$	$\neg inv_w$- -> inv_w (untimed) $A[]$ (*obs.invalid* `imply` $c + d \leq t$) (timed)
3)	$AG\ inv_s$	$A[]\ inv_s$
4)	$AG\ (true \rightarrow AF\ \forall i \in$ *rules*. $\neg r_i.$*active*$)$ $AGAF\ (\forall i \in$ *rules*. $\neg r_i.active \wedge stable \geq min\_stable)$	$true$ - -> $\forall i \in$ *rules*. $\neg$*active*$_i$ $true$ - -> $((\forall i \in$ *rules*. $\neg r_i.active) \wedge stable \geq min\_stable)$

we also have to translate the second formula to the equivalent UPPAAL formula by using the *leads-to* operator.

Example We have verified the successful execution of all initial rules in our case study (given in Section 5.1.2) and all learned rules (Chapter 7).

2) **Weak Invariant:** The TCTL formula that expresses that a weak invariant inv_w may only be continuously invalid for a fixed amount of time is not supported in the TCTL subset of UPPAAL. To check this, a clock would need to be reset as soon as the weak invariant inv_w is not satisfied any more, which is in general not realisable in UPPAAL. Instead, we can check either a weaker untimed property $\neg inv_w$ -> inv_w, or a stronger timed property that needs an additional invariant observer automaton as depicted in Figure 8.6. It checks with a sampling rate d whether the invariant still holds or not, and if not, a fresh clock c is reset to 0 and the observer switches to *obs.invalid*. When inv_w holds again the observer automaton switches back to its initial location. Now, we can check the formula $A[]$(*obs.invalid* `imply` $c + d \leq t$).

Example For our illustrating case study, such a weak invariant is, for example, that the indoor temperature usually is at $20°C$ at daytime.

3) **Strong Invariant:** To show that a strong invariant inv_s is globally satisfied, we can use the equivalent formula $A[]\ inv_s$.

Example For our illustrating case study, such a strong invariant is, for example, that the flow temperature always resides in a certain interval, and that the system is deadlock free.

4) **Stability:** The reachability of a stable operating system state, can be equivalently expressed by using the *leads-to* operator. To check how long the system stays

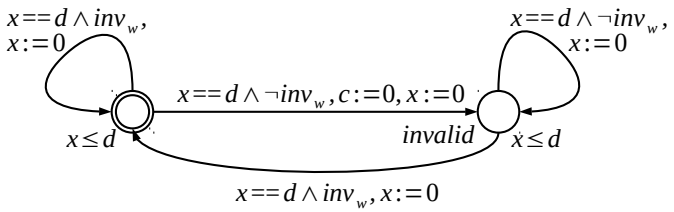


Figure 8.6: Observer for Weak Invariants

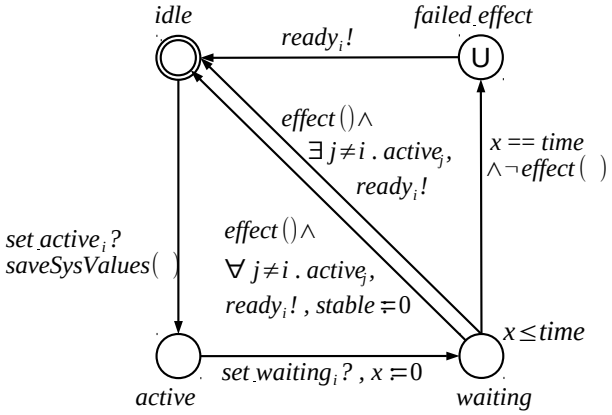


Figure 8.7: Modified Rule Observer for Stability

there, we slightly extend the rule observers by resetting a clock *stable* as soon as the currently last active rule observer returns to its idle location. Then, we can verify $true$ -> $(\forall i.\neg active_i) \wedge stable \geq min\_stable$. This extension is depicted in Figure 8.7. It is done by duplicating the *effect* edge. One of them carries the additional guard $\exists j \neq i.\ active_j$. The other one carries the additional guard $\forall j \neq i.\ \neg active_j$ and the update $stable := 0$ where $stable$ is a fresh shared clock. Now, stable is set to 0 in the moment when all rules become *idle* (again). Note that this works under the condition that the effect of a rule never fails. If the effect of a rule may fail, the edge from r_i.*failed_effect* to r_i.*active* has to be duplicated accordingly.

Summary In this chapter, we have described our verification processes that are embedded in our framework. We combine lightweight safety monitoring and comprehensive system verification, to find a balance between resource consuming model-based verification with a high coverage of possible runtime behavior and continuous verification to enable a fast detection of property violations due to unanticipated environment behavior. Our safety monitoring consists of three parts. It continuously observes whether

a) important properties, which have to encoded in the adaptation goals, are satisfied, b) adaptation rules still fit to the current environment, and c) adaptation rules still fit to the current system topology. For comprehensive system verification, we have presented our general verification process, have introduced our abstract MAPE process, and have formalized important adaptation properties. Afterwards, we have presented an exemplary instantiation of our verification tool chain based on an automatic extraction of timed automata models from a SystemC implementation of the managed system and an abstract timed automata model of our abstract MAPE-K process, together with a timed automata formalization of our timed adaptation rules. The adaptation logic can be expressed at an arbitrary level of abstraction. This enables analysis of an existing implementation of the adaptation layer, as well as model-based development of this layer using simulation and verification results to gradually refine the adaptation logic. Furthermore, we have shown how general property classes including timing properties can be verified automatically, and provided corresponding observer automata.

In the last chapters (Chap. 5 to 8), we have presented all parts of our framework in detail and have given illustrating examples on our running example of a smart temperature control system. In the next chapter, we evaluate our framework on three case studies from different domains. With respect to formal verification, we present our setting for verification experiments and respective verification times for our smart temperature case study.

9 Evaluation

We have implemented our approach and evaluated it on three case studies. We have used our smart temperature control system as a complete case study, where we have evaluated all parts of our framework. The temperature control system is well suited to illustrate all aspects as it was designed for this purpose. In addition, we have developed two further case studies, each for a specific evaluation purpose. We evaluate our goal model and its quantitative distance evaluation on an autonomous drone delivery system and the performance of the genetic algorithm on a self-organizing production system.In this chapter, we first describe important aspects of our implementation. Afterwards, we describe our case studies and provide experimental results.

9.1 Implementation

We have implemented most parts of our framework in the system description language SystemC to easily integrate it with SystemC implementations of a manged system. To this end, all processes are implemented in C++ and embedded into SystemC Processes. Communication with the managed system is realized via SystemC channels. We have implemented our simulation-based learning separately as it may run in parallel, e.g. on an external server. In the following we describe interesting aspects of our implementation.

Goal Model A first prototype of our goal model was developed by Adrian Lohr in his bachelor thesis [Loh16]. We have improved the distance calculation and heavily extended the modeling features for the goal model afterwards. To achieve an efficient and modular implementation, we have build a completely new implementation. We have implemented our goal model and our distance calculation in Java as described in [KGG18d], as well as in C++ to integrate the goal model into our framework. The basic classes are the goal classes, which refer to their respective children and preceding goals. For various kinds of local distance functions for leaf goals (optimization, exact) and parent goals (AND, OR), and different kinds of precedences (linear, "0/1"), we provide templates for predefined and user-defined distance and precedence functions, respectively. With them, concrete local distance functions that capture the behavior of a goal's formula can conveniently be derived.

Our goal model implementation is modular, extensible, and realizes all features presented in Chapter 6. We have realized our distance calculation algorithm close to its formal definition as a top-down recursive function, initially applied to the root node of the goal model. To increase efficiency, we perform distance calculation, derived importance calculation, and activeness calculation only once per goal. After that, the current values are stored as attributes of each goal. They are reused if the same goal is visited again because of precedences, for example. As a result, our distance calculation algorithm has a linear complexity in the number of goals. For later distance calculation for new system states, these values are reset.

Evolution Layer Our rule accuracy evaluation consists of a rule status evaluation and a deviation classification. Our status evaluation determines for each executed rule whether the expected effect has occurred in time. To achieve this, we have parallelized our evaluation algorithm (cf. Algorithm 7.1). We have defined a generic observer thread that waits until the effect time (earliest time the effect is expected to be observable) has past and checks whether the effect occurred. If not, we periodically poll the sensors and evaluate the effect for at most timeTolerance additional time units. The cycle time for this polling can be set to zero to specify that reevaluation is performed only once after timeTolerance time units. We dynamically instantiate an observer thread for each execution of an adaptation rule. Each observer thread copies its evaluated rule instance in the evaluation data base.

Our observation based learning consists of the extraction of context conditions, for which we use the JRip implementation of classification rule learning with RIPPER. It is included in WEKA [WFHP16] and provides a fast calculation of classification rules, as well as the option to validate the results with ten-fold cross-validation. We have not automatized the execution of JRip and the construction of our corrected context-specific adaptation rules. However, this only requires a minor adjustment of the printing routine for our evaluation results to include the header information for WEKA, parsing of the resulting classification rules and the construction of new rule objects based on our description in Section 7.2.

Our simulation-based learning is implemented separately to enable the deployment on an external server. It consists of our genetic algorithm, our initial rule extraction and our rule generalization. The first prototype of the genetic algorithm was developed by our bachelor student Kevin Styp-Rekowski. He had developed a genetic algorithm that operates on parameterized UPPAAL timed automata in his bachelor thesis [SR16]. We have improved his prototype and extended it to support parameterized SystemC models as runtime models (RTMs). We use text files as interface to the RTMs. To this end, we have implemented parser routines to extract the controllable parameters from a parameter specification text file and the recorded parameter values from RTM-specific trace files. The parameter specification contains the amount of controllable parameters and pairs of parameter name and initial value. For parameter adaptation and subsequent simulation

of the resulting model, we have implemented a find-and-replace routine for our exemplary RTM languages UPPAAL Timed Automata and SystemC. We have implemented our initial rule extraction in Java, too. Our rule generalization is currently not automatized. For our evaluation of the rule generalization, we have performed all steps manually.

Explanation Basis Our explanation basis is realized as vector of history objects within our SystemC framework. In the end of the SystemC simulation, our explanation basis is stored in a CSV-file for further processing. Currently, our history objects only include the explanation information for decisions of the adaptation layer. We have not implemented the explanation objects for the evolution layer, as rule learning is realized in a separate implementation.

In the following, we describe our evaluation on our smart temperature control system. Afterwards, we proceed with our further case studies.

9.2 Case Studies and Evaluation

In this section, we describe the case studies that we have used to evaluate our framework and present the experimental results.

9.2.1 Smart Temperature Control System

We have implemented our case study in the system description language SystemC and evaluated our observation- and simulation-based learning procedures, as well as our formal verification with different scenarios.

Rule Accuracy Evaluation and Observation-Based Learning

We have evaluated our rule accuracy evaluation and observation-based learning with two scenarios. In the first scenario, the radiator of our heating system is exchanged. For the new radiator, a constant shift in the effect of adaptation rules can be observed. In the second scenario, the effect of flow temperature changes depends on the outdoor temperature. For temperatures that are greater or equal to $15°C$ degree, any increase in the flow temperature results in the same increase in the room temperature.

Experiment 1 - New Radiator To evaluate the accuracy of the rule increase_n after the radiator exchange, we have conducted the following experiment. We have removed the air conditioning and the sun intensity sensor from our temperature control system. We thereby focus on the heating system and avoid disturbances from sun. To enable the observation of several executions of this rule, we initialized the heating parameters of our heating controller with $m = -1.2$, and $n = 32$ and the ideal heating curve of the environment with $m = -1.2$, and $n = 44$. We simulated one day (24 hours) in spring in our heating system after the installation of the radiator. This took a few seconds

temp_in	temp_out	time	deviation (temp_in, energy)
4	5	1	(1,0)
6	5	1	(1,0)
8	5	1	(1,0)
10	5	1	(1,0)
12	5	1	(1,0)
14	5	1	(1,0)

Table 9.1: Rule Accuracy Evaluation for increase_n of Experiment 1

only. The results of our rule accuracy evaluation are shown in Table 9.1. The constant shift in the resulting indoor temperature was correctly observed and classified by our rule accuracy evaluation.

Experiment 2 - Context-dependent Shift To evaluate the extraction of context conditions with JRip, we have conducted the following experiment. We have removed the air conditioning and the sun intensity sensor from our temperature control system. We thereby focus on the heating system and avoid disturbances from sun. We initialized the heating parameters of our heating controller with $m = -1.2$, and $n = 32$ and the ideal heating curve of the environment with $m = -1.2$, and $n = 44$. Additionally, we have introduced a thread into our environment that switches between three ideal heating curves: 1) $-1.2x + 44$, 2) $-1.2x + 36$, and 3) $-1.2x + 52$. The change from 1) to 2) is introduced after eight simulation hours, and afterwards the environment switches between 2) and 3) every eight simulation hours. The results of our accuracy evaluation after 96 simulation hours in a spring time scenario are depicted in Table 9.2 and 9.3. Note that we have depicted the equivalence class for the temperature only. The two observations with a deviation of $2°C$ are caused both caused by an environmental switch from rule 2) two rule 3) and the subsequent stabilization of the environment. In a normal scenario, the environment would not switch as often as in our experiment. Thus, these observations would not be fed into learning as they would not occur often. However, we used them as additional noise to see whether JRip can handle this. We have manually fed the results into WEKA and applied JRip, the RIPPER implementation of WEKA, for both rules increase_n and decrease_n.

JRip returned with the following rules for increase_n after 0.01 seconds:

- *IF* $(\text{time} >= 17)$ *and* $(\text{temp\_in} <= 15)$ *and* $(\text{temp\_in} >= 15)$ *THEN* $\text{deviation}_{\text{temp\_in}} = 2$
- (*ELSE IF* $(\text{temp\_out} >= 15)$ *THEN* $\text{deviation}_{\text{temp}} = 1$
- *ELSE* $\text{deviation}_{\text{temp\_in}} = 0$

temp_in	temp_out	time	$\text{deviation}_{\text{temp\_in}}$
15	15	17	2
15	15	17	2
17	6	8	0
18	6	8	0
17	6	8	0
18	6	8	0
18	15	13	1
18	15	13	1
14	15	17	1
16	15	17	1
18	15	17	1
14	15	17	1
16	15	17	1
18	15	17	1
10	5	1	0
11	5	1	0
12	5	1	0
13	5	1	0
14	5	1	0
12	5	1	0
13	5	1	0
14	5	1	0

Table 9.2: Rule Accuracy Evaluation for $\text{increase}_{\text{n}}$ in Experiment 2

temp_in	temp_out	time	$\text{deviation}_{\text{temp\_in}}$
22	15	13	-1
22	15	13	-1
26	15	17	-1
24	15	17	-1
22	15	17	-1
26	15	17	-1
24	15	17	-1
22	15	17	-1
28	15	17	-1
28	15	17	-1
23	6	8	0
22	6	8	0
23	6	8	0
22	6	8	0
20	5	1	0
19	5	1	0
18	5	1	0
20	5	1	0
19	5	1	0
18	5	1	0

Table 9.3: Rule Accuracy Evaluation for $\text{decrease}_{\text{n}}$ in Experiment 2

We have performed 10-fold cross-validation in WEKA, which resulted in two miss-classified rules. This happens, because if the first two rows of Table 9.2 are not included in the training set, the first rule will not be learned. Thus, these two data sets will be miss-classified.

For decrease$_n$ the rules are:

- *IF* (temp_out $>= 15$) *THEN* deviation$_{\text{temp}} = -1$
- *ELSE* deviation$_{\text{temp_in}} = 0$

Based on the result from WEKA we can perform our rule correction as explained in Section 7.2.

In our experiments, we have shown that our rule accuracy evaluation is able to detect and classify recurring deviations, and that we can optimize our our adaptation rules to handle recurring context-dependent deviations based on our proposed observation-based learning.

Simulation-Based Learning

To evaluate our genetic algorithm, we demonstrate the learning of a suitable adaptation rule for a concrete situation in a variant of our smart temperature control system without an air conditioning. We consider the case where the indoor temperature exceeds the desired temperature due to a incorrect assumption on the heat transmission of the building (an incorrect heating curve). Note that this situation could be resolved using our existing general adaptation rules in several adaptation steps. For this experiment, we disregard our manually engineered set of adaptation rules and consider our environment as black box to evaluate whether the genetic algorithm is able to learn a suitable rule. We have instantiated the environment with the ideal heating curve $f(x) = -1.2x + 44$ and the system with the initial heating parameters gradient $m = 1$ and offset $n = 30$. We compare learning based on formal UPPAAL timed automata simulation to learning based on SystemC simulation.

We have instantiated our genetic algorithm on stochastic timed automata to enable the simulation with trace recording. The mutation rules specify the step size and range of changes on m (0.1 in [-0.3, 0.3]) and n (1 in [-5, 5]). We have set the number of iterations (15), the amount of generated children per iteration (20), the size of the parent population (2), the simulation length (22 hours in the model) and the number of simulation runs per mutated automaton (1, due to the deterministic models). As fitness function we have used the quadratic error between current and desired indoor temperature measured on several samples of the same simulation run to avoid wrong values at the intersection of the current and the ideal heating curve. We have applied our genetic algorithm in a spring environment scenario (outdoor temperature values between 5 and $20°C$) in which the current parameters lead to a situation where the indoor temperature reaches $24°C$ at 11 a.m.

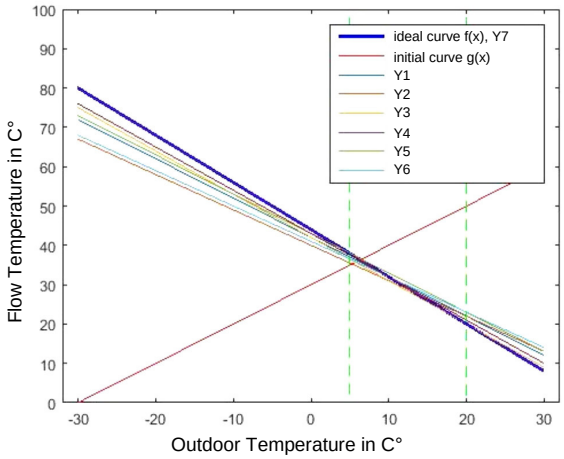


Figure 9.1: Optimization of Heating Curve Parameters

children	10 iterations	15 iterations	20 iterations
10	2.2s (101, -0.97)	3.1s (151, -0.06)	4.1s (201, 0.00)
15	3.3s (151, -0.11)	4.6s (226, -0.01)	6.3s (301, 0.00)
20	4.2s (201, -0.20)	6.2s (301, 0.00)	8.9s (401, 0.00)

Table 9.4: Comparison of Genetic Algorithm Variants Using UPPAAL SMC: runtime (#simulations, achieved fitness)

In the following, we present the results of our evaluation. All learning experiments were carried out on a 64 bit Linux system with an Intel Core i7-3520M CPU with 2.9 GHz and 16 GB of RAM.

Figure 9.1 shows the learned heating curves of the GA during seven iterations. Here, our learning algorithm has found the optimal parameters ($m = -1.2, n = 44$) after seven iterations. Note that that there are other "optimal" parameters (e.g. $m = -1.1$, $n = 43$) that do not match the current ideal heating curve, but which also establish the intended indoor temperature in the considered simulation interval. We have performed simulations on an environment that showed outdoor temperatures in the highlighted area of $5°C$ to $20°C$. Here, in the last iteration the optimal solution was reached. In our example, we were able to use a monotone fitness function without local optima that ensures that eventually a global optimum (fitness of 0.0) is reached after sufficiently many iterations. In more complex case studies, we can only expect the GA to find a better solution, but not necessarily the best one, as the GA could get stuck in a local optimum. Table 9.4

children	10 iterations	15 iterations	20 iterations
10	0.9s (101, -2.10)	1.1s (151, -0.45)	1.4s (201, 0.00)
15	1.1s (151, -0.56)	1.7s (226, 0.00)	2.2s (301, 0.00)
20	1.5s (201, -0.18)	2.2s (301, 0.00)	2.8s (401, 0.00)

Table 9.5: Comparison of Genetic Algorithm Variants Using SystemC Simulation: runtime (#simulations, achieved fitness)

shows a runtime comparison of GA variants that differ in the number of iterations and children on our temperature example. Note that we did not let the algorithm terminate before the entire number of iterations was performed. The given runtimes are average values of 10 runs for each variant. Furthermore, we include the performed number of simulations and the fitness value of the learned model. The most time consuming task in the GA are the model simulations for the fitness calculation, thus we get similar results for the symmetric variants. Generally, a GA achieves better results, if we increase the number of iterations and children, thus, the number of evaluated parameters.

We have also performed experiments using the SystemC implementation for simulation, directly. To this end, we have used the SystemC simulation capabilities to calculate the fitness of individuals. The results are shown in Table 9.5. In comparison to the UPPAAL SMC-based learning, runtimes were better when using SystemC directly. The reason is that there is some overhead in the generated UPPAAL model of a SystemC design. Moreover, SystemC is highly optimized for simulation purposes. However, the results also show that it is feasible to learn based on formal models, which also has the advantage of providing a basis for formal verification of learned rules.

Verification

To evaluate our formal verification approach, we use a smaller version of our smart temperature control system as presented in [KGG16, KGG18b]. In this version, we omit the air conditioning, the sun intensity sensor, and the goal to minimize the energy consumption. Thus, our managed system consists of the heating unit only. To verify the corresponding subset of adaptation rules that specify how to adapt the heating parameters *m* and *n*, we have transformed the SystemC code of the managed system and the environment into UPPAAL timed automata (UTA) with the *SystemC to Timed Automata Transformation Engine* [HPG15]. We have instantiated our generic MAPE-K template (see Figure 8.4, p. 118) with our four adaptation rules (increase and decrease of m and n). Furthermore, monitor keeps track of indoor and outdoor temperature, analyze compares the desired and the current indoor temperature and planning is captured in the function `findBestRules()`. Here, it simply returns the set of all four adaptation rules, as only one of these rules can be executed at each time.

We have defined the following properties that should not be violated (invariants):

- The system never deadlocks: $A[]\ not\ deadlock$
- The flow temperature never reaches the minimal and maximal values: $A[]\ (tempFlow > MIN\_FLOW\ \&\&\ tempFlow < MAX\_FLOW)$

or that should be restored (weak invariant) by the rules:

- The room temperature should be close to the desired temperature: $not\ (temp\_in\ \pm c = refTemp)$ - -> $(temp\_in\ \pm c = refTemp)$

properties	M=1	M=2	M=3	M=4	M=5
not deadlock	00:03	00:40	01:03	02:35	02:39
range of tempFlow	00:02	00:26	00:42	01:42	01:45
effect of rules	00:02	00:26	00:41	01:41	01:45
$\neg inv_w$->inv_w	00:03	00:37	00:54	02:14	02:33
goal reached in time	00:03	00:33	00:54	02:12	02:15
stable for some time	00:03	00:36	00:54	02:03	02:29

Table 9.6: Verification Times for a Fixed Amount of Deterministic Changes [min:sec]

properties	M=1	M=2	M=3	M=4	M=5
not deadlock	00:07	01:39	12:36	17:10	44:31
range of tempFlow	00:04	01:04	08:19	11:17	29:51
effect of rules	00:04	01:04	08:18	11:17	29:41
$\neg inv_w$->inv_w	00:07	01:32	11:20	15:04	41:26
goal reached in time	00:06	01:23	10:39	14:40	38:05
stable for some time	00:06	01:28	10:41	14:15	40:01

Table 9.7: Verification Times for a Fixed Amount of Non-Deterministic Changes [min:sec]

- The temperature should be restored within at most 10.000 seconds:
  $A[](obs.invalid \ \texttt{imply} \ c + d \leq 10.000)$

To verify the above properties and to additionally show that all rules are executed successfully and the system is always eventually in a stable state for a while, we have generated the observer automata and queries as described in Section 8.3.2 and have used the UPPAAL Model Checker to verify the properties.

Experimental Results To evaluate the scalability of our verification approach, we compare verification times in different environment settings. We have examined the adaptation behaviour in an environment with a finite amount of MAX (abbreviated to M in Tables 9.6 and 9.7) changes between three reference heating lines: 1. $-1.2x + 44$, 2. $-1.6x + 50$, 3. $-0.4x + 38$, and have comparde deterministic vs. non-deterministic changes. We have modeled corresponding automata that change the environment parameters (gradient and offset) after a fixed time (in our setting every three hours).

All verification experiments were carried out on a 64 bit Linux system with an Intel Core i7-4770 with 3.4 GHz and 32 GB RAM running Ubuntu 12.4 and averaged over 3 runs.

Table 9.6 shows the verification times in an environment with a fixed amount of deterministic changes between the reference heating lines. In a deterministically changing environment all properties could be verified very quickly. The number of visited states ranges from about $250,000$ $(MAX = 1)$ consuming 32 MB of RAM to about $11,000,000$ $(MAX = 5)$ consuming 874 MB of RAM. Table 9.7 shows the results in an environment

with a fixed amount of non-deterministic changes between the reference heating lines. The number of visited states ranges from about $500,000$ $(MAX = 1)$ consuming 50 MB of RAM to about $167,000,000$ $(MAX = 5)$ consuming 13.6 GB of RAM. The results show that with a non-deterministically changing environment, verification times are considerably higher, but still less than one hour in a setting where the environment changes five times.

Summary In this section, we have shown our experimental results for our smart temperature control system. We have successfully detected and classified introduced deviations in the effect of adaptation rules with our rule accuracy evaluation. Our observation-based learning has extracted correct context conditions that enable the introduction of precise and context-dependent effect expectations. Furthermore, we have shown that our simulation-based learning can find optimal parameters very quickly. The most time consuming part of the underlying genetic algorithm is the simulation of candidate solutions to evaluate their fitness. We have shown that this can be optimized by using the fast simulation capabilities of the SystemC simulation environment. We have performed formal verification of a smaller version of our smart temperature control system and compared runtimes for different environment settings. We were able to verify all properties in all scenarios. Due to the state space explosion problem of model checkers the verification times and memory consumption increase exponentially in the non-deterministic environments. For large and complex system and environment models, abstraction techniques and modular verification would help to reduce verification effort. In the following, we evaluate our goal model on our autonomous drone delivery case study.

9.2.2 Goal Model Evaluation with an Autonomous Drone Delivery System

In this section, we present our autonomous drone delivery case study, which we use to evaluate the expressiveness of our goal model and the suitability of our distance metric as a basis for autonomous decisions. We have presented this case study and the evaluation in [KGG18d].

Autonomous Drone Delivery System

Our autonomous drone delivery system consists of independent flying drones such as quadrocopters, which have to deliver goods from a starting position to several target addresses. Each drone has to fulfill the following requirements.
The main high-level requirements of our drone delivery service are, on the one hand, to maximize the satisfaction of customers and, on the other hand, to minimize the operation costs. Thus, we have two business requirements as shown in Figure 9.2a. As our drones

R1 The delivery times experienced by customers should be as low as possible.
R1a The length of the route a drone has to fly to deliver a good should be as short as possible.
R1b The velocity of the drone should be as high as safely possible to decrease delivery times.
R2 The operation costs, mainly influenced by the fuel consumption of drones, should be minimized.
R2a Drones should use the electricity-mode (e-mode) whenever the battery level is sufficiently high.
R2b To reduce fuel consumption in fuel-mode, the length of the route a drone has to fly to deliver a good should be as short as possible.
R2c To reduce fuel consumption in fuel-mode, the velocity of the drone should be reduced whenever this is possible.

(a) Business Requirements

R3 To avoid hazards with planes, the allowed height near to an airport is between 10 and 50 m.
R4 In cities the flight height is restricted to a corridor between 10 and 100 m.
R5 At the country side, height restrictions are relaxed to be between 5 and 200 m.
R6 The drones are allowed to fly with a velocity of 80 km/h at maximum, unless the drone has to perform a safety manoeuvre to avoid drone collisions.
R7 To avoid collisions between drones, the drones have to keep a safety distance of 10 m to all neighbouring drones.
R8 The noise of the drones is regulated during nighttime. Then, the volume of flying drones is not allowed to exceed 60 dB.

(b) Safety Requirements

Figure 9.2: Requirements for the Drones of our Delivery System

are part of the overall air traffic, they additionally have to adhere to the safety regulations as shown in Figure 9.2b.

This example was created by ourselves to illustrate our concepts. Thus, requirements were inspired by real drone regulations, but do not necessarily capture real-world drone regulations.

Safety is the most important aspect of activities that affect the air traffic. Thus, the more severe safety regulations are violated, the less important the business goals become.

Goal Model

We have build a goal model that reflects all these requirements. In Figure 9.3, we present the resulting goal model for a single drone. For simplicity, we set all tolerance values to 0. If they are greater than 0, they only reduce the observed distances. We use a linear precedence to model that the importance of business goals decreases the more severe safety regulations are violated. To capture that it is allowed to exceed the maximum velocity to avoid collision between drones (R6), we use "$0/1$"-precedence. This means that only if collisions can be excluded, i.e. the distance of the goal *Collision Avoidance* is

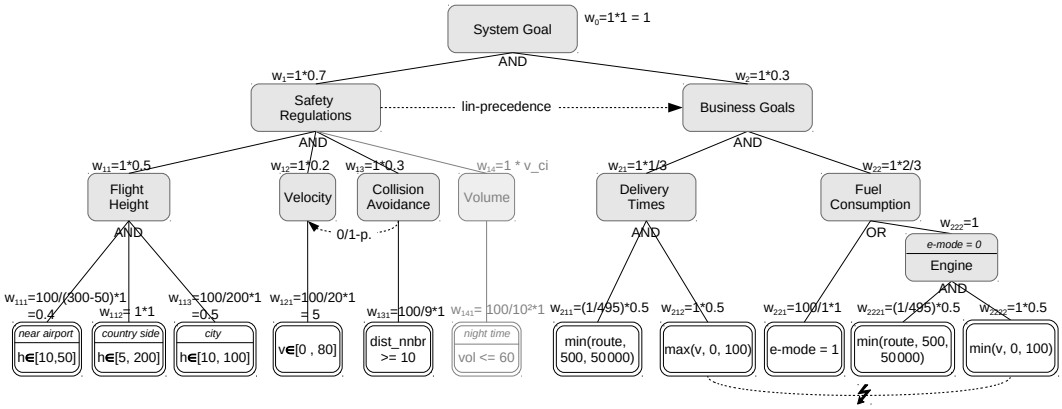


Figure 9.3: Goal Model of an Air Drone Delivery System

0, the velocity goal is active. As a consequence, in dangerous situations, speed regulations can temporarily be ignored.

Our requirements include one conflict: Requirements R1b and R2c both require to optimize the velocity of the drone, but in different directions (maximize vs. minimize). To indicate that this conflict is on purpose (e.g. to enable multi-objective optimization), we add a conflict edge in our model.

In our drone scenario, we manually specify the following qualitative priorities of the subgoals.

- *Safety* $>$ *Business*
- *Flight Height* $>$ *Collision Avoidance* $>$ *Velocity*
- *Delivery Times* $>$ *Fuel Consumption*
- *Volume*: context-dependent (most severe in the city where many people live and less severe near the airport)

For the remaining leaf goals in AND-aggregations, we assume an equal importance. This prioritization is captured in our weights in Figure 9.3. For the *Volume* subgoal, we use the context-dependent importance factor v_ci to describe that different importance values apply in different situations. We define v_ci to be 0.1 near the airport, 0.2 in the countryside, and 0.4 in the city.

Scenario-Based Evaluation of the Distance Calculation

In this section, we perform an evaluation on the expressiveness of our distance metric w.r.t. runtime changes in system and environment parameters and their influence on the system goals. We illustrate the development of the calculated distance in dynamic example scenarios and sketch how this enables to autonomously choose appropriate actions if the calculated distance significantly increases. Moreover, we illustrate the impact of goal changes on the goal model and the calculated distance.

Impact of Parameter Changes on the Distance Calculation In the runtime evaluation of our goal model, the following aspects are required to be visible in the calculated distance: a) The system and/or environment parameters change. b) A guard becomes satisfied so that the respective goal "suddenly" contributes to the distance of the parent goal. c) A goal is satisfied that has a precedence dependency to another goal.
To enable the observation whether those aspects are captured in our calculated distance, we have created a scenario, which contains all three aspects.

In Figure 9.4, we depict the calculated distance over time for our scenario. Here, we assume that the volume-related goals as depicted in Figure 9.3 are initially not part of the goal model. The scale of the parameters velocity and height is given by the left y-axis, while the scale of the system and safety goal evaluation is given by the right y-axis.

Left to the vertical bar, the scenario starts with a drone flying at the speed of 80 km/h, a height of 200 meters, in the countryside (not depicted in the Figure), no other drones nearby, and e-mode off. At time point 12, the drone enters the city, which deactivates the current height goal (*guard not satisfied anymore*) and activates the one for the city (*guard satisfaction*). This leads to an increase of the safety goal distance, because the enforced flight height in the city is 100 meters. At time point 14, the system reacts by descending to the height of 90 meters (*parameter changes*). Thus, the distances of the system and safety goals decrease again.

Right to the vertical bar, the situation that another drone approaches is depicted. From time point 30 to time point 33, a drone comes close up to 7 meters. At time point 32, the system reacts by increasing the speed, which reduces the overall distance, because the velocity goal is ignored as long as collision avoidance cannot be guaranteed (*"0/1"-precedence*). At time point 34, the distance to the drone is over 10 meters again. However, as the drone keeps on accelerating, the distance now increases, because the velocity goal is active and violated. At time point 40, the drone reduces its speed again down to 40. At time point 42.6, the velocity goal is satisfied again. From that time on, only the business goals influence the overall system goal. Because of its *conflicting goals* that aim at minimizing and maximizing the velocity, we see a curve, whose local minimum is at time point 45.7 with a speed of 57.25.

In Figure 9.5, the evolution of the involved distances is depicted for these conflicting goals in detail. For illustration, from time point 5 on, we decrease the velocity of the drone from 80 km/h to 10 km/h. While the fuel consumption distance decreases because less fuel is consumed, the delivery times distance increases, because it takes longer until the good is delivered. As above, the optimal velocity is 57.25 km/h, which is marked with a cross on the system goal curve.

In the following, we consider the case where not only parameters change at runtime but also the goal model itself.

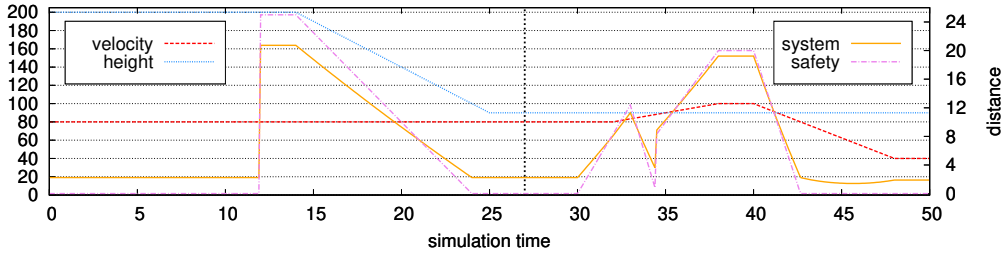


Figure 9.4: Scenario for Parameter Changes

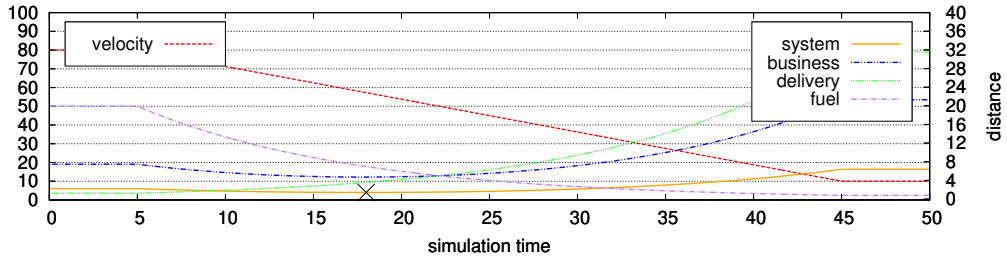


Figure 9.5: Scenario for Exploiting Conflicts

Impact of Local Goal Changes on the Calculated Distance To evaluate the modularity of our goal model and its distance calculation, we illustrate possible (local) goal changes within our drone example and illustrate its impact on the calculated distance.

Tolerances capture a flexibility when it comes to the violation of a subgoal. Tolerances can be changed in our model without the need for further adjustments. The effect of changed tolerances is a shift in the distance calculation. For example, if the tolerance is 0, the "$v \in [0, 80]$" leaf goal in our example has a calculated distance of 0 for $v = 80$, of 5 for $v = 85$, and of 10 for $v = 90$. If the tolerance is 5, the calculated distance is 0 for $v = 80$, 0 for $v = 85$, and 5 for $v = 90$. Thus, violations are considered less severe and higher-level goals are informed "later" in case of violations.

Changes of the **local distance function**, such as changing the ranges of an interval goal or changing the way child distances are combined in a parent goal, require an adjustment of the normalization factors to reobtain distance values between 0 and 100. As example, consider the case where the allowed velocity range is changed from $v \in [0, 80]$ to $v \in [0, 60]$. Then, the maximal possible deviation changes from 20 to 40 (given the maximal speed of 100 km/h for a drone) and the normalization factor needs to be changed to $100/40 = 2.5$. This adjustment of the normalization can be done automatically if the maximal deviation w.r.t. the new local distance function is given.

Precedences and guards of goals can easily be changed at runtime. For example, it may be allowed to additionally exceed the flight height restrictions if collisions between drones need to be avoided. To capture this in the goal model, a new "0/1"-precedence from the collision avoidance goal to the flight height goal has to be included. Another example is that the countryside height regulations extend to outer city parts, whereas city

flight height regulations apply to inner city areas only. These changes can be achieved by changing the guards of the flight height leaf goals accordingly. Here, no further adjustments are necessary.

The change of **weights**, i.e. normalization or importance, has to be done with caution. Depending on the local distance function of a parent goal, a changed weight of some subgoal may result in an overall contribution of all subgoals of the same parent that deviates from 100%. However, our local distance function of AND-decomposition automatically normalizes the overall contribution of all active children to 100%. As an example, we illustrate a scenario in Figure 9.6, in which we assume that the drone is in e-mode and that the drone needs to increase the speed to deliver the good as soon as possible. Thus, the speed limit shall be less important temporarily, which is modeled by changing the relative importance of business and safety goals. In our scenario, this is done at time point 12. The weight of the business goal is increased to 1.35 leading to a normalized importance of 0.4 for the safety goals and 0.6 for the business goals. With this, the overall distance drops and adaptations are postponed.

Our uniform distance function of parent nodes operates on multisets to enable that subgoals can easily be added or removed. For the **addition or removal of a subgoal**, it may be necessary to adjust the importance and normalization factors of local subgoals in the same branch, on the same level. This is, for example, the case if the distance function of the parent goal assumes an overall 100% contribution of its children subgoals. With our proposed design of the distance function of AND-decompositions, the importance factors of all active children goals are automatically normalized to an overall contribution of 100%. This facilitates the definition of importance factors in the presence of guarded goals and context-dependent importance factors as only active goals contribute to an overall of 100%. Thus, the guarded volume-related goal with context-dependent importance factors can be added without further ado. To show the influence of adding a subgoal on the distance value, we illustrate the distance evolution in the case of a runtime addition of the volume goal in Figure 9.7. For this scenario, we assume that it is night time, that the drone generates noise of 80 dB, and e-mode is deactivated. At time point 5, the additional safety volume goal as shown in Figure 9.3 is added. This leads to an increase of the overall safety goals. At time point 8, we assume that the system reacts by switching to e-mode, which immediately lets the business goal distance drop. The linear reduction of the volume to assumed 50 dB in e-mode itself takes two seconds. At the same time, the safety goal distance drops until a volume of 60 dB is reached.

Summary With our scenario-based evaluation, we have illustrated how the hierarchical and modular structure and the efficient quantitative evaluation of our goal model enable us to precisely analyze the impact of various sources of change at runtime. We have shown that all introduced changes of system and environment parameters, guard satisfaction, and precedences are reflected in the calculated distance. Furthermore, we have shown that

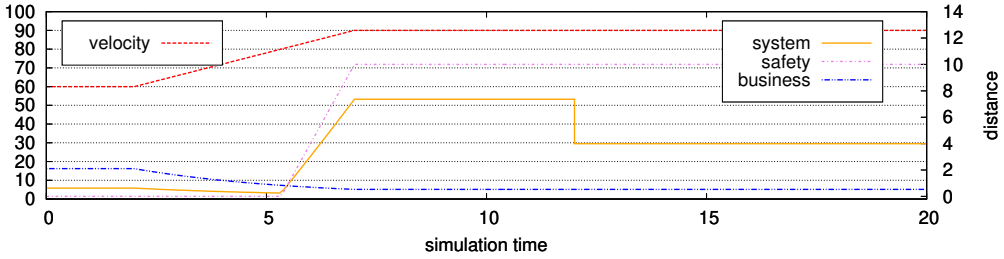


Figure 9.6: Scenario for Change of Weights

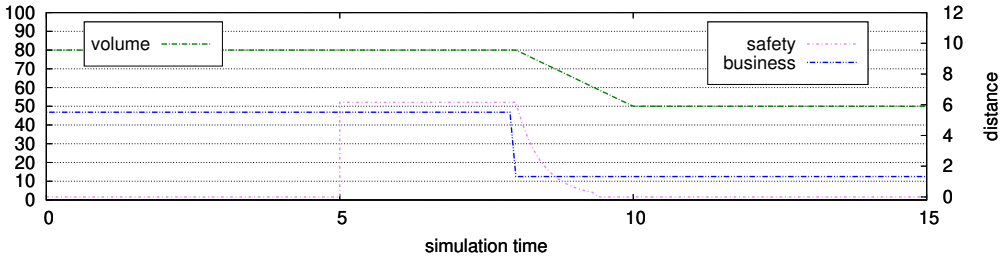


Figure 9.7: Scenario for Additional Goal

our distance calculation enables the detection of an optimal solution w.r.t. individual goal importances for conflicting goals. Thus, it is well-suited as a basis for our distance-based analysis and for comparing the expected result of different adaptation actions during planning. In addition, we have examined necessary adjustments on the goal model after different local goal changes and the impact of those changes on the calculated distance. We have shown that our model supports runtime changes of system goals, as in most cases only local changes to the weights of the changed goal itself and, in few cases, of additionally local surrounding subgoals are necessary.
In the following, we evaluate the scalability of our genetic algorithm that is used for simulation-based rule learning.

9.2.3 Scalability of Simulation-Based Adaptation Rule Learning for a Self-Organizing Production System

In this section, we present our self-organizing production system, which we use to evaluate the scalability of our genetic algorithm. It is parameterized on the number of involved robots in the production cell. Thus, this case study enables us to compare runtimes of our genetic algorithm for an increasing number of robots. We have published this case study and its evaluation in [KGG18b].

Self-Organizing Production Cell

Our self-organizing production cell consist of N industrial robots, a test bench and a switch that passes workpieces to robots. The general structure of a production cell is

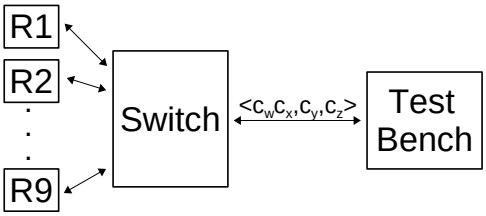


Figure 9.8: Autonomous Production Cell with N Robots

given in Figure 9.8. Robots have a currently installed capability and a list of available capabilities. To avoid damage on robots and workpieces, we introduce a wear limit (modeled by a number of steps that can be performed with the tool) for each tool and set capabilities unavailable if the corresponding tool reaches this limit. A workpiece is given by a list of capabilities that need to be processed in the given order. The robots collaborate to accomplish a workpiece. To this end, the switch passes each incoming workpiece to the next available robot with the required capability. Processing the workpiece at a robot is abstractly modeled by a passage of time that depends on the processing speed for its current capability. The test bench produces a stream of different workpieces that have to be accomplished by the robots and receives accomplished workpieces. The stream has some recurring pattern to enable learning for observed input scenarios. We have implemented this case study in SystemC.

Self-adaptation is realized by reorganizing current capabilities to agents at runtime if capabilities of agents become unavailable. To this end, we assume that reorganization is costly because it can only be performed during a short production stop. Thus, we only adapt the production cell if a capability is not available among the current capabilities of all robots.

Evaluation Scenario To illustrate learning of adaptation rules in this case study, we use an example instantiation of the production cell with *N=9 Robots*, *3 different capabilities* and the following start configuration of robots (3 robots for each capability):

Current Capabilities:
$[(R1,0),(R2,1),(R3,2),(R4,0),(R5,1),(R6,2),(R7,0),(R8,1),(R9,2)]$

Steps Left (wear limit):

Robot-ID	Capability 0	Capability 1	Capability 2
R1	**40**	35	20
R2	15	**40**	15
R3	25	30	**45**
R4	**40**	35	20
R5	15	**40**	15
R6	25	30	**45**
R7	**40**	35	20
R8	15	**40**	15
R9	25	30	**45**

Now, consider the following situation, where all available steps for capability 0 are exhausted. In this case, no robot with capability 0 is available and thus, reorganization has to take place.

Steps Left (wear limit):

Robot-ID	Capability 0	Capability 1	Capability 2
R1	**0**	35	20
R2	15	**0**	15
R3	25	30	**0**
R4	**0**	35	20
R5	15	**40**	15
R6	25	30	**41**
R7	**0**	35	20
R8	15	**7**	15
R9	25	30	**45**

Rule Learning with our Genetic Algorithm

To learn a suitable adaptation rule for this concrete scenario, we have used the SystemC model of the production cell as runtime model and applied our genetic algorithm (with 20 iterations, 20 generated children per iteration, 4 surviving individuals per iteration) to find an optimal solution with respect to maximizing the expected amount of workpieces that can be processed before the next reorganization. To reduce the amount of changed robots during reorganization, we added the number of unchanged robots to the fitness value, and weighted the amount of packages with 0.7 and the amount of robots with 0.3.

A resulting configuration is shown below. It proposes to change all robots as this results in an increase of finished workpieces that is higher than the punishment for changing all robots.

Current Capabilities:
$[(R1,1),(R2,2),(R3,0),(R4,2),(R5,1),(R6,0),(R7,1),(R8,0),(R9,0)]$

Steps Left (wear limit):

Robot-ID	Capability 0	Capability 1	Capability 2
R1	0	***35***	20
R2	15	0	***15***
R3	***25***	30	0
R4	0	35	***20***
R5	***15***	40	15
R6	***25***	30	41
R7	0	***35***	20
R8	***15***	7	15
R9	***25***	30	45

From this, we can extract an initial adaptation rule that consists of a guard describing the initial configuration together with the history of received workpieces, the commands to enable the capability changes for all robots and an effect that basically states that all capabilities are available again (immediately). Additionally, we could add the expected amount of accomplished workpieces before the next reorganization to enable a comparison of different applicable adaptation rules.

To evaluate the runtimes of our genetic algorithm, we have performed learning within our scenario for systems with different amounts of robots. The experiments were carried out on a 64 bit Linux system with an Intel Core i7-3520M CPU with 2.9 GHz and 16 GB of RAM. The results are shown in Table 9.8. The runtimes only moderately increase with increasing numbers of robots. This is due to the heuristic nature of genetic algorithms together with the limited number of iterations and children per iteration, which quickly leads to nearly optimal results. The different fitness values are due to the fact that the production cell is able to accomplish more workpieces if it contains more robots.

N = 3	N = 4	N = 6	N = 9
0.2s (4, 6.30)	1.1s (108, 10.10)	1.7s (253, 13.90)	3.2s (338, 24.10)

Table 9.8: Comparison of Genetic Algorithm for Production Cell Variants: runtime (#simulations, achieved fitness)

9.3 Summary

In this chapter, we have evaluated the applicability of our approach with our smart temperature control system. We have shown that our rule accuracy evaluation successfully detects introduced context-dependent deviations of the adaptation effect and our observation-based learning correctly classifies the context-dependencies. Thus, we can correct the effect expectations with rules that capture the identified context-dependencies. Furthermore, we have shown that our genetic algorithm learns suitable control parameters for a given situation very quickly and that our simulation-based learning can extract adaptation rules from the simulation traces of our runtime models. We have successfully verified safety and adaptation properties in different environments with our formal verification. In addition, we have evaluated the effectiveness and modularity of our distance evaluation on our goal model with an autonomous drone delivery system. Our evaluation shows that all kinds of runtime changes of the system goals are easily possible and require minor and local adjustments on the normalization factor that is part of our weight definition, only. Our evaluation has also confirmed that our distance metric is well-suited for detecting dynamic changes in system and parameter values and for evaluating and comparing different adaptation possibilities. Afterwards, we have evaluated the scalability of our genetic algorithm (GA) on a parameterized production system that allows for comparing runtimes of systems with different amounts of robots. Our results show that runtimes only increase moderately with an increasing amount of robots and that the GA has always lead to a near optimal solution quickly. Thus, we conclude that our genetic algorithm scales well.

10 Conclusion

In this chapter, we summarize the results of this thesis and show how our approach meets our objectives as presented in the introduction. Afterwards, we give an outlook on future work.

10.1 Results

We have presented a framework that enables the design and runtime evolution of safe, intelligent and explainable self-adaptive systems. We have closed a gap by providing an integrated approach that is capable of safely evolving the adaptation logics of rule-based self-adaptive systems in order to handle dynamic changes in the system, its environment and goals. Before, other approaches had addressed the design of self-adaptive systems, or had presented solutions for single aspects of our research question, but there was no integrated solution for runtime learning and verification in the presence of such dynamic changes. Furthermore, most other approaches do not consider explainability of runtime decisions.

Our main contributions are a rule- and distance-based adaptation process, a quantitative and context-dependent goal model, an approach for the resource-efficient runtime evolution of adaptation logics, and a continuous verification methodology. Furthermore, our rule format and our explanation basis, which consists of collected runtime knowledge, enable the generation of detailed explanations about adaptation and learning decisions at runtime.

We have evaluated our approach on three case studies from different domains, namely a smart-temperature control system, an autonomous drone delivery system and a self-organizing production system. We have proven the applicability of our approach by successfully applying it to our smart temperature control system. In addition, we have demonstrated the strength of our quantitative distance evaluation in scenarios with various runtime changes of parameters and of the goal model itself. We have performed these experiments on our drone delivery system because its requirements contain all features of our goal model. Furthermore, we have shown that the runtimes of our our genetic algorithm increase only moderately with an increasing number of adaptation options, i.e. number of robots within our self-organizing production system.

We summarize our key ideas for each main contribution:

Our rule- and distance-based adaptation process is based on our novel notion of timed adaptation rules. We have used a comprehensible condition-action-effect structure for our timed adaptation rules to support the explainability of autonomous decisions. The key idea of these rules is to explicitly encode timed effect expectations on observable environment and system parameters within the adaptation rules. We have separated the effect expectations and their contribution to the system goals to achieve modularity and reusability in the context of dynamic goal changes. We have evaluated the expected contribution of adaptation rules with a novel notion of distance between a system state and the goals.

Our quantitative and context-dependent goal model encodes context-dependent goals, e.g. setpoints or optimization objectives, and their dependencies. We have combined essential modeling elements from existing standard goal modeling languages, such as the Goal-oriented Requirements Language (GRL) [Int12], i* [Yu97] or KAOS [vLL00], and have provided additional modeling elements to describe context-dependent goal relations and importances. To quantify the context-dependent achievement of goals during analysis and planning of adaptations, we have provided an efficient and modular distance evaluation algorithm.

Our novel approach for the resource-efficient runtime evolution of adaptation logics combines a continuous accuracy evaluation, an observation-based optimization of adaptation rules and a stochastic search-based learning of new comprehensible rules. Within our observation-based learning, we have used classification rule learning to extract context conditions that cause recurring deviations from the expected effect. Based on these conditions, we have specified modified adaptation rules with context-specific effect expectations. To enable online learning of new adaptation rules, we have applied a genetic algorithm that evaluates found solutions on a model simulation to avoid the costs and risks of active exploration in the real system. Our rule learning includes stepwise rule generalization process to increase the applicability of new rules. We have stored structured analysis and learning results to provide processed data for explaining autonomous rule learning.

Our continuous verification methodology consists of formal verification during the design process and in the runtime evolution process. In addition, we have showed how runtime monitoring of safety properties can be integrated in the monitoring process of the MAPE-K loop for self-adaptive systems. To enable formal verification, we have provided a formalization of our timed adaptation rules and our proposed rule- and distance-based adaptation process in timed automata.

10.2 Discussion

In the introduction, we have presented the objectives that should be fulfilled by a framework for safe, intelligent and explainable self-adaptive systems. In the following, we discuss how our solution matches these objectives.

10.2.1 Continuous learning of (timed) adaptation logics

The first criterion we discuss is the continuous learning of (timed) adaptation rules. Intelligent cyber-physical systems control complex processes in a physical environment showing behavior that may not have been considered at design time. Furthermore, they operate for a long time after deployment and may be subject to change during their lifespan, i.e. goals and system structure may change due to component failure or installation. One aim of our thesis is to design intelligent systems that are flexible enough to deal with such ever-changing operation contexts. Thus, they continuously have to evaluate whether their adaptation logic still fits to the environment, their system topology and capabilities, and their goals. If necessary, they have to learn suitable changes of their adaptation logic that enable them to cope with these run-time changes.

To detect dynamic changes, we have proposed a rule accuracy evaluation, a heartbeat approach, and an adaptation planning algorithm that is able to decide if it is possible to achieve the system goals with the current set of rules. Our rule accuracy evaluation assesses whether executed adaptations achieve their expected effect, and whether the encoded effect fits the environment. We have resolved detected mismatches with our observation-based learning. To detect changes in the system topology, we have integrated a heartbeat into the monitoring of the manged system and have demanded an explicit registration of new components. After detection, rules that rely on lost components are removed from the active rule set and rules for new components are learned. If changed goals cannot be achieved with the current set of adaptation rules, our planning process provokes learning of new rules.

Furthermore, we require the adaptation logic to include the latency of adaptation actions to account for time constraints and to enable proactive adaptation in time. Our novel notion of timed adaptation rules includes an expectation when the effect should be observable in the environment. We have used this expectation during adaptation planning to favor faster solutions in case of rules with the same improvement on our quality measure, i.e. the distance towards the goals. We have introduced an additional time tolerance to express that there is an acceptable delay for the effect observation. We have used the timing expectation (plus tolerance) to decide whether a rule execution led to the expected effect, and the tolerance to cope with the usual problem of missing phenomena due to discrete sampling rates.

10.2.2 Independence between adaptation logics and system goals

Our second criterion concerns the independence between adaptation logics and system goals. As goals may change at runtime, we require our solution to provide an independent encoding of the expected effect of adaptation rules and of the system goals. We thereby ensure that the adaptation logics is robust with respect to dynamic goal changes, i.e. adaptation rules should still be applicable after goal changes.

We have achieved this separation by splitting our encoding of the expected effect on environment and system parameters and the separate quantitative evaluation of the distance between the expected system state and the system goals. In addition, we have designed our distance evaluation in a modular way to ensure that goal changes require local recalculations of the distance only. Thus, we have further minimized the impact of goal changes.

10.2.3 Continuous analysis of safety properties

Our framework should be applicable for safety-critical cyber-physical systems. Thus, our third criterion is the continuous analysis of safety properties to ensure safety of the autonomous decisions.

With our formalization in UPPAAL timed automata, we have enabled formal verification via model checking. It is thereby possible to formally verify the initial adaptation logic at design time and the evolved adaptation logic at runtime. To achieve continuous verification, we have defined a verification process that is deeply integrated into our adaptation and evolution processes. A safety monitoring detects violations of properties that are encoded in the system goals or in the analysis process and initiates countermeasures, such as applying suitable adaptations or switching into a safe operation mode. During learning of adaptation rules, a validation of the effect expectations is performed on the runtime models. Before a change in the adaptation rules is applied on the system, safety is ensured by performing a comprehensive system verification.

10.2.4 Explainability of autonomous decisions

Our fourth criterion is to ensure the explainability of autonomous decisions. Intelligent cyber-physical systems make autonomous decisions in safety-critical domains, such as automotive, industrial production, and medicine. As we hand over control to those systems, we need to trust them. We have argued that providing comprehensible explanations for autonomous decisions is crucial to obtain trust.

To achieve this, we have based our adaptation and learning decisions on comprehensible models: The condition-action-effect structure of our adaptation rules resembles the way humans specify actions and their effect. As example, consider the following sentence in natural language: "If it is Sunday *condition* and I sleep in *action*, I will feel rested *effect*." We use human-readable runtime models in SystemC and UPPAAL timed

automata for our simulation-based learning and verification. We thereby ensure that results in the model can be easily retraced to the system. In addition, to supported the traceability of these decisions, we have stored structured and processed knowledge in an explanation base that can be queried for generating explanations at runtime or at inspection time.

10.2.5 Resource-efficiency

Intelligent cyber-physical systems (CPS) are connected embedded systems that control a physical environment and that are connected to the cyber-space. Cyber-space refers to remote computing power or remote system components that are connected via the internet. In embedded devices, resources, such as computation time, energy, and memory, are restricted. Although remote computation power, i.e. servers, may be accessible, communication is expensive in terms of energy and delays. Thus, it is desirable to enable local decision-making, at least for frequent and time-critical decisions. We therefore require our approach to be efficient in terms of time and computational overhead to be applicable in real-time and embedded devices.

With our rule-based adaptation and modular distance calculation, we have provided a fast self-adaptation with a low computational overhead. In our evolution layer, we must perform resource-intensive tasks, such as learning and verification. However, we have reduced the resource-consumption with several means: We have reduced the learning overhead by applying our fast observation-based learning where possible, and by increasing the applicability of learned rules with our rule generalization. For verification, we differentiate between simulation-based rule effect validation and a comprehensive formal system verification. The former is less resource-consuming, and is used during rule generalization. The latter is based on model checking, which is known to be resource-demanding. However, verification is only performed after learning, which we assume to be necessary infrequently only.
As we assume, that the system can be set into a safe operation mode at any time, learning and comprehensive verification can be outsourced to an external server with more computational power and memory.

To summarize our discussion, our solution fulfills our criteria and enables the design of intelligent systems that are flexible enough to deal with ever-changing operation contexts while ensuring the safety and explainability of their autonomous decisions. To the best of our knowledge, it is the only approach that provides an integrated solution to this crucial research topic.

10.3 Outlook

In our thesis, we have focused on the integration of different aspects, such as self-adaptation, learning, verification and explanations, into one framework that can be

instantiated for several domains. Our framework can be improved further and can serve as a basis for investigating future research directions. We first describe possible enhancements of our framework and discuss future research directions afterwards.

We have chosen a straight-forward adaptation planning strategy to focus on our evolution layer. Our framework can be extended by enhancing our timed adaptation rules with probabilities, complex dependencies between adaptation commands and time delays between commands. Our rule-based planning algorithm can be expanded to consider these new features. Another possibility is to integrate different planning strategies, such as online planning on model simulations or approaches for automatic controller synthesis. The latter could be achieved by using the UPPAAL extension UPPAAL TIGA.
The performance of our rule learning can be increased. One possibility is to further reduce the frequency of learning with heuristics. These heuristics describe when to delete inaccurate or inapplicable rules from the knowledge base, and when to store them for later re-evaluation and adjustment. A second option worth exploring, is the applicability and performance of different learning strategies as alternative for our genetic algorithm. Also the scalability of our formal verification can be improved. A promising approach is to apply modular verification, e.g. based on assume-guarantee contracts, and to statically analyze which components are influenced by newly learned rules. Thus, only affected system and environment components have to be verified again. This also enables the integration of externally verified adaptation rule sets of new components.

During our thesis, further research questions arose. We have assumed that the adaptation layer is based on a single MAPE-K loop. It appears interesting to investigate decentral intelligent self-adaptive systems, e.g. for smart grids or traffic control. A particular challenge is to distribute our evolution layer, which currently assumes certain knowledge of the overall system. Research in this direction can be based on results for different design patterns for MAPE-K feedback loops that describe realizations for hierarchical and distributed MAPE-K loops (see [WSG+13]).
Our evolution layer adapts the rule set of our adaptation layer. A similar meta adaptation worth to investigate is to adapt the behavior of single MAPE-phases, such as introducing a context-dependent monitoring rate, switching between different planning strategies or adjusting the analysis strategy. We have discussed several options for such adaptations in [KGG17]. In addition, it seems promising to follow this idea of meta adaptation and to investigate adaptations of the evolution layer, such as switching between different learning and verification strategies. This would enable a higher-order self-adaptation approach where adaptation can take place on arbitrary levels of abstraction.
With our explanation basis, we provide a basis for automatically generated explanations. However, explanations have to be adjusted to the recipients need to be valuable for users, engineers or other stakeholders. It is worth investigating which decisions and actions require an explanation and which explanation style should be used. Explanation styles may differ w.r.t. their efficiency and acceptance rate for different recipient groups. For

example, a user will need less technical explanations than an engineer, and a lawyer cares about different aspects than engineers. Moreover, different situations may require different kinds of explanations. For example, an explanation why a plane on autopilot needs a human pilot to take over must be understandable quickly to avoid a crash. However, if a crash has occurred, a detailed explanation is required to learn from this crash. As a first step into this fascinating research topic, we have presented our vision of self-explainable cyber-physical systems in [BGC+19]. Based on our experience from this thesis, we have proposed a reference framework for self-explainable CPS that has a similar structure to our framework for intelligent self-adaptive systems. The proposed framework is based on the MAPE-K feedback loop and includes a second layers to adapt the knowledge base of the first layer.

List of Figures

List of Listings and Algorithms

List of Tables

Bibliography

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[ADG10] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458, 2010.

[AGJ+14] Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. A reference architecture and roadmap for models@run.time systems. In *Models@run.time*, pages 1–18. Springer, 2014.

[ARS15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 13–23. IEEE, 2015.

[ASSV07] Rasmus Adler, Ina Schaefer, Tobias Schüle, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. In *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA*, pages 76–95. Springer, 2007.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, LNCS 3185, pages 200–236. Springer, 2004.

[BFT+14] Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Mosterman, Walter Cazzola, Fabio M. Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Akşit, Pär Emmanuelson, Huang Gang, Nikolaos Georgantas, and David Redlich. Mechanisms for leveraging models at runtime in self-adaptive software. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, volume 8378 of *Lecture Notes in Computer Science*, pages 19–46. Springer, 2014.

[BGC+19] M. Blumreiter, J. Greenyer, F. J. Chiyah Garcia, V. Klös, M. Schwammberger, C. Sommer, A. Vogelsang, and A. Wortmann. Towards self-explainable cyber-physical systems. In *22nd International Conference on*

Model Driven Engineering Languages and Systems Companion (MODELS-C), The 14th International Workshop on Models@run.time, pages 543–548. IEEE, Sep. 2019.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[BSG+09] 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*, pages 48–70. Springer, 2009.

[BY04] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer–Verlag, 2004.

[BZL06] Josh Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121, 2006.

[CdL12] Javier Cámara and Rogério de Lemos. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 53–62. IEEE Press, 2012.

[CG12] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, 2012.

[CGK+11] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic QoS management and optimization in service-based systems. *Transactions on Software Engineering*, 37(3):387–409, 2011.

[CGLG15] Zack Coker, David Garlan, and Claire Le Goues. SASS: Self-adaptation using stochastic search. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 168–174, Piscataway, NJ, USA, 2015. IEEE Press.

[CGRL+18a] Francisco J. Chiyah Garcia, David A. Robb, Atanas Laskov, Xingkun Liu, Pedro Patron, and Helen Hastie. Explainable autonomy: A study of explanation styles for building clear mental models. In *11th International Natural Language Generation Conference (INLG)*, pages 99–108. ACM, 2018.

[CGRL+18b] Francisco J. Chiyah Garcia, David A. Robb, X. Liu, Atanas Laskov, Patron Patron, and Helen Hastie. Explain yourself: A natural language interface for

scrutable autonomous robots. In *Explainable Robotic Systems Workshop (HRI)*, 2018.

[CGSP15] Javier Cámara, David Garlan, Bradley Schmerl, and Ashutosh Pandey. Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In *30th Annual ACM Symposium on Applied Computing*, pages 428–435. ACM, 2015.

[CLGS16] Javier Cámara, Antonia Lopes, David Garlan, and Bradley Schmerl. Adaptation impact and environment models for architecture-based self-adaptive systems. *Science of Computer Programming*, 127:50–75, 2016.

[Coh95] William W Cohen. Fast effective rule induction. In *Machine learning proceedings 1995*, pages 115–123. Elsevier, 1995.

[CvL17] Antoine Cailliau and Axel van Lamsweerde. Runtime monitoring and resolution of probabilistic obstacles to system goals. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 1–11, 2017.

[Dil89] David L Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Conference on Computer Aided Verification*, pages 197–212. Springer, 1989.

[DLFG18] Rolf Drechsler, Christoph Lüth, Goerschwin Fey, and Tim Güneysu. Towards self-explaining digital systems: A design methodology for the next generation. In *3rd International Verification and Security Workshop (IVSW)*, pages 1–6. IEEE, 2018.

[DLGM+13] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[dIIW15] Didac Gil de la Iglesia and Danny Weyns. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *TAAS*, 10(3):15, 2015.

[DLL+15] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[DM10] Scott A. DeLoach and Matthew Miller. A goal model for adaptive complex systems. *International Journal of Computational Intelligence: Theory and Practice*, 5(2), 2010.

[dMPCdS12] André de Matos Pedro, Paul Andrew Crocker, and Simão Melo de Sousa. Learning stochastic timed automata from sample executions. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 508–523. Springer, 2012.

[EEM13] Naeem Esfahani, Ahmed Elkhodary, and Salim Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *Software Engineering, IEEE Transactions on*, 39(11):1467–1493, 2013.

[EGMT09] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *31st International Conference on Software Engineering*, pages 111–121. IEEE Computer Society, 2009.

[EHCR18] Upol Ehsan, Brent Harrison, Larry Chan, and Mark O Riedl. Rationalization: A neural machine translation approach to generating natural language explanations. In *AAAI/ACM Conference on AI, Ethics, and Society*, pages 81–87, 2018.

[ES+03] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[Fan18] Xiuyi Fan. On generating explainable plans with assumption-based argumentation. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 344–361. Springer, 2018.

[FGKV19] Erik M. Fredericks, Ilias Gerostathopoulos, Christian Krupitzer, and Thomas Vogel. Planning as optimization: Dynamically discovering optimal configurations for runtime situations. In *13th International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2019, Umea, Sweden, June 16-20, 2019*, pages 1–10, 2019.

[FGL12] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrač. *Foundations of rule learning*. Springer Science & Business Media, 2012.

[FRLB15] A. Frömmgen, R. Rehner, M. Lehn, and A. Buchmann. Fossa: Learning ECA rules for adaptive distributed systems. In *International Conference on Autonomic Computing*, pages 207–210, July 2015.

[FTG16] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *Transactions on Software Engineering*, 42(1):75–99, 2016.

[Fut15] Future of Life Institute. An Open Letter: Research Priorities for Robust and Beneficial Artificial Intelligence, 2015.

[GBH+16] Ilias Gerostathopoulos, Tomás Bures, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, Frantisek Plasil, and Noël Plouzeau. Self-adaptation in software-intensive cyber-physical systems: From system goals to architecture configurations. *Journal of Systems and Software*, 122:378–397, 2016.

[GCB14] Simos Gerasimou, Radu Calinescu, and Alec Banks. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 115–124. ACM, 2014.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[Ghe10] Carlo Ghezzi. Adaptive software needs continuous verification. In *8th International Conference on Software Engineering and Formal Methods (SEFM 2010)*, pages 3–4. IEEE, 2010.

[GKB15] Thomas Göthel, Verena Klös, and Björn Bartels. Modular design and verification of distributed adaptive real-time systems based on refinements and abstractions. *EAI Endorsed Transactions on Self-Adaptive Systems*, 15(1), 2015.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GVD+17] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, Nelly Bencomo, Kurt Geihs, Samuel Kounev, and Kirstie L. Bellman. State of the art in architectures for self-aware computing systems. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware Computing Systems*, pages 237–275. Springer International Publishing, Cham, 2017.

[H+92] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 1992. 1st edition: 1975.

[HFG08] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model Checking SystemC Designs Using Timed Automata. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 131–136. ACM press, 2008.

[HKP12] Jiawei Han, Micheline Kamber, and Jian Pei. Classification: Basic concepts. In Jiawei Han, Micheline Kamber, and Jian Pei, editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 327 – 391. Morgan Kaufmann, Boston, third edition edition, 2012.

[HPG15] Paula Herber, Marcel Pockrandt, and Sabine Glesner. STATE – a SystemC to Timed Automata Transformation Engine. In *International Conference on Embedded Software and Systems (ICESS)*. IEEE Computer Society, 2015.

[HTKS12] Shinpei Hayashi, Daisuke Tanabe, Haruhiko Kaiya, and Motoshi Saeki. Impact analysis on an attributed goal graph. *IEICE Transactions*, 95-D(4):1012–1020, 2012.

[IBM04] IBM Corp. *An architectural blueprint for autonomic computing*. IBM Corp., USA, October 2004.

[IEE11] IEEE Standards Association. IEEE Std. 1666–2011, Open SystemC Language Reference Manual, 2011.

[Int12] International Telecommunication Union: Recommendation Z.151 (10/12). User requirements notation (URN) - language definition, 2012.

[IW14] M. Usman Iftikhar and Danny Weyns. Activforms: Active formal models for self-adaptation. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 125–134, New York, NY, USA, 2014. ACM.

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KCW+18] Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le Goues. Managing uncertainty in self-adaptive systems with plan reuse and stochastic search. In *13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 40–50, New York, NY, USA, 2018. ACM.

[KGG15] Verena Klös, Thomas Göthel, and Sabine Glesner. Adaptive knowledge bases in self-adaptive system design. In *41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 472 – 478. IEEE, 2015.

[KGG16] Verena Klös, Thomas Göthel, and Sabine Glesner. Formal models for analysing dynamic adaptation behaviour in real-time systems. In *1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 106–111. IEEE, 2016.

[KGG17] Verena Klös, Thomas Göthel, and Sabine Glesner. Parameterisation and optimisation patterns for MAPE-K feedback loops. In *2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 13–18, 2017.

[KGG18a] Verena Klös, Thomas Göthel, and Sabine Glesner. Be prepared: Learning environment profiles for proactive rule-based production planning. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 89–96. IEEE, 2018.

[KGG18b] Verena Klös, Thomas Göthel, and Sabine Glesner. Comprehensible and dependable self-learning self-adaptive systems. *Journal of Systems Architecture*, 85-86:28 – 42, 2018.

[KGG18c] Verena Klös, Thomas Göthel, and Sabine Glesner. Comprehensible decisions in complex self-adaptive systems. In *Software Engineering 2018, Fachtagung des GI-Fachbereichs Softwaretechnik*, pages 215–216. Gesellschaft für Informatik, 2018.

[KGG18d] Verena Klös, Thomas Göthel, and Sabine Glesner. Runtime management and quantitative evaluation of changing system goals in complex autonomous systems. *Journal of Systems and Software*, 144:314–327, 2018.

[KGLG17] Verena Klös, Thomas Göthel, Adrian Lohr, and Sabine Glesner. Runtime management and quantitative evaluation of changing system goals. In *43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, Vienna, Austria, August 30 - Sept. 1, 2017*, pages 226–233, 2017.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: An architectural challenge. In *Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[KP09] Dongsun Kim and Sooyong Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 76–85. IEEE, 2009.

[Loh16] Adrian Lohr. Modellierung von Systemzielen für selbst-adaptive Systeme. 2016. Bachelor Thesis.

[LSYM14] Yanji Liu, Yukun Su, Xinshang Yin, and Gunter Mussbacher. Combined propagation-based reasoning with goal and feature models. In *4th International Model-Driven Requirements Engineering Workshop, MoDRE 2014, 25 August, 2014, Karlskrona, Sweden*, pages 27–36, 2014.

[MCGS15] Gabriel A Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *10th Joint Meeting on Foundations of Software Engineering*, pages 1–12. ACM, 2015.

[MJJ+17] Anders L Madsen, Nicolaj Søndberg Jeppesen, Frank Jensen, Mohamed S Sayed, Ulrich Moser, Luis Neto, Joao Reis, and Niels Lohse. Parameter learning algorithms for continuous model improvement using operational data. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 115–124. Springer, 2017.

[MNBB17] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, and Saddek Bensalem. Improved learning for stochastic timed models by state-merging algorithms. In *NASA Formal Methods Symposium*, pages 178–193. Springer, 2017.

[MSSU11] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. *Organic computing—A paradigm shift for complex systems*. Springer Science & Business Media, 2011.

[Nev18] Nathalie Nevejans. Open Letter to the European Commision Artificial intelligence and Robotics, 2018.

[NSSR13] Florian Nafz, Jan-Philipp Steghöfer, Hella Seebach, and Wolfgang Reif. Formal modeling and verification of self-* systems based on observer/controller-architectures. In *Assurances for Self-Adaptive Systems*, pages 80–111. Springer, 2013.

[PHKG13] Marcel Pockrandt, Paula Herber, Verena Klös, and Sabine Glesner. Model Checking Memory-Related Properties of Hardware/Software Co-Designs. In *Embedded Systems: Design, Analysis and Verification. International Embedded Systems Symposium (IESS) 2013, Paderborn*, 2013.

[PMCG16] Ashutosh Pandey, Gabriel A Moreno, Javier Cámara, and David Garlan. Hybrid planning for decision making in self-adaptive systems. In *10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 130–139. IEEE, 2016.

[PRA11] Alireza Pourshahid, Gregory Richards, and Daniel Amyot. Toward a goal-oriented, business intelligence decision-making framework. In *E-Technologies: Transformation in a Connected World - 5th International Conference, MCETECH 2011, Les Diablerets, Switzerland, January 23-26, 2011, Revised Selected Papers*, pages 100–115, 2011.

[PSRV16] Vittorio Perera, Sai P. Selveraj, Stephanie Rosenthal, and Manuela Veloso. Dynamic generation and refinement of robot verbalization. In *25th International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 212–218, New York, NY, USA, August 2016. IEEE.

[RCGL+18] David A. Robb, Francisco J. Chiyah Garcia, Atanas Laskov, Xingkun Liu, Pedro Patron, and Helen Hastie. Keep me in the loop: Increasing operator situation awareness through a conversational multimodal interface. In *20th*

ACM International Conference on Multimodal Interaction (ICMI), pages 384–392. ACM, 2018.

[RCR+18] Arthur Rodrigues, Ricardo Diniz Caldas, Genaína Nunes Rodrigues, Thomas Vogel, and Patrizio Pelliccione. A learning approach to enhance assurances for real-time self-adaptive systems. In *13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 206–216. IEEE, 2018.

[RMB+06] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Towards a generic observer/controller architecture for organic computing. *GI Jahrestagung (1)*, 93:112–119, 2006.

[RRL+13] Liliana Rosa, Luís E. T. Rodrigues, Antónia Lopes, Matti A. Hiltunen, and Richard D. Schlichting. Self-management of adaptable component-based applications. *IEEE Trans. Software Eng.*, 39(3):403–421, 2013.

[Rus18] Stuart Russell. Uncertainty in objectives, 2018. Invited talk at Conference on Uncertainty in Artificial Intelligence (UAI) 2018, Monterey, California, USA.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SCM+13] Daniel Sykes, Domenico Corapi, Jeff Magee, Jeff Kramer, Alessandra Russo, and Katsumi Inoue. Learning revised models for planning in adaptive systems. In *International Conference on Software Engineering (ICSE)*, pages 63–71. IEEE Press, 2013.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08)*, pages 1–8. ACM, 2008.

[SM17] Amir Molzam Sharifloo and Andreas Metzger. Mcaas: Model checking in the cloud for assurances of adaptive systems. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 137–153. Springer, 2017.

[SMQ+16] Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, and Klaus Pohl. Learning and evolution in dynamic software product lines. In *11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 158–164. IEEE, 2016.

[SR16] Kevin Styp-Rekowski. Generierung von Adaptationen mithilfe von Genetischen Algorithmen. 2016. Bachelor Thesis.

[SS13] Amir Molzam Sharifloo and Paola Spoletini. Lover: Light-weight formal verification of adaptive systems at run time. In *Formal Aspects of Component Software*, volume 7684 of *Lecture Notes in Computer Science*, pages 170–187. Springer Berlin Heidelberg, 2013.

[SSDD17] Estefanía Serral, Paolo Sernani, Aldo Franco Dragoni, and Fabiano Dalpiaz. Contextual requirements prioritization and its application to smart homes. In *Ambient Intelligence - 13th European Conference, AmI 2017, Malaga, Spain, April 26-28, 2017, Proceedings*, pages 94–109, 2017.

[SSG18] Roykrong Sukkerd, Reid Simmons, and David Garlan. Towards explainable multi-objective probabilistic planning. In *4th International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 19–25. ACM, 2018.

[SSP+17] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. A berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855*, 2017.

[SST06] Klaus Schneider, Tobias Schuele, and Mario Trapp. Verifying the adaptation behavior of embedded systems. In *International Workshop on Self-Adaptation and Self-Managing Systems*, pages 16–22. ACM, 2006.

[SWM17] Stepan Shevtsov, Danny Weyns, and Martina Maggio. Handling new and changing requirements with guarantees in self-adaptive systems using SimCA. In *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 12–23, 2017.

[TALL18] Martin Tappler, Bernhard K Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Learning timed automata via genetic programming. *arXiv preprint arXiv:1808.07744*, 2018.

[TPB+11] Sven Tomforde, Holger Prothmann, Jürgen Branke, Jörg Hähner, Moez Mnif, Christian Müller-Schloer, Urban Richter, and Hartmut Schmeck. *Observation and Control of Organic Systems*, pages 325–338. Springer, Basel, 2011.

[TVM+13] Gabriel Tamura, Norha Villegas, Hausi Müller, João Pedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, et al. Towards practical runtime verification and validation of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 108–132. Springer, 2013.

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, October 2000.

[VTM+13] Norha M Villegas, Gabriel Tamura, Hausi A Müller, Laurence Duchien, and Rubby Casallas. Dynamico: A reference model for governing control objectives and context relevance in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 265–293. Springer, 2013.

[WBC+17] Danny Weyns, Nelly Bencomo, Radu Calinescu, Javier Camara, Carlo Ghezzi, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jezequel, Sam Malek, et al. Perpetual assurances for self-adaptive systems. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 31–63. Springer, 2017.

[Wey17] Danny Weyns. Software engineering of self-adaptive systems: an organised tour and future challenges. *Chapter in Handbook of Software Engineering*, 2017.

[WFF19] Dustin Wüest, Farnaz Fotrousi, and Samuel Fricker. Combining monitoring and autonomous feedback requests to elicit actionable knowledge of system use. In Eric Knauss and Michael Goedicke, editors, *Requirements Engineering: Foundation for Software Quality*, pages 209–225, Cham, 2019. Springer International Publishing.

[WFHP16] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. Online appendix for the WEKA workbench. In *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016.

[WMA12] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(1):8, 2012.

[WSG+13] 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*, pages 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Yu97] E. S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

[ZGC09] Ji Zhang, Heather J Goldsby, and Betty HC Cheng. Modular verification of dynamically adaptive systems. In *8th International Conference on Aspect-oriented Software Development*, pages 161–172. ACM, 2009.

[ZKGG19] M. Baha E. Zarrouki, Verena Klös, Markus Grabowski, and Sabine Glesner. Fault-tolerance by graceful degradation for car platoons. In *Workshop on Autonomous Systems Design, ASD 2019, March 29, 2019, Florence, Italy*, pages 1:1–1:15, 2019.

[ZS19] Ellin Zhao and Roykrong Sukkerd. Interactive explanation for planning-based systems. In *10th International Conference on Cyber-Physical Systems (ICCPS 2019)*, 2019.