A Model-Based Runtime Environment for Adapting Communication Systems

Inauguraldissertation

zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
der Universität Mannheim

vorgelegt von

Martin Anton Pfannemüller

aus Frankfurt am Main

Dekan: Joachim Lutz
Referent: Prof. Dr. Christian Becker
Korreferent: Prof. Dr. Andreas Schürr

Tag der mündlichen Prüfung: 16. Februar 2021

Abstract

With increasing network sizes, mobility, and traffic, it becomes a challenging task to achieve goals such as continuously delivering a satisfying service quality. Self-adaptive approaches use feedback loops to adapt a managed resource at runtime according to changes in the execution context. Adding self-adaptive capabilities to communication systems—computer networks as well as supporting structures such as overlays or middleware—is a major research focus. However, making a communication system self-adaptive is a challenging task for communication system developers. First, the distributed nature of such systems requires the collection of monitoring information from multiple hosts and the adaptation of distributed components. Second, communication systems consist of heterogeneous components, which are, e.g., developed in different programming languages. Third, system developers typically lack knowledge about the development of self-adaptive systems. Hence, this work's overall goal is to allow system developers to focus on making a (legacy) communication system adaptive.

Motivated by these observations, this thesis proposes a model-based runtime environment for adapting communication systems called REACT. In contrast to self-adaptation frameworks, which offer a standard way to build self-adaptive applications, we refer to REACT as a runtime environment, i.e., a platform that is additionally able to plan and execute adaptations based on user-specified adaptation behavior. REACT includes the support for decentralized adaptation logics and distributed systems, multiple programming languages, as well as tool support and assistance for developers. The developer support is achieved using model-based techniques for specifying the reconfiguration behavior of the adaptation logic. Also, this thesis proposes an easy-to-follow development process. As part of that, it is needed to monitor the reconfiguration behavior of the self-adaptive system. Hence, this work also presents two dashboard-based visualization approaches called CoalaViz and EnTrace for providing traceability of self-adaptive systems for system developers and administrators.

This thesis follows a design science research methodology resulting in the design and implementation of the final artifacts. By that, this dissertation presents different REACT Loops, including specific ways to model and plan the adaptive behavior using satisfiability, mixed-integer linear programming, and constraint solvers. The prototypes of these approaches, including the two visualization solutions, are evaluated in multiple use cases. Therefore, this work provides an end-to-end solution for specifying the adaptive behavior, connecting a managed resource, deploying the system, as well as debugging and monitoring it.

Acknowledgments

This thesis would not have been possible without the support of several people. I would like to thank these people for their support throughout the past years.

First, I would like to thank my supervisor Prof. Dr. Christian Becker, for allowing me to pursue the PhD project in his research group, his overall support, and the possibility to always stop by at his office with any problem. Christian, thank you for your guidance and, in general, for the great time and atmosphere at your chair. It also has always been a pleasure to travel with you around the world, to explore all the nice places, and, of course, thank you for the education considering good food and wine.

I would like to thank Prof. Dr. Andreas Schürr for the great cooperation working together as part of the MAKI project and for agreeing to act as the second reviewer of this thesis. Andy, thank you for the cooperation and for your comments and ideas. The past years working together with you in the project have always been very productive, and your detailed remarks always helped a lot in improving any paper or the big DFG proposal.

I would like to thank Prof. Dr. Hartmut Höhle for finding the time to join the board of examiners.

Further, I would like to thank all the people I had the pleasure of working with throughout the years at the Chair of Information Systems II, namely Dr. Patricia Arias-Cabarcos, Martin Breitbach, Melanie Brinkschulte, Dr. Janick Edinger, Kerstin Goldner, Melanie Heck, Benedikt Kirpes, Prof. Dr. Christian Krupitzer, Dr. Sonja Klingert, Markus Latz, Dr. Jens Naber, Yugo Nakamura, Dr. Vaskar Raychoudhury, Dr. Felix Maximilian Roth, Dr. Dominik Schäfer, and Anton Wachner. It always has been like working with friends instead of just colleagues. In particular, I want to thank Christian. Thank you that you gave me the opportunity to get to know academic life by writing a paper as a student together with you. Also, thanks for teaching me how to write papers, the great collaborations, and all the very helpful comments.Thank you, Martin, for the collaboration in writing the most important papers regarding this thesis. Your writing skills and comments pushed every paper forward a lot. Thanks also for reading this entire thesis giving my very helpful comments. Janick, thank you for always supporting everybody in the team, even if you had a very high workload. Further, thank you, Jens and Max, for the possibilities to join your last papers as a second author to improve my writing skills. Thank you, Melanie and Melli,

for your valuable comments concerning this thesis. Also, thank you, Kerstin, for always helping with all kinds of administrative tasks throughout the years.

Also, I would like to thank all the people and co-authors I had the pleasure to work with from other groups and universities as well as from the industry. This especially includes colleagues from the MAKI project. Thank you Bastian Alt, Felix Fastnacht, Dr. Alexander Frömmgen, Stefan Haas, Stefan Herrnleben, Prof. Dr. Matthias Hollick, Jean Kaddour, Sounak Kar, Prof. Dr. Anja Klein, Robin Klose, Prof. Dr. Heinz Koeppl, Prof. Dr. Boris Koldehofe, Prof. Dr. Samuel Kounev, Dr. Wasiur R. Khuda Bukhsh, Prof. Dr. Philippe Lalanda, Veronika Lesch, Manisha Luthra, Michael Matthé, Dr. Mahdi Mousavi, Prof. Dr. Max Mühlhäuser, Magnus Nigmann, Prof. Dr. Amr Rizk, Dr. Bradley Schmerl, Prof. Dr. Andreas Schürr, Dr. Michele Segata, Dr. Roland Speith (né Kluge), Prof. Dr. Ralph Steinmetz, Vincent Voss, and Dr. Markus Weckesser. Especially, I want to thank Markus and Roland for the very seamless and complementary collaboration as part of the (sub) project.

This work has been funded by the German Research Foundation (DFG) as part of project A4 of the Collaborative Research Center (CRC) 1053–MAKI. I would like to thank each student involved in the project for their contributions, especially Anton Grauer, Axel Herbstreith, Aaron Kirner, Alexander Makarevich, Michael Matthé, Erik Penther, Alina Pollkläsener, Johannes Schaum, and Lukas von Hohnhorst.

Last but not least, I would like to thank my family and friends for always being there for me. To my parents, thank you for always supporting me and giving me the possibility to pursue my way. To my sisters Laura and Eva, for always encouraging me and simply being such outstanding sisters. To my girlfriend, Amrit, thank you for your motivation and your support. You always had my back in the case, e.g., an evaluation did not work as initially intended and, in general, supported, motivated, and helped me in pursuing my work regardless of the ups and downs.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ADL	Architecture Description Language
API	Application Programming Interface
CEP	Complex Event Processing
CFM	Context Feature Model
CNF	Conjunctive Normal Form
CP	Constraint Programming
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science
DNS	Domain Name System
DSL	Domain-Specific Language
DSPL	Dynamic Software Product Line
ECA	Event-Condition-Action
EQ	Evaluation Question
FAI	Feature Attribute Item
FESAS	Framework for Engineering Self-Adaptive Systems
FODA	Feature-Oriented Domain Analysis
IDE	Integrated Development Environment
IDL	Interface Definition Language
IoT	Internet of Things
JSON	JavaScript Object Notation
MANET	Mobile Ad-Hoc Network
MILP	Mixed-Integer Linear Programming
MQTT	Message Queuing Telemetry Transport
NoSQL	Not only SQL
OMG	Object Management Group

OSGi	Open Services Gateway initiative
PLE	Product Line Engineering
QoS	Quality of Service
RAM	Random-Access Memory
REACT	Runtime Environment for Adapting Communication sysTems
$\boldsymbol{R_F}$	Functional Requirement
$\boldsymbol{R_{NF}}$	Non-Functional Requirement
RPC	Remote Procedure Call
RQ	Research Question
SAS	Self-Adaptive System
SAT	Satisfiability
SDN	Software-Defined Networking
SEAMS	(Symposium on) Software Engineering for Adaptive and Self-Managing Systems
SLA	Service-Level Agreement
SLOC	Source Lines of Code
SOA	Service-Oriented Architecture
SPL	Software Product Line
SQL	Structured Query Language
SUMO	Simulation of Urban MObility
SWIM	Simulator for Web Infrastructure and Management
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
WSN	Wireless Sensor Network
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1. Introduction

Trends such as the Internet of Things (IoT) lead to a growing number of networked devices. According to International Data Corporation (IDC), the number of connected IoT devices will reach 41.6 billion in 2025 [1]. Looking at the example of connected cars alone, also, according to IDC, they will reach 76.3 million units by 2023 with a five-year compound annual growth rate of 16.8 % between 2018 and 2023 [2]. The rising number of networked devices is one of the reasons for the increasing complexity of the deployed systems. Nevertheless, the expanding complexity is not only due to the increasing number of devices but also to the high mobility of the devices and the increasing network traffic [1]. Additionally, the category of IoT devices represents heterogeneous groups of devices with different software implementations and requirements for the wireless and wired networks. Hence, it gets gradually more challenging to achieve goals, such as continuously delivering a satisfying service quality in the networks.

Self-adaptation enables a system to adapt itself at runtime according to changes in the execution context for taming this complexity [3, 4]. Self-adaptive systems (SASs) are able to adjust parameters or change the structure of themselves [5, 6]. From an architectural perspective, a SAS is separated into a system providing a service, called managed system or resource, and an adaptation logic that plans and executes adaptations [4, 7]. Taking the field of communication systems into account, the managed resource can be a hardware device such as a network switch or a node in an overlay network. Developing a SAS is a complex task that requires expertise in monitoring sensor data, analyzing this data, planning adaptations, and executing them [8]. For simplifying the development, SAS frameworks can help. However, when applying a framework for engineering a SAS, it should not be a requirement always to build a system from scratch. Also, a developer has to be able to specify the adaptation behavior in a concise way at design time. The goal of this thesis is to provide a model-based approach for engineering SASs in the communication systems domain, addressing the aforementioned problems.

1.1. Problem Definition

As developing SASs is a challenging task, frameworks for engineering SASs can be used for simplifying the development process and decreasing the development time [9]. The main goal here is to provide reusable structures that can be employed for avoiding to build self-adaptive systems from scratch. As seen in [10] and [11], there are many possibilities for frameworks such as general overarching component-based systems [12, 13] or approaches only focussing on specific use cases [14]. In order to address the challenge of adapting communication systems, a general framework must be able to provide means to support the heterogeneity and inherent distribution requirement of these systems. Distribution, in this case, implies that the adaptation logic itself can be deployed distributedly, which enables decentralized control [15] and that the adaptation logic is able to cope with distributed managed resources. Additionally, system developers typically lack knowledge about the development of SASs, which requires them to always work together with SAS engineers. Enabling system developers in the communication systems domain to directly plan their systems including adaptivity, or to enhance existing systems with adaptivity for improving them, can considerably enhance the domain system's performance and expand the use of SAS concepts. Furthermore, as distributed systems, especially in the IoT domain, are dynamic considering mobility and network churn, a (distributed) SAS deployment must be capable of being changed due to the changing resources. Accordingly, it must be possible to alter the deployment of the SAS components as well as the specification of the adaptation behavior at runtime. Finally, runtime changes of the deployment and the specification allow self-improvement [16].

In the field of self-adaptive systems, many researchers already tried to improve the development experience and efficiency. Approaches into this direction include, for instance, the MAPE-K [4] feedback loop structure or methods for changing a SAS at runtime using self-improvement [16]. As abstract architectures and methodologies cannot directly be applied to bring SAS techniques into actual systems, the goal of this thesis is to provide a novel and applicable development approach. Even though there already exist frameworks such as Rainbow [17, 18] or FESAS [19–21], they miss the explicit support of communication systems, assistance for system developers, or do not allow runtime reconfigurations.

1.2. Research Questions

Based on the problem definition presented in the previous section, the overall objective of this thesis is to provide a *generic* and *reusable runtime environment* for adapting *communication systems* with the possibility to *change the deployment at runtime*. Accordingly, this thesis answers the following three research questions.

RQ1: How to engineer a generic and reusable runtime environment targeting communication systems?

The first research question is concerned with providing the base functionality, which is needed for adapting communication systems in a generic and reusable way. Therefore, an answer to this question must include specific system facilities, which target communication systems and their distributed nature. The question also raises the problem for a generic and reusable way to specify the behavior of the SAS.

RQ2: How to support system developers in creating adaptation logics for new or existing systems without SAS engineering knowledge?

The second research question aims at enabling system developers, who work in a specific field of communication systems, to build new systems with adaptivity in mind or to enhance existing systems with adaptive behavior. In this case, especially the level of abstraction of the used specification approach, as well as its explicit representation, will be part of the answer. As an example, optimally, the system developer only has to learn a limited set of higher-level concepts, and existing knowledge in the domain of software engineering can be reused.

RQ3: How to engineer a runtime environment that enables changes of the adaptation logic after deployment?

As communication systems are considered distributed, dynamic, and mobile, it must be possible to change the deployment of a distributed and decentralized SAS using this thesis' approach. Changing the deployment includes the possibility to update the specification of the adaptation behavior. Updating the deployment can be needed due to different types of uncertainty, such as when specifying the system's behavior at design time. The third question examines how the runtime environment has to be engineered to allow these changes at runtime.

1.3. Contributions

As the overall objective and research questions imply, this thesis' contribution is a generic and reusable runtime environment for adapting communication systems.

As part of [11], we analyzed the landscape of existing frameworks for developing self-adaptive systems with a focus on the requirements for adapting communication systems. We show that none of the existing approaches fulfills all requirements, which will be presented in detail in Chapter 4. Hence, approaches are either limited to specific use cases, do not provide a ready-to-use decision engine, or in general, do not support a system developer in any way.

Based on the requirements, a design for the main artifact named REACT (**R**untime **E**nvironment for **A**dapting **C**ommunication sys**T**ems) is developed. REACT consists of reusable core components (*REACT Core*) as well as a ready-to-use *REACT Loop*. REACT Core represents the infrastructure for reusable feedback loop instances. A feedback loop instance named REACT Loop represents an actual decision engine planning and executing adaptations. This thesis employs model-based specifications for the REACT Loops due to the level of abstraction they provide. Apart from REACT's design and architecture, the approach suggests features that allow for extensions and high applicability. REACT Core also provides an optional context module, which can be used to increase the execution speed of a REACT Loop for already observed contexts, and for distributing context information for external software components outside of REACT. As part of the design, this thesis also proposes a development process. This process supports system developers in applying REACT, addressing the missing developer support in related works. Implementation-wise, this thesis contributes with a prototypical implementation of REACT Core, as well as implementations of specific REACT Loops using REACT Core.

Each REACT Loop is evaluated in different use cases and in different evaluation settings. This includes a comparison with the well-known approach Rainbow [17, 18] for determining the strengths and weaknesses of this thesis' approach. Further, in a feasibility study, the REACT Loops are compared directly in the same use case. This study also explores the potentials of combining multiple REACT Loops. Apart from quantitative measurements, REACT, and the REACT Loops are discussed qualitatively considering, e.g., capabilities and modeling expressiveness.

In order to support developers even further and additionally to the development process, Chapter 8 proposes two approaches named CoalaViz and EnTrace connected to REACT for visualization and making adaptation decisions traceable. Both approaches support system developers and administrators alike in observing if the specified adaptation behavior is implemented correctly at runtime.

Hence, this thesis provides an end-to-end solution starting with the specification of the adaptive behavior, the connection to a managed resource, the deployment of the SAS, as well as debugging and monitoring the SAS using one of the visualization approaches.

1.4. Structure

Beginning with Chapter 2, fundamentals in the field of SASs, the concept of feedback loops, and possibilities for decision-making in SASs are presented. As this thesis follows a model-based approach using the idea of (context-aware) feature models originating from Dynamic Software Product Lines, Chapter 2 covers an introduction of the same. Chapter 3 outlines the applied design science research methodology of this thesis as introduced by Peffers *et al.* [22]. Chapter 4 introduces requirements that must be fulfilled for a runtime environment aiming at enhancing communication systems with adaptive behavior. Based on the requirements, Chapter 5 analyzes related works in the fields of self-adaptive systems as well as Autonomic Networking. Chapter 6 presents the design and implementation of REACT Core as a foundation for executing REACT Loops, its context module, and also includes the presentation of a corresponding development process. Next, Chapter 7 presents three REACT Loops containing respective designs and implementations. The prototype implementations are evaluated in different use cases from the communication systems domain. This chapter also compares the loops and identifies the potentials of combining multiple loops as part of a feasibility study. The two visualization modules, CoalaViz and EnTrace, are presented and evaluated in Chapter 8. Chapter 9 answers the research questions by discussing the results considering the different functional and non-functional requirements of Chapter 4. Additionally, this chapter outlines threats to validity and limitations as well as prospective future mitigation of them. Finally, Chapter 10 concludes this thesis and gives an outlook on future work.

2. Fundamentals

The first chapter motivated this thesis, specified the research questions, and briefly described the contributions of this work. This chapter presents fundamentals as background for this thesis. Section 2.1 defines the term "self-adaptive system" and outlines related general concepts and architectures. Next, Section 2.2 presents the approach of having an (external) adaptation logic deciding to adapt the managed resource. This includes a description and explanation of the state-of-the-art MAPE-K control architecture. In order to specify the adaptation behavior, different decision criteria can be used. Therefore, Section 2.3 presents different ways of deciding how to adapt a system. Finally, Section 2.4 introduces the idea of feature modeling following the concept of (Dynamic) Software Product Lines for specifying the reconfiguration options of a system.

2.1. Self-Adaptive Systems

This section gives an overview of SASs. First, this section introduces the concept using multiple definitions and descriptions of the term "self-adaptive system". Oreizy *et al.* gave one of the first definitions for self-adaptive systems in 1999 [5, p. 55]:

> *Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program.*

Another definition is given by Laddaga *et al.* in 2001 [3, p. 1]:

> *Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.*

In comparison, Oreizy *et al.* explicitly incorporate the context by mentioning the operating environment, including user input, while the definition of Laddaga *et al.* emphasizes the "self-" and evaluation aspect of a self-adaptive system. The term "self-" in self-adaptive means that the software system decides on its own (also named autonomously [7]) to adapt its behavior corresponding to a perceived change of the environment. Laddaga's definition also mentions the idea to have an internal evaluation of the adaptation behavior that tries to improve the system's performance constantly over time. The improvement of the performance is, e.g., possible using a learning-based component [4]. Hence, the definition of Laddaga *et al.* is rather related to the concept of self-aware computing systems, which inherently contains a learning component [23].

There are many more definitions present, such as in [7] or [8]. Recently, Weyns took multiple definitions into account and combined them into "two basic principles" determining a self-adaptive system from his point of view [24, p. 402]:

> *1.* ***External principle****: A self-adaptive system is a system that can handle changes and uncertainties in its environment, the system itself, and its goals autonomously (i.e. without or with minimal human interference).*
>
> *2.* ***Internal principle****: A self-adaptive system comprises two distinct parts: the first part interacts with the environment and is responsible for the domain concerns (i.e. concerns for which the system is built); the second part interacts with the first part (and monitors its environment) and is responsible for the adaptation concerns (i.e. concerns about the domain concerns).*

The first principle specifies the capability to autonomously achieve certain goals without any or with minimal human intervention. The second principle describes the separation between adaptation logic managing a connected domain system providing a service. This thesis follows these two basic principles for defining a SAS, as they consider the general goal of SASs on the one hand and the (internal) architecture of them on the other hand. The principles include the ideas of the first two definitions while omitting the learning aspect. For this thesis, a SAS does not automatically contain a learning component as present in self-aware [23] or organic computing [25].

Finally, considering the terminology, according to Salehie and Tahvildari [26] many researchers such as Huebscher and McCann [27] use the terms "self-adaptive system", "autonomic system", and "self-managing system" synonymously. When only comparing the terms, Salehie and Tahvildari consider self-adaptive systems to be a limited subcategory of Autonomic Computing [26]. Looking at the layered architecture in [6]—consisting of application(s), middleware, network, operating systems, and hardware—self-adaptive software can be found mainly on the application and middleware layer [26]. Opposing to that, the term Autonomic Computing has been applied on the network layer (see, e.g., [28]) as well as on the operating system layer (e.g., see the reincarnation server of the Minix operating system [29]) [26]. Still, even though the terms have been used in different domains, the underlying concepts can be used interchangeably [26]. Hence, this thesis follows this statement and does not make a difference between the terms self-adaptive, Autonomic Computing, or self-managing system.

2.1.1. Self-* Properties

The so-called self-* or self-CHOP (**c**onfiguration, **h**ealing, **o**ptimizing, **p**rotecting) properties are defined as fundamental for engineering self-adaptive systems [4, 26, 30]. More detailed, as specified by Salehie and Tahvildari and depicted in Figure 2.1, self-adaptiveness is built on top of a *primitive* and *major level* of properties [26]. Accordingly, Salehie and Tahvildari consider the primitive and major levels as the foundation for the concept of self-adaptiveness.

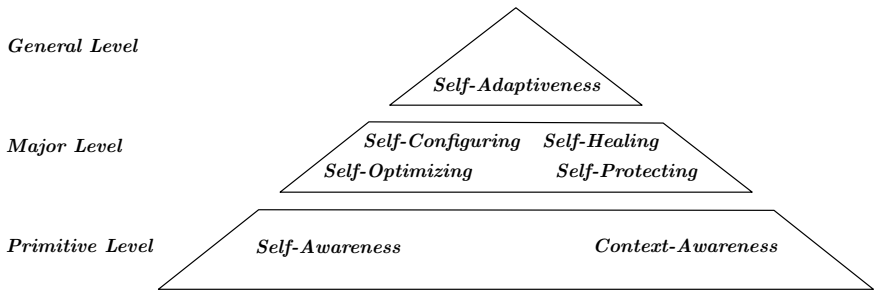


Figure 2.1.: Hierarchy of the self-*/self-CHOP properties [26].

On the *primitive level* exist two fundamental properties a self-adaptive system must have: *Self-awareness* and *context-awareness*. While the first concept describes the ability to sense the internal state of a system, the second concept means

that the surroundings or context of the system can be monitored. Without these capabilities, a system is not able to monitor the current situation resulting in no possibility to decide if an adaptation is needed in the first place. The original definition of self-awareness, meaning a system is aware of its own states and behaviors, is given by Hinchey and Sterrit [31]. In their case, self-awareness only results in a system "*being aware of its internal state*" [31]. This is the definition followed by Salehie and Tahvildari [26] as well as by this thesis. Although having the same name, self-aware computing systems inherently contain a reasoning and learning component [23]. Hence, self-awareness, as defined on the primitive level, is not directly related to self-aware computing systems.

Looking at the context and *context-awareness*, both terms are largely coined by the pervasive computing community [32]. A popular, rather broad definition of context is given by Dey in 2001 [33, p. 5]:

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.*

Context-aware systems have been defined in 1994 by Schilit *et al.* as systems that "*can examine the computing environment and react to changes to the environment*" [32]. In the domain of SASs, context-aware systems have been defined as systems being aware of their operational environment [34]. This definition was applied by Salehie and Tahvildari, complementing the primitive level.

On the following layer, four self-* capabilities reside on the so-called *major level*. These capabilities are also subsumed in the term self-managing system [4]. Self-managing software results in a system that tries to work all the time without interruptions. This aspect frees system administrators from low-level tasks. The major level and self-management consist of the following four self-* capabilities: *self-configuration*, *self-optimization*, *self-healing*, and *self-protection* [4]. A self-configuring system intends to set itself up according to high-level policies of the overall IT environment. Thus, it embeds itself seamlessly into the existing IT systems. Self-optimization describes a learning component of the system, which adjusts the adaptations for better results. This means the system is able to improve its performance on its own gradually over time. Of course, problems

can occur in this process or in general while running a system. If a problem arises, the self-healing mechanism is employed. This mechanism tries to locate, analyze, and correct problems at runtime. The last capability is self-protection. It automatically detects and defends against attacks or cascading problems that could not be solved by the self-healing process. Additionally, it reacts to early reports based on sensor data to reduce the impact of arising problems. Following the definition of Kephart and Chess [4], all self-adaptive systems are supposed to have these properties in common. However, in practice, there exist self-adaptive systems focussing only on a subset of the self-* capabilities. As self-configuration is the foundation for executing adaptations, this is considered as present whenever any of the other self-* properties is fulfilled [30]. There are approaches focussing on self-healing (e.g., [35]), self-optimizing (e.g., [36]), or self-protection (e.g., [37]). Additionally, there are approaches targeting multiple self-* properties, such as [38] aiming at self-healing and self-optimizing. Overall, most approaches target either self-healing or self-optimization, while self-protection has not been the focus of the research community until now [39].

2.1.2. Architecture of Self-Adaptive Systems

Considering the possible architectures of a SAS, there are two compositional approaches to build a self-adaptive system [26]. As shown in Figure 2.2, the structure of a self-adaptive system can be categorized into the two categories *internal* and *external* [26]. The two compositional approaches define how the managed resource and the adaptation logic are combined. The adaptable system or software is also called managed resource [40] or managed element [4], representing the system actually performing a task. The adaptation engine, also named autonomic manager [4] or adaptation logic [17], reconfigures the former. This thesis will use the terms managed resource and adaptation logic in the following.

In the architecture of a SAS, either the adaptation logic is part of and interwoven with the managed resource (see Figure 2.2 (a)), or the adaptation logic is designed as an external component (see Figure 2.2 (b)) communicating with the managed resource. The internal approach is faster to implement and may be an option in smaller local systems [26]. The maintainability is higher in the external approach. However, this approach needs communication between the adaptation logic and the

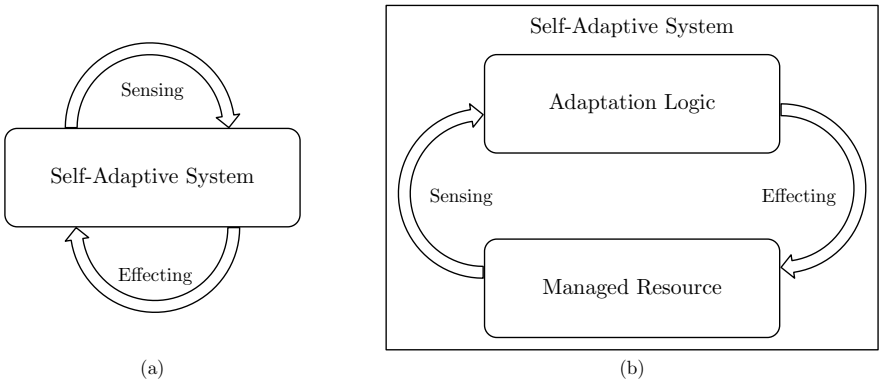


Figure 2.2.: Internal (a) and external (b) adaptation logic architectures [26].

managed resource. The adaptation logic constantly adapts the system according to the information received from the managed resource, while the managed resource provides the actual domain functionality of the system. The managed resource can be a hardware or software component.

The described external architecture is in line with the "Internal Principle" of Weyns' definition [24, p. 402]. As the external architecture is more scalable, exchangeable, and reusable, this is the broadly used method to implement self-adaptive systems [9,26]. Scalability is achieved, e.g., by having dedicated machines only for the adaptation logic. The independence of the adaptation logic in the external approach also makes it easy to use the same adaptation logic for multiple managed resources or to compare different adaptation logic approaches by exchanging them. This is not easily possible with the internal approach when the managed resource is interwoven together with the adaptation logic. Salehie and Tahvildari have published a survey on self-adaptive systems in which no system uses the internal approach [26]. More recently, Krupitzer *et al.* identified a single system following only the internal approach and two systems supporting both the external and internal approaches [9]. Thus, most self-adaptive systems consist of a separated adaptation logic and managed resource [41]. Based on these observations, the following focuses on external adaptation logics.

The adaptation logic and the managed resource are connected in two ways. The adaptation logic sends messages containing configuration changes to update the managed resource, while the managed resource sends data about itself and its context (cf. self-awareness and context-awareness in Section 2.1.1) to the

adaptation logic. This data can, e.g., be sensorial or statistical. The adaptation is accomplished by either changing parameter values or by exchanging components as part of the managed resource [6, 41]. Parameter adaptation changes the system parameters, while compositional adaptation changes structure, architecture, or both. A system can be used as managed resource if it is able to provide sensor information and receive adaptation actions.

The important component that makes a system self-adaptive is the adaptation logic. The adaptation logic must sense changes, understand them, plan adaptations, and execute them. Thus, much research has been done on finding effective ways to design this component. In the last years, a universal architecture for developing the adaptation logic has emerged, which is presented in the next section.

2.2. Adaptation Logics and MAPE-K

According to Brun *et al.*, the generic way to achieve self-adaptation is to use feedback loops [7]. A feedback loop consists of four steps: collect, analyze, decide, and act. This model is an advancement of the sense-plan-act approach [42,43] taken from the early development of artificial intelligence [7]. The *collect* component collects relevant data from the environment. The data can consist of, e.g., sensorial data or user input. With the data, the adaptation logic is able to determine the state of the system. The next step is to analyze the selected raw data. The *analyze* component structures the data and reasons about it using, e.g., models or policies. Based on this structured data, the *decision* component determines how the system state may be improved. In this step, it may be possible to use probability theory to determine the best adaptation according to the current state. The *act* component executes the adaptation by sending a message with the planned changes to the managed resource. Then, the *managed resource* adapts according to the received plan.

Kephart and Chess have used this generic control loop to specify an adaptation logic using four functional parts using a shared knowledge base: *Monitor*, *Analyze*, *Plan*, *Execute* with *Knowledge* [4]. The initial letters are the reason to call this approach the MAPE-K cycle. The MAPE-K cycle is embedded in a component called autonomic manager that represents the adaptation logic [4]. Figure 2.3

shows the loop as part of an adaptation logic, which is connected to a managed resource via a sensor and an effector. The MAPE-K cycle starts with monitoring the raw data coming from sensors—sometimes also called probes [17]—of the managed resource(s) [40]. As in a context-aware system, sensors either push data to the monitors, or they pull data from the sensors [44]. The monitor not only gathers and monitors the data but—according to Brun *et al.*—it also filters the data [7]. Next, the analyzer of the adaptation logic analyzes this prepared raw data. This includes the identification of constraint violations and their reasons. The following planning phase determines necessary changes in order to get the best possible result for the system or to resolve any problem identified in the analysis phase. Finally, the execute part executes the developed plan using effectors in the managed resource. This can include the orchestration of the execution or the decomposition of a plan into specific commands.

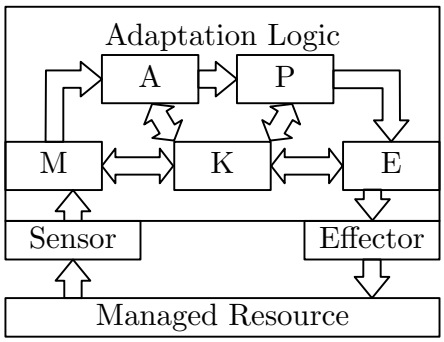


Figure 2.3.: MAPE-K feedback loop architecture with connected managed resource [4].

The MAPE components communicate via direct communication channels. Additionally, Kephart and Chess introduced a knowledge component resulting in the MAPE-K architecture [4]. This knowledge component can, e.g., be used to store all inputs and outputs of each MAPE component for future reference. The data enables the use of, e.g., (external) machine learning techniques for self-improvement [16].

Although the MAPE-K approach is a good guideline for developing self-adaptive systems, it does not define how a particular MAPE-K-based feedback loop works specifically. As there are multiple ways to specify how an adaptation logic using the MAPE-K architecture adapts a managed resource, the following section introduces different decision-making approaches for SASs.

2.3. Decision Making in Self-Adaptive Systems

As defined by Lalanda et al., the adaptation behavior of a SAS can be specified using different approaches, i.e., rules, models, goals, and utilities [30]. While rules, also named policies, constitute the most simple form of knowledge and specification, model-, goal-, and utility-based systems enable more complex ways to specify the behavior of a SAS. Additionally, it is possible to combine multiple ways for specifying the decision making in a hybrid approach. However, this also leads to the fact that it is not always possible to clearly distinguish one approach from another. Hence, there exists also an overlap between some approaches. In the following, this section introduces the four different approaches as presented in [30].

2.3.1. Rules/Policies

Rules, also named policies, typically follow the event-condition-action (ECA) pattern [30]. Accordingly, in the case of an event and if a condition is met, an action is performed. Rules are easy to specify and understand. This fact also allows stakeholders, such as end-users, who are not familiar with system development, to express simple rules. However, it can get hard to manage a large set of rules resulting in overlapping policies or conflicts [30]. Additionally, rules are defined and can be verified statically at design time resulting in fixed non-dynamic behavior [9] without an additional self-improvement [16] layer. Hence, rules are considered mainly for simpler systems, which do not need a large rule base for their adaptive behavior [30].

In the simplest case, the evaluation of rules happens only on the foundation of current sensor information [30]. Then, there is no state that is stored as part of the rule-based adaptation logic simplifying the evaluation of the rules, as no historical values are taken into account. This is also the reason to call the behavior of these kinds of approaches reflex-based [30]. ECA rules directly produce adaptation plans from the status events, which can be executed. Even though rule-based adaptation logics are rather simple, it still is possible to combine them with learning capabilities to update the rule base at runtime [45].

2.3.2. Models

Models in software systems can be defined as follows [46]:

> *A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.*

In SASs, models can be used to represent context information as well as the architecture of the managed resource [30]. Accordingly, many different model types exist [9]: *system models*, *goal models*, and *environmental models*. The system model category can further be divided into *architectural models*, *feature models*, and *behavioral models*. Figure 2.4 shows the hierarchy of these models.

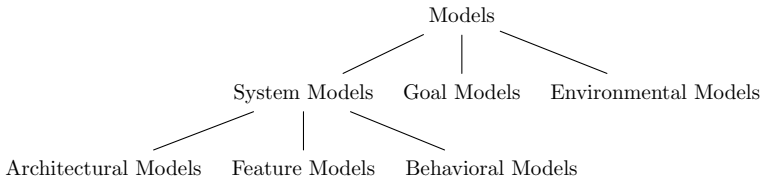


Figure 2.4.: Model types for decision making in self-adaptive systems [9].

Beginning with system models, architectural models represent the architecture of the managed resource. This representation can be achieved using, e.g., Unified Modeling Language-based [47] techniques or customized architecture models such as Acme models [48], which are applied in the Rainbow framework [17]. Feature models, which are visualized in hierarchical tree representations [49], specify the configuration options of a software system as part of Software Product Lines (SPL). This model type is explained in more detail in Section 2.4. Behavioral models represent the adaptation behavior on a higher abstraction level without direct links to the managed resource, e.g., using state machines [50]. Going back to the overall category of models, goal models can be used to specify one or multiple system goals of a SAS [51]. Goal models are considered more dynamic in pursuing the goals of a system than rules or system models [30]. This is due to the fact that rules and system models are statically defined, which can lead to unspecified situations. Finally, environment or context models are used for capturing the context of a system [52]. A context model can represent physical context, captured using physical sensors, as well as software-based and user context. In the same way that it is possible to combine multiple decision-making strategies, it is possible to

combine multiple modeling approaches. As an example, Contex-Aware Dynamic Software Product Lines (DSPL) [53] constitute a combination of feature models representing the reconfiguration space of the managed resource and an explicit model of the observed context.

Either way, when using a single representation or a combination of the presented model representations, a SAS can use problem solvers for reasoning and for finally planning and executing adaptations (as, e.g., in [54]). This involves the transformation of the model into a problem domain and the use of problem solvers such as satisfiability (SAT) (e.g., [55]), mixed-integer linear programming (MILP) (e.g., [56]), or constraint satisfaction problem (CSP) (e.g., [57]) solvers.

2.3.3. Goals

In the previous section representing the category of models, goal models have already been introduced. However, goal-based decision making does not only consist of models. In general, goals do not rely on fixed adaptation rules or models, but rather a goal-based adaptation logic has to create specific plans fulfilling the goals [30]. This makes goal-based approaches more dynamic than fixed rules or models created at design time. As part of a goal-oriented requirements engineering process, a developer has to specify high-level objectives, including managed resource-specific constraints [58]. In this process, the high-level goals have to be decomposed into subgoals [59]. A specific definition considering goal-based systems has been given by Salehie and Tahvildari as follows [60]:

> *Given an adaptation goal set G, an adaptation action set AC, and an attribute set AT from a software system, the problem is how to build a goal-action-attribute model, and to select the appropriate action ac_i at run-time to satisfy goals under different conditions.*

Based on this definition, in this specific case, the authors propose the Goal-Action-Attribute Model (GAAM) as a solution [60]. This approach enables to express goal hierarchies, attributes, and actions, which influence goals. Additionally, it allows for prioritizing the goals resulting in multi-objective optimization. This already shows that models are often used to express goals. Hence, especially for goal-based approaches, it is not easily possible to distinguish them from, e.g., the (goal) model decision criterium.

Still, for the planning process in general, the task is to map the current situation and the goals to the available adaptation actions [30]. In [61] and [62], Kramer and Magee's three layer architecture [51] is used, consisting of a specific layer responsible for creating plans based on higher-level goals the adaptation logic can choose from. In the process of the mapping, techniques such as forward and backward search-based algorithms are often used when planning adaptation actions [30]. As indicated by the multiple objectives stated as goals, the planning process typically also includes conflict handling, as goals can contradict each other [9]. Goal-based approaches enable to monitor the fulfillment of the specified goals of the system [63,64]. In contrast to rule- or general model-based approaches, this enables observing the accomplishment of the system's objectives.

2.3.4. Utilities

The final category for decision making consists of utility-based systems [30]. The main objective of these approaches is to compare different adaptation options for choosing the best one in a certain state. For this to work, a utility function measuring the usefulness of adaptations is needed. A utility can be positive when the system should pursue some behavior, such as a high performance, or negative when some behavior should be avoided, such as high costs. Utility functions typically combine many parameters, representing the different (performance) attributes of the system status, into a single metric [30]. It is also possible to have separate utility functions for different non-functional properties, as present in, e.g., [36]. In general, the specification of utility functions is more complex and less intuitive compared to rules and models and considered as a hard task [30].

In combination with one of the other categories, utilities can increase the planning possibilities as an action can also be mapped to benefits and costs. In the adaptation process, utilities can be used as additional constraints or optimization objectives for problem solvers. As an example, the performance in a system should be as high, while the costs should be as low as possible. This shows that especially utility-based approaches capture a tradeoff between the usefulness of a non-functional objective and the costs of pursuing it. Tesauro *et al.* call their approach goal-based by applying utilities [65]. This again shows the problem of distinguishing approaches from one another.

As shown in this section, each decision criterium has certain advantages or disadvantages. Additionally, it is not possible to clearly distinguish the different decision criteria from each other. As specific examples, it is also possible to consider a set of rules as a model, while using a goal model can be regarded as a model- or goal-driven decision criterium. Nevertheless, considering the four categories for decision criteria as they are presented in [30], this thesis uses a model-based approach. Specifically, the approach of this thesis is based on the concept of feature modeling, which is part of the software product line (SPL) methodology. Accordingly, the following section introduces SPLs as well as the extended variant called Dynamic SPLs.

2.4. Software Product Lines

This section presents details of the SPL-based feature modeling approach used for specifying the configuration space of a software product. Software product line concepts can also be used for modeling the reconfiguration behavior of SASs, including the internal and external context of the managed resource. First, Section 2.4.1 presents SPLs and their static configuration approach. This includes the introduction of the SPL lifecycles as part of Section 2.4.2. Section 2.4.3 specifically presents the feature diagram methods for modeling the feature models in a graphical way. Section 2.4.4 shows an extension for supporting dynamic feature selection at runtime: Dynamic software product lines (DSPLs). Based on the idea of dynamic reconfiguration in DSPLs, context-aware feature models or, in short, context feature models used later in this work are introduced.

2.4.1. Introduction to Software Product Lines

According to [66], the idea of SPLs emerged from general economics. Starting with the development of the conveyor belt by Ford, the concept of *economies of scale* arose. Economies of scale "*arise in the production of multiple implementations of a single design*", leading to cost reductions [67, p. 17]. This mass production was cheaper but did not have many diversification possibilities compared to individual handcrafted items [68, p. 4]. Based on this concept, the idea of reusing major parts of similar products that are only distinct in smaller individual parts devel-

oped. This approach is called Product Line Engineering (PLE), and the goal is *economies of scope*. Economies of scope mean "*efficiencies wrought by variety, not volume*" [69, p. 142]. The result of applying PLE are mass-produced but individualized products resulting in the concept of mass-customization. Accordingly, Davis defines this idea of mass-customization as follows: "*Mass customisation is the large-scale production of goods tailored to individual customers' needs.*" [70]. PLE enables companies to build up a generic platform that can be used as the basis for all product variants. Reusability is the key here for the resulting cost reductions. The software development community became aware of this idea, which resulted in the SPL method [66]. The tradeoff between individual handcrafted items and mass-produced items can be seen in software engineering as the difference between individual development and standard software [68, p. 4].

The Software Engineering Institute of the Carnegie Mellon University defines SPLs as follows [71]:

> *A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

The definition demonstrates the original PLE idea of having a common platform and developing multiple individual features on top for meeting the needs of one specific area. This common platform is created using so-called core assets. Hence, the definition shows an important step in SPL development–defining commonalities of the whole product line. This step is part of one of the two SPL lifecycles, which the next section introduces.

2.4.2. SPL Engineering Process

The two parts of the SPL engineering process are domain engineering and application engineering. Figure 2.5 depicts the whole SPL process, including the two cycles [66]. As depicted, both lifecycles consist of multiple steps and require to already have business planning, product, and requirement information present. Based on these fundamentals, the following lifecycles aim at identifying product-specific as well as common features, which apply to the whole product line. In the following, the two lifecycles are introduced.

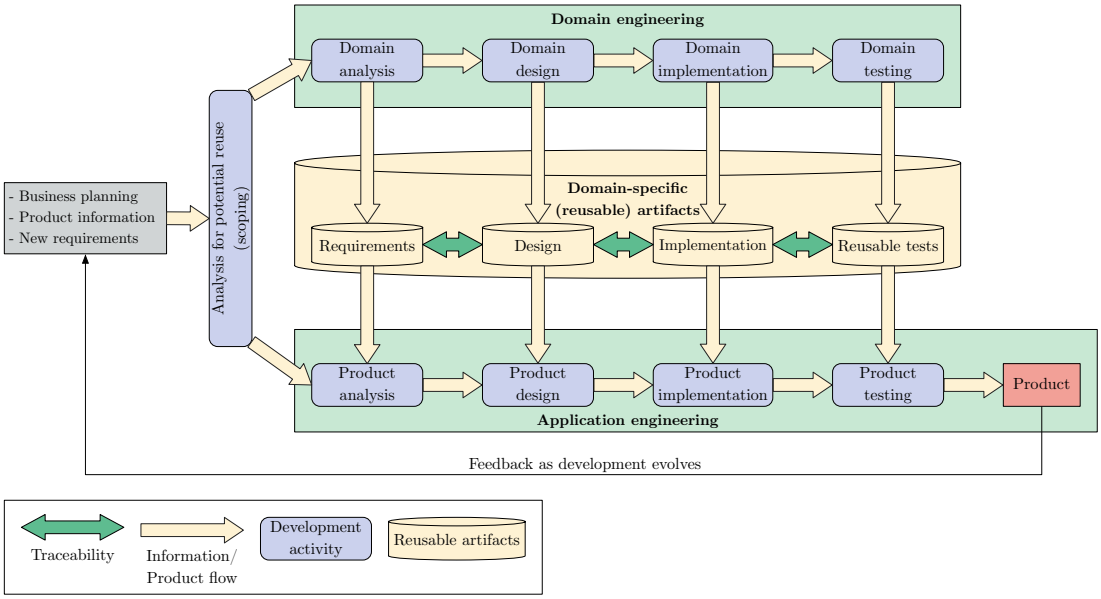


Figure 2.5.: SPL Lifecycles [66].

The main goals of the *domain engineering* process are to define the commonality and the variability of the product line [68, p. 21]. Commonality and variability are defined using a variability model, which defines common and exchangeable system parts. Additionally, the set of applications of the software product line is defined. Each step creates reusable artifacts that employ the defined variability. These domain-specific artifacts, which are shown in yellow in Figure 2.5, compose the platform the software products rely on. The artifacts are connected by traceability links to retain consistency avoiding inconsistent artifacts, which may result in unusable or broken application products.

The *domain engineering* process begins with the *domain analysis*. This includes requirements engineering to define and document the "*common and variable requirements of the product line*" [68, p. 25]. The most interesting product of the domain analysis process for this thesis is also created here—the variability model. This variability model represents the configuration options of an SPL, and it will be introduced in detail in Section 2.4.3. *Domain design* results in a high-level reference architecture usable for the whole product line. The requirements from the first step are the input for this step. Then, the *domain implementation* step creates specific designs and implementations that are common to the whole SPL based on the reference architecture. *Domain testing* is a verification and validation

step, checking all the steps that happened before. Furthermore, this measure tests the common artifacts to reduce errors in the common platform right from the start [68, p. 27].

Application engineering aims to exploit the common platform of the SPL as well as possible and to relate the software product to the reusable domain-specific artifacts [68, p. 21]. Additionally, it binds the variability model to the actual product instance that is to be built. *Product analysis* is also concerned with requirements engineering. Here, the focus lies on identifying the differences between platform and product requirements. *Product design* uses the reference architecture created in the domain design step to instantiate an actual product architecture and configures it to the needs of the product. *Product implementation* creates the application as a combination of the common platform implementation artifacts and product specific modules. This results in the finished application exploiting as many domain-specific artifacts as possible. The last step, *product testing*, runs tests on the finished software product. The outcome is a report with the test results. This ends the application engineering and results in the finished product. As seen in Figure 2.5, the products are used as feedback for possible new business planning requirements.

After the brief introduction of the whole SPL process, the next section focuses on the models to define variability in the product line.

2.4.3. Variability Models

For specifying variability models, features are used. A feature is a "*system property that is relevant to some stakeholder and is used to capture commonalities or discriminate between systems*" [72, p. 267]. According to Pohl *et al.*, variability models can be created using standard Unified Modeling Language (UML) modeling techniques [68, p. 75 f.]. However, since UML is not specifically designed for facilitating SPL development processes, so-called feature models are the common way of specifying dependencies between features of an SPL [49].

Features are organized in feature diagrams. They are a tree structure representing the software system as a whole. The tree consists of a root feature with several layers of child features. A feature model generally consists of a feature diagram and additional information such as information on the binding time or priorities.

Benavides *et al.* identified three major categories in the domain of feature models: basic feature models, cardinality-based feature models, and extended feature models [73]. In the following, they are briefly introduced.

Basic feature models: Based on the literature review of Chen *et al.*, most basic feature modeling approaches are grounded on the Feature-Oriented Domain Analysis (FODA) approach by Kang *et al.* [49, 74]. Kang *et al.* were the first who introduced the term feature model and proposed a hierarchical feature tree structure for specifying all features of an SPL [73]. The original FODA notation includes the elements shown in Figure 2.6 (a): *and* as well as *xor* groups, and the possibility to define *optional* features. Also, features can require each other or can be declared as mutually exclusive. These properties are called cross-tree constraints. However, these characteristics were not depicted graphically yet. In the graphical representation, plaintext at the ends of the edges was used for the features themselves. Later, Kang *et al.* extended their original approach, e.g., by representing features as text boxes [75, 76]. Parent features that have multiple child features are provided by either one or multiple of these child features. In this case, child features specialize a parent feature. Furthermore, new elements were introduced to the original FODA notation later [77]. Griss *et al.* [77] added an *or* operator as well as graphical representations for the cross-tree constraints. These new elements are depicted in Figure 2.6 (b).

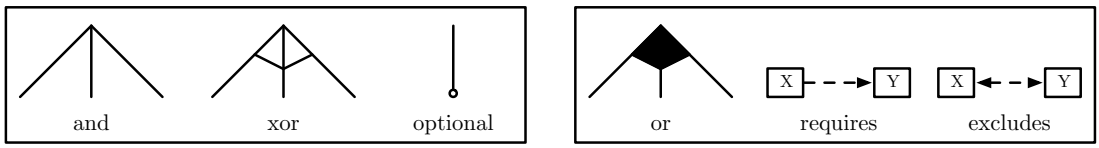


Figure 2.6.: Basic feature diagram elements: Original FODA notation (a) [49] and extended FODA notation (b) [77].

Cardinality-based feature models: Riebisch *et al.* propose that there are UML-like multiplicities covered by feature models [78]. In order to improve the understanding and to formally define them, they introduce an annotation for representing the multiplicities of feature sets. Later, these cardinalities were defined more specifically as group type cardinalities [72, 73]. A group type cardinality defines explicitly when a parent feature is part of the system, how many child features can be selected in a configuration. As an example, a group

type cardinality of *0..** means that the child features are all optional. Also, there are feature instance cardinalities, which denote how many instances of a feature can exist at runtime [72]. For distinguishing both cardinality types, group type cardinalities are denoted with angle brackets and feature instance cardinalities with square brackets (see Figure 2.7 (a)). Analogously to the UML notation, a cardinality is annotated with a lower and an upper limit. In summary, cardinalities state in a clear way how to interpret a feature diagram. They enhance the overall expressiveness and modeling capabilities to state the needed constraints more specifically.

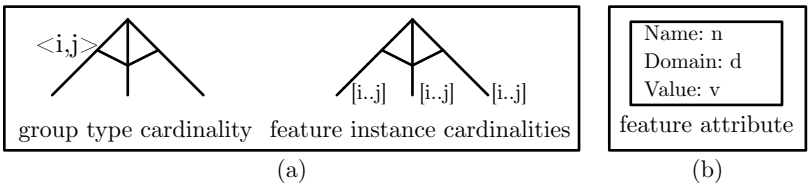


Figure 2.7.: Extended feature diagram elements: Cardinality-based notation (a) [72, 78] and feature attributes (b) [79].

Extended feature models: According to Benavides *et al.*, extended feature models, also called advanced or attributed feature models, are able to express additional attributes of features [73]. There is no consensus on the information an attribute should contain. However, most approaches state that an attribute contains a name, a domain, and a value. Using these attributes, it is possible to, e.g., describe requirements for a certain feature more specifically. Since there are multiple approaches for describing attributes, it is also not clear how to depict them. This thesis uses the notation introduced by Benavides *et al.* [79]. The notation can be seen in Figure 2.7 (b). Additionally, it is possible to express constraints between features and attribute values in the form of Boolean and arithmetic formulas. This allows expressing conditions between attribute values and features, e.g., requiring a specific attribute value if a particular feature is activated.

This section presented the generic and static SPL approach as well as extensions of the FODA notation for feature diagrams. Based on this introduction, Dynamic SPLs, as well as context-feature models, are introduced in the following section.

2.4.4. Dynamic Software Product Lines

Due to the demand of today's environments, adaptability gets gradually more important for software systems [66]. Static SPLs do not fulfill this requirement as the variability defined in feature models gets bound at design time. The difference between SPL and DSPL binding can be seen in Figure 2.8 (a) and (b). A software product built using the SPL approach is configured once at design time. The first step is to apply the feature model for selecting overall valid configurations from all possible configurations, shown as a hexagon in the figure. Then, a configuration for the product to be built is selected. Thus, the developer builds such a variant for a rather static execution environment. Hence, the software might and probably will perform sufficiently in exactly this environment. With a dynamically changing context, SPL-based software possibly does not perform well anymore due to the requirement for adaptation and reconfiguration at runtime [80].

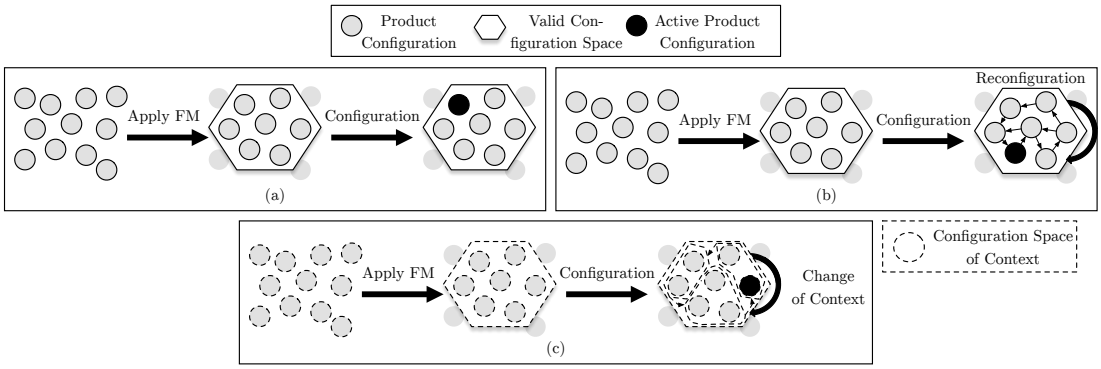


Figure 2.8.: SPL (a), DSPL (b), and context-aware (c) configuration. FM: Feature Model [81].

Software built using a DSPL is able to adapt itself, e.g., to changing user preferences or other context changes. This is realized by binding features at the start of the software and at runtime repeatedly shown in Figure 2.8 (b). Like in the SPL approach, the feature model is applied for selecting valid configurations. Then, a valid start configuration is selected at design time. In the DSPL approach, the valid configurations are connected by arrows building a directed graph. The product changes its configuration based on this graph defining possible transitions between all valid configurations. This enables adaptive behavior for the software. As configuration changes are triggered by the context, it is crucial to monitor the context while always storing a model of the current system and the state of

its environment. In fact, in order to plan a good reconfiguration, it is the most important task for the application to monitor itself and the context and change the configuration based on the monitoring results [66].

Since the context is essential for reconfigurations (cf. context-awareness in Section 2.1), extended feature models have been further enhanced with context modeling capabilities called context feature models (CFMs) [53, 81]. CFMs contain a parallel tree structure in the same way as the (system) features in feature models for specifying the context. This parallel context tree enables to specify constraints between context and system features and attributes. When using a CFM at runtime, the context tree is instantiated, which automatically restricts the possible ways the system can be reconfigured. Figure 2.8 (c) shows this process. In a first step, the feature model restricts the set of all configurations to valid configurations only. After the system is started with an initial configuration, reconfigurations between product configurations are constrained by the current context, as indicated with the dashed lines in the figure. As the context of the system changes, the available possible configurations change as well.

Related to this, the context monitoring and modeling can be divided into the *closed* and *open (world) approach* [82, 83]. The closed approach means that the possible states of the DSPL get fully defined at design time. In the open approach, the system is supposed to find new context situations and configurations at runtime. According to [82] and [83], this is usually tackled with an online learning approach with an own MAPE-K loop on top of the first MAPE-K loop. This can be seen as adaptation of the adaptation logic or as self-improvement [16].

3. Research Methodology

Research methodologies propose a structured way of pursuing a research project. In the field of information systems, the design science research methodology as proposed in [84] can be used. It follows the principle of an iterative process for solving a specific problem. This general principle can be defined more specifically by proposing a particular process instance others can apply for their research, such as [22] or [85]. This thesis follows the design science research method process of Peffers *et al.* [22], as the proposed steps can easily be mapped to the different steps of developing software artifacts, and this fits this thesis' objective to develop a runtime environment for adapting communication systems.

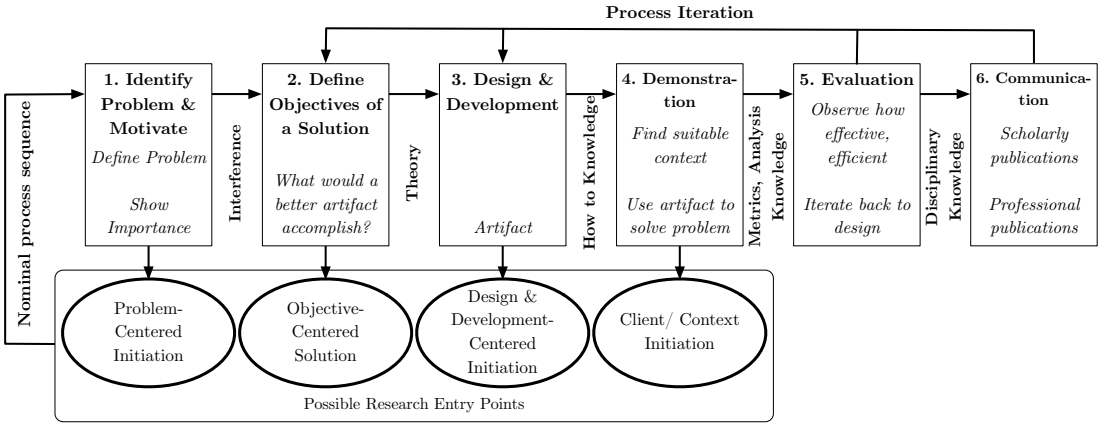


Figure 3.1.: Design science research methodology by Peffers *et al.* [22].

The design science research method of Peffers *et al.* consists of six steps, shown in Figure 3.1. The process allows for having different research entry points. There are problem-centered, objective-centered, design & development centered, and client/context initiations possible. Either way, after the instantiation, the process follows the six steps. This thesis begins the process with a problem-centered approach, as the problem is that it is difficult to make communication systems adaptive, especially for non-SAS experts.

In the beginning, the process starts with the *Identify Problem & Motivate* step. In this step, the problem that is supposed to be solved is defined. Additionally, this step shows the relevance of the problem. Ideally, the problem can be divided into smaller sub-problems for better understanding. In the case of this thesis, the problem is motivated in Chapter 1, including the proposition of research questions. Also, the research gap is identified more specifically as part of Chapter 4 outlining requirements and Chapter 5 presenting related work.

The second step is called *Define Objective of a Solution*. Based on the problem definition in the first step, this step specifies the overall objectives of the design science research process. This includes evaluation criteria so that a comparison between the new artifact and already existing artifacts is possible. The functional and non-functional requirements of this thesis' artifact are presented in Chapter 4.

Step 3 is the core step for all design science research processes–*Design & Development*. This step includes the design and creation of the artifact. Depending on the problem and the objective, the result of this step can be "*constructs, models, methods, or instantiations*" [22, p. 55] or even "*new properties of technical, social, or informational resources*" [22, p. 49]. Besides the actual artifact development, this step includes the specification of the needed functionality and the architecture of the artifact. The artifact of this thesis is REACT, including different model-based specification possibilities. REACT is presented as part of Chapters 6 and 7.

The following step is *Demonstration*. This step demonstrates the use of the artifact in the problem domain. According to Peffers *et al.*, it may be possible to, e.g., use experiments, simulations, or case studies in this step. For the demonstration, the use of the artifact and required knowledge about the problem domain is needed. The demonstration of the implementations consists of the application of REACT Core (Chapter 6) in combination with the REACT Loops (Chapter 7). Additionally, the visualization approaches in connection with REACT are demonstrated as part of Chapter 8.

After the demonstration of the possibility to use the artifact in the problem domain, an *Evaluation* is needed for quantifying its benefits. The evaluation method depends on the type of the artifact. Ideally, step 2 defines already some metrics, which can be evaluated here. However, the evaluation step may also include additional quantifiable metrics. In this step, it is also possible to compare

existing solutions with the new artifact. As the evaluation determines how well the artifact performs according to the defined metrics, it is possible to iterate back to step 2 or 3. Evaluations and feasibility studies are presented as part of Chapters 7 and 8. This includes a comparison with existing approaches from the literature. The discussion considering the fulfillment of the different requirements presented in Chapter 4 takes place in Chapter 9.

The last step is to communicate the results. The step *Communication* includes a description of the artifact, its importance, novelty, as well as the effectiveness in the evaluation. This can be done as part of any publication.

As indicated by the arrows in the process figure, it is possible to iterate back to either the objective definition or the artifact creation. The communication of the results is handled with this thesis and as part of the publications related to it.

Figure 3.2 shows the applied methodology as well as the iterations and the corresponding publications. Beginning with the problem and motivation, as presented in Chapter 1, it is complex to make communications systems adaptive. Based on this fact, the objective of a solution is a model-based runtime environment. Using this objective as foundation, multiple iterations took place. These iterations consist of the design & development, demonstration, and evaluation of different adaptation logics, REACT itself, and the two visualization approaches CoalaViz and EnTrace. Each iteration resulted in a publication, which is identified by its respective conference name in the communication step. The publications are ordered by date, with the most recent one at the bottom.

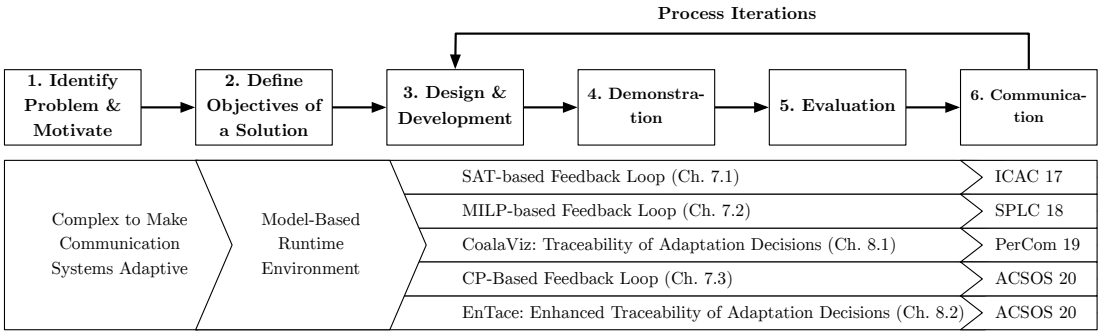


Figure 3.2.: Iterations following the design science research methodology of Peffers *et al.* [22]. Ch.: Chapter.

4. Requirements

Following the methodology of Peffers *et al.* [22] described in the previous section, as well as building on the observations, motivation, and research questions presented in the introduction, this section outlines requirements for the approach of this thesis. Hence, pursuing the process for requirement engineering proposed in [86], this section identifies stakeholders and corresponding functional and non-functional requirements. Parts of this chapter are based on [11][1] and [87].

4.1. Stakeholders

Two scenarios necessitate the introduction of adaptivity to a communication system. First, an adaptive communication system may be developed from scratch. In the second case, the objective is to add adaptive capabilities to an existing communication system for, e.g., improving the system's performance at runtime.

In both scenarios, a software development process, also known as software development life cycle, is followed. There are many different software development life cycles available [88]. Such a life cycle determines how to conduct the software development process. Typically, a software development life cycle provides a list of consecutive steps a development team should follow when creating or changing software components. Therefore, this paradigm can be applied to the two scenarios mentioned above. Accordingly, for the stakeholder analysis of this thesis, we use the sequence of development stages presented by Rosove, as they represent the broad steps of a development process [89, 90]. Later, the similar *Waterfall Model* has been introduced, which has a comparable structure, either with or without the possibility of iterations [91]. Rosove's process is shown in Figure 4.1 and follows the steps *requirements*, *design*, *production*, *installation*, and *operations*. Additionally, it is possible to have iterations of the process based

[1] [11] is joint work with M. Breitbach, C. Krupitzer, M. Weckesser, C. Becker, B. Schmerl, and A. Schürr.

on feedback. More modern development processes, such as the various agile development methodologies, consider similar development steps [92]. Hence, the general steps presented by Rosove's process still apply today.

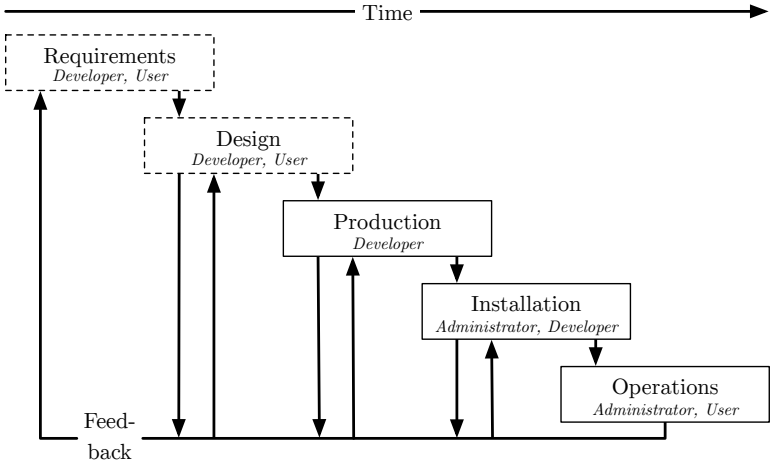


Figure 4.1.: Development stages as presented by Rosove [89, p. 18] with the identified stakeholders per stage. The dashed boxes indicate steps, which are not directly addressed by this thesis' approach.

Starting with the requirements step, this involves the system *developers* conducting the requirements engineering process as well as the (potential) *users* of a system. The users are needed in the process of defining the requirements of the (new) system. Hence, these two groups represent the first stakeholders. Next, according to Rosove [89, Chapter 4], the design of a system also includes users of the (new) system for evaluating it as well as developers. As shown by the dashed boxes in Figure 4.1, these two stages are not directly addressed in this thesis. Subsequently, the production step represents the actual development of the system. In this case, the system developers are involved exclusively. Considering the goal of this thesis, the system developers are supposed to use the artifact of this thesis to either implement a new adaptive communication system or add adaptive capabilities to an existing system. Then, there is the installation step, which is executed by *administrators* in cooperation with the developers. In this step, the developed system is deployed on the available computing resources. The artifact of this thesis should support the administrators and system developers in deploying the final system distributedly in a simple manner. Finally, the administrators have to operate and monitor the deployed system at runtime in the *operations* step.

In this step, the users are involved as well, as they utilizes the running system. In the following, each stakeholder group is defined more clearly, focussing on the development of adaptive communication systems.

System Developers: A system developer builds the communication system and represents a system developer in the corresponding field. Hence, considering SAS, the system developers engineer the managed resource, which gets adapted to changes at runtime. System developers do not necessarily know about SAS engineering but are experts in their respective fields of communication systems. These can, e.g., be overlay networks or software-defined networking (SDN). If system developers want to make their communication system adaptive, they want to change as little as possible in their system for saving developing effort and, subsequently, time and money. By using an available framework or runtime environment, changes are mainly needed for integrating the connection between the managed resource and the (external) adaptation logic. If the communication system already exists, system developers want to be able to add adaptive behavior to the existing system without the need to redevelop everything. As the communication system can be developed in different programming languages, system developers require a high compatibility with different programming languages from a framework supporting them. When thinking about the adaptive behavior, they need an abstract way of defining how the adaptation logic reconfigures their system at runtime without the need for building an adaptation logic from scratch. For conducting this task, it helps them if the technologies for modeling the adaptive behavior are related as closely as possible to well-known software engineering methodologies. At development time, they want to make sure that it is as easy as possible to check the system's adaptive behavior for correctness and goal accomplishment.

Users: In general, the users of communication systems demand a working system with a high performance. In the case of communication systems, the users can be end-users utilizing a network using their smartphones or service providers using a network for providing their service. In both cases, the network's performance can be characterized using different metrics such as available bandwidth, response time, or—more general—the quality of service (QoS) or experience.

Administrators: Administrators of a communication system need to deploy and manage the (running) system together with the system developers. Considering

deployment, this task should be as easy as possible. This includes a simple way to specify the deployment of the final system as well as a mostly automatic procedure to start the distributed components of the adaptive communication system. Looking at the management possibilities of the running system, the administrator must be able to observe and understand the behavior of the adaptive system. In case the adaptive system behaves not as intended, an administrator must be able to influence or even change the adaptive behavior of the adaptive communication system at runtime.

4.2. Functional Requirements

Based on the previous observations and the analysis of the stakeholders, this thesis' objective is to address the following functional requirements. As a notation, this thesis uses the abbreviation R_F with an index for the functional requirements.

***R_F1*—Support for all Self-* Properties:** As described, already in the field of IoT, the heterogeneity of SAS is high. Additionally, SASs (can) support up to four self-* properties. Developers should be able to implement either self-* property in any combination with any kind of managed resource. Accordingly, the first functional requirement demands the possibility to support possibly all self-* properties. This generic approach also targets *RQ1*. Providing this functional requirement makes it possible to use the final artifact for the entire self-management capability without limitations.

***R_F2*—Ready-to-Use Decision Engine:** Considering *RQ1* and *RQ2*, which aim at supporting system developers, a solution for adapting communication systems should provide a ready-to-use decision engine. If no decision engine is present, a system developer is required to integrate one manually, resulting in implementation effort and the need for SAS and particularly planning knowledge.

***R_F3*—Multi-Language Support:** As legacy systems are heterogeneous and typically written in many different programming languages, for supporting system developers (cf. *RQ2*), the runtime environment should functionally support multiple programming languages. Also, this enables developers to use their preferred programming languages when developing new systems.

***R_F*4—Language-Independent Predefined Interfaces:** For connecting a managed resource to an adaptation logic, e.g., predefined interfaces defined in an Interface Definition Languages (IDLs), can be used. Accordingly, approaches such as CORBA [93, 94] enable to specify interfaces in a programming language-independent way. Then, specific bindings for programming languages can be generated. This requirement targets *RQ2* and is connected to R_F3, as it supports system developers in using their favorite programming language for engineering a new system or easily support existing heterogeneous legacy systems.

***R_F*5—Support for Existing Systems:** In order to achieve a broad and easy applicability (cf. *RQ1*) and possibilities to add adaptive behavior to existing systems, this thesis' approach should support legacy systems. Hence, it should not be necessary to write completely new applications or systems. Instead, this thesis' approach should support system developers (cf. *RQ2*) in making their legacy systems adaptive.

***R_F*6—Development Process:** Even the best framework or middleware does not help in making systems adaptive if it is unclear how to use it. Hence, this constitutes the requirement of a clearly defined development process for system developers addressing *RQ2*. By tackling this requirement, the runtime environment is able to allow developers to follow a process for efficiently using the approach.

***R_F*7—Distributed and Decentralized Feedback Loops:** Considering *RQ1* and according to the installation step in Figure 4.1, for targeting communication systems, the framework inherently must support distributed managed resource deployments. Additionally, the feedback loop itself should be able to run in a distributed way supporting MAPE-K distribution patterns (as presented in [15]), which increases scalability and deployment options.

***R_F*8—Runtime Monitoring and Modifications:** In order to make sure everything works as specified, the framework should enable to monitor the adaptation behavior of the system as part of the operations step. Additionally, when the deployed system is in the operations phase, changes in the execution environment or requirements can also foster the need to influence the goals of the adaptive system or even change the deployment at runtime. Hence, this functional requirement, addressing *RQ3*, targets the ability to monitor and update a system's objectives and deployment when it already runs.

4.3. Non-functional Requirements

Besides outlining functional requirements, the objective of this thesis is to address the following non-functional requirements. As a notation, this thesis uses the abbreviation R_{NF} including an index for the non-functional requirements.

***R*$_{NF}$1—Generalizability:** An approach allowing to make any communication systems adaptive must provide high generalizability. This induces no use case-specific architecture or implementation, reusable interfaces, and specification capabilities. High generalizability leads to broad applicability in many cases.

***R*$_{NF}$2—Simple Specification:** For targeting system developers in the field of communication systems, the technique for specifying the adaptation behavior must be as simple as possible. This means that a universally applicable and small set of easy-to-use concepts is beneficial for the application of this thesis' framework. This also includes that system developers should be able to reuse existing software engineering knowledge for specifying the adaptation behavior.

***R*$_{NF}$3—Performance:** As with every software, the runtime environment as part of this thesis should have a high performance. Possibly, the performance should allow using the approach in systems where fast adaptations are needed. This helps in delivering every user the desired QoS.

***R*$_{NF}$4—Reusability:** As a framework should increase the development speed by providing structures and interfaces, the reusability must be high. If the overall reusability is low, a framework is not used by developers, which renders it useless. Thus, the developer should be able to reuse the framework's facilities as much as possible, reducing the development effort and increasing the development speed.

***R*$_{NF}$5—Flexibility:** Other than static software systems, SASs are exposed to constant change. Hence, the feedback loop itself should be flexible as well, e.g., in the case of changing requirements. Flexibility is connected with R_F8 proposing runtime modifications and determines the degree of possible modifications such as moving or replacing MAPE components or changing the knowledge of a SAS.

***R*$_{NF}$6—Extensibility:** Finally, the extensibility of this thesis' artifact should be high. A high extensibility makes sure that system developers can extend the runtime environment with custom logics or algorithms without the need to adjust the available structures and services in any way.

5. Related Work

This section outlines the related work of this thesis. First, Section 5.1 presents the used classification approach for comparing the subsequent related works. Then, Section 5.2 gives an overview of related work in the field of frameworks and implementation approaches for engineering self-adaptive systems. These approaches are created from the perspective of SAS research without taking communication systems specifically into account. As this thesis aims at making communication systems adaptive, Section 5.3 presents the related work in the field of Autonomic Networking as the second stream of research, which aims at similar goals as Autonomic Computing in the communication systems domain. Finally, the last section uses the classification presented in Section 5.1 for discussing the results and for summarizing the research gap this thesis addresses. This chapter is based on [11].

5.1. Classification

This section outlines the classification for comparing the related works presented in the following sections. The categories of the classification depicted in Figure 5.1, which are based on the previously presented requirements, aim at making it possible to compare different approaches for engineering networked self-adaptive systems. Hence, the requirements from Chapter 4 are shown in combination with the different categories. The top-level categories consist of *Adaptation Capabilities*, *Development Support*, *Deployment*, and *Evaluation Capabilities*.

For comparing the adaptation capabilities, the first category determines if an approach supports all four self-* properties. A generic framework or runtime environment for developing SAS is able to support all four properties. Only by supporting all four properties, a complete solution providing self-management capabilities, as defined in Section 2.1, is possible. This category directly maps to the functional requirement *Support for all Self-* Properties* (R_F1), as well as

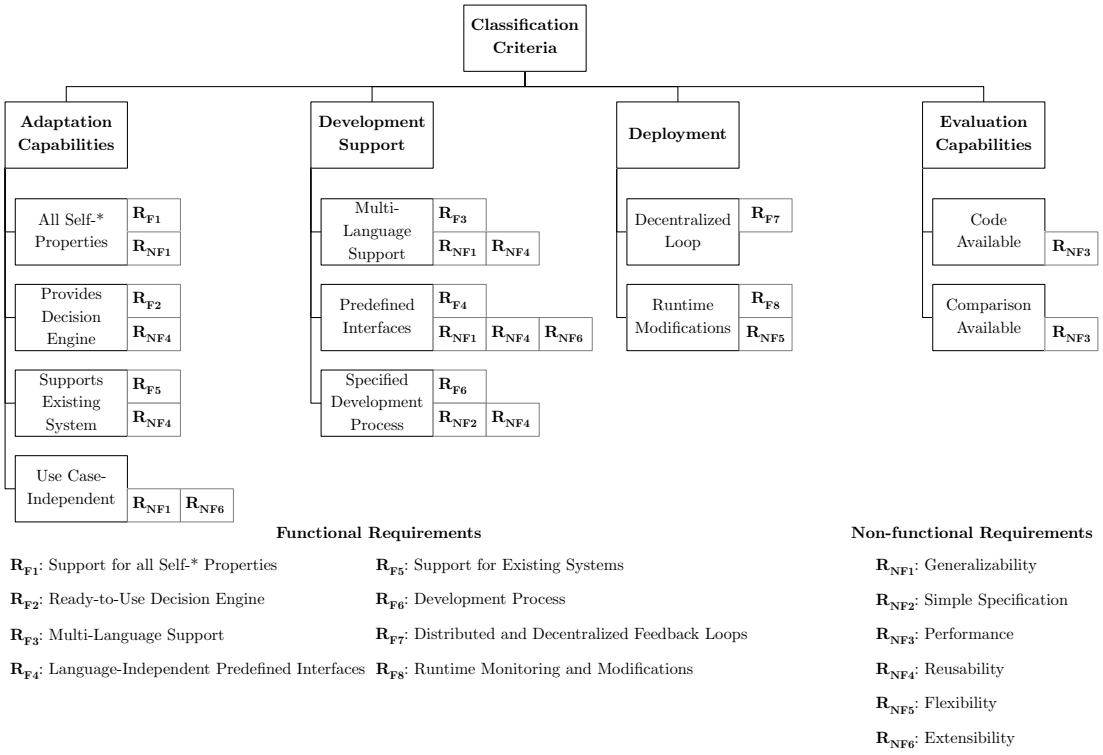


Figure 5.1.: Classification criteria and mapped requirements, presented in [11].

the non-functional requirement $R_{NF}1$ aiming at generalizability. The following category, *Provides Decision Engine*, determines if a ready-to-use decision engine is provided. If this is not the case, a system developer must implement the logic, increasing development time and creating the need for SAS expertise. This category directly maps to functional requirement R_F2 and non-functional requirement $R_{NF}4$ involving reusability. Next, the category titled *Supports Existing System* checks if an approach can be applied for an existing system or if it is only applicable for integration into newly developed systems. This maps to requirement R_F5 (Support for Legacy Systems). Also, non-functional requirement $R_{NF}4$, reusability, can be taken into account. The final category, *Use Case-Independent*, checks if an approach can be used independently of a specific use case or managed resource. This maps to the non-functional requirement $R_{NF}1$, aiming at generalizability. Additionally, if the system is extensible, it can be adjusted for new use cases ($R_{NF}6$).

The following top-level category handles the development support of an approach. First, for development support, the category indicates if an approach supports

multiple programming languages (functional requirement R_F3 and non-functional requirements $R_{NF}1$ and $R_{NF}4$, reusability). However, it is not only important that a framework can be used with multiple programming languages, but it also helps if it features predefined interfaces, which can be reused (functional requirement R_F4 and non-functional requirements $R_{NF}1$ and $R_{NF}4$). Also, this increases the extensibility ($R_{NF}6$). Finally, an overall development process a system developer can follow helps in applying a framework or an approach (functional requirement R_F6 and non-functional requirements $R_{NF}2$, simple specification and $R_{NF}4$).

The deployment category contains two subcategories: *Decentralized Loop* and *Runtime Modifications*. The former subcategory determines if the loop instance(s) of an approach can be executed in a distributed way as well as if it supports decentralized control. This means that, e.g., the monitor of one feedback loop instance runs on a different host than the rest of the instance (functional requirement R_F7). The latter subcategory describes if the deployment, either in terms of running instances or in terms of the specification of the adaptive behavior, can be changed at runtime (functional requirement R_F8, runtime modifications and non-functional requirement $R_{NF}5$, flexibility). By that, it describes the possibility of self-improvement [16]. Hence, if runtime modifications are possible, it also is automatically possible to add a higher-level feedback loop for automatically changing another feedback loop, as described in the hierarchical control pattern [15].

The last top-level category is concerned with capabilities for evaluation. First, for a direct comparison of different implementations, it is important that the source code of an approach is available. This makes it possible to, e.g., implement the same case with multiple approaches for measuring and comparing the performance (non-functional requirement $R_{NF}3$). Additionally, this is important when comparing a new approach to an existing one, e.g., in terms of implementation effort and techniques. Second, for a quantitative and qualitative comparison between multiple existing approaches, an already available comparison as part of a research paper is helpful. Performance, as defined in $R_{NF}3$, is one dimension for a quantitative comparison. The dimension allows reusing the same categories for observing differences and similarities between the existing approaches and a new approach.

After the introduction of the classification, the following section presents related work in the field of engineering approaches for SASs.

5.2. Approaches for Engineering Self-Adaptive Systems

Engineering of self-adaptive systems is a prominent research area with a large body of excellent related work that this thesis can build upon. The research landscape has been reviewed in [95]. Several related approaches perform adaptations based on architectural models (e.g., [5, 96, 97]) or specify architecture definition languages for self-adaptive systems (e.g., [98–100]). Other model-based engineering approaches such as [101–104] often use DSPLs [66] with feature diagrams. The models@run.time research stream proposes to use runtime models that represent the system and environment for reasoning [105, 106]. However, all of the approaches mentioned above do not offer an implementation explicitly designed to be used by others. Since the goal of this thesis is to design an approach that aims at high applicability for practitioners and fellow researchers, this thesis focuses on related work aiming at providing an implementation or a framework in the remainder of this section.

ActivFORMS (Active FORmal Models for Self-adaptation) is a model-driven and reusable approach for designing and executing verified adaptation logics [107–110]. It is based on the FORMS modeling approach for specifying formal models of self-adaptive systems [111, 112]. The overall idea is that the developer can provide a verifiable model to ActivFORMS, which gives guarantees considering the correctness of the adaptation behavior beforehand and executes the specified adaptation logic. For this, ActivFORMS provides generic templates, which can be used with any verifiable modeling language that is able to be executed as well. Additionally, the approach continuously verifies at runtime whether the current instantiation is able to achieve the specified adaptation goals. As verification approach, timed automata in combination with statistical model checking are used. One requirement for deployment is that the managed resource is already completely instrumented with probes and effectors. There is no guidance or constraint on how the managed resource is connected to the adaptation logic. At runtime, ActivFORMS supports evolution by using explicit interfaces for updating the models. This allows for runtime modifications and self-improvement. Based on ActivFORMS, Weyns *et al.* created a specific instance named ActivFORMSi,

which is available for download[1] [110]. It uses the UPPAAL [113] tool suite for specification and verification of the used models. As verification is rather costly, the authors mention that ActivFORMS is only suitable if the adaptation logic has enough time for making a decision. This limits the applicability in (very) dynamic systems. It is a generic approach targeting all self-* capabilities, provides a decision engine (as part of ActivFORMSi), supports existing systems, which, however, need to be already instrumented, and is use case-independent. Weyns *et al.* specify a development process as part of their publications, and it is possible to conduct runtime modifications. The source code of a prototype is available.

Cetina *et al.* aim to adapt pervasive systems using an SPL-based approach [114]. Their approach focuses on the reaction to changes in the managed resource's infrastructure and is based on previous work presented in [115]. Changes in the system resources are limited to adding or removing a resource. Hence, the approach only supports self-configuration and self-healing. The authors follow the models@run.time approach [105, 106] and use multiple models for separating the problem and solution space. This is achieved by using feature models and realization models for the problem, as well as component and structural models for the solution space. For adaptation, in case a resource gets added, the system proposes the activation of additional features, which the user has to confirm. If a resource gets removed, the system automatically adjusts its configuration. The approach provides a decision engine and is able to be incorporated into existing pervasive systems. Other than that, it is rather limited and focuses on a subset of the classification categories.

EUREMA (ExecUtable RuntimE MegAmodels) is a model-driven approach for specifying and executing feedback loops [116, 117]. It uses the Megamodel approach [118, 119] consisting of multiple models with mappings between them for relating them as foundation. EUREMA provides a domain-specific modeling language for defining two types of diagrams, namely feedback loop diagrams (FLD), and layer diagrams (LD). The FLDs are used for specifying the internal (runtime) model and data flow of the MAPE activities. LDs are concerned with the architecture of MAPE loops, which are specified using FDLs. This enables specifying hierarchies of MAPE loops and defines how loops are connected to each

[1] `https://people.cs.kuleuven.be/~danny.weyns/software/ActivFORMS/`, accessed 2020-12-08

other as well as to the managed resource. Also, the LD-based models determine how multiple feedback loops are coordinated. The specified models can be used by the EUREMA interpreter to be executed directly. Besides the FLDs and LDs, a developer also has to provide runtime models, adaptation models specifying the variability space of the managed resource along with evaluation criteria for determining if an adaptation is needed. Additionally, EUREMA also needs a so-called causal connection model defining how the managed resource is connected to the formerly mentioned runtime models. EUREMA can be applied for modeling all self-* properties, can be used in combination with existing systems without limitations on the use case, and enables to update the models at runtime. However, EUREMA only provides high-level modeling capabilities without reusable MAPE components. These MAPE components have to be provided in the first place for applying the EUREMA approach.

FESAS (Framework for Engineering Self-Adaptive Systems) is a generic approach focussing on code reuse and simplified exchange of adaptation logic components [19–21]. FESAS uses the MAPE-K feedback loop as a template for providing an instance of an adaptation logic. The approach defines two stakeholders in the system at design time. First, the role of a system developer is to write the actual logic components based on FESAS' reusable structures provided by its reference implementation. The finished components are stored inside the FESAS repository. Second, system designers are able to define the (deployment) configuration using the previously developed logic components. This includes selecting the logic components targeting a specific use case and the specification of distribution patterns. At runtime, the FESAS middleware uses the FESAS repository and the configuration files for deploying the system. The available Java-based reference system uses a publish/subscribe system for automatic communication between the MAPE-K components [20]. FESAS provides an Eclipse-based IDE for the development of the logic components as well as for designing the deployment configuration [21]. This simplifies the proposed development process. Summarizing, with this use case-independent framework, FESAS allows to target all self-* properties and is able to be connected to existing systems. As it only provides the scaffold for feedback loops, there is no ready-to-use decision engine. This means the complete development of the feedback loop is the developers' responsibility. The implementation of FESAS is only available in Java. FESAS provides

predefined interfaces and specifies a development process. For deployment, it can be instantiated in a decentralized way, supports self-improvement, and the development environment is available as public download[2].

Genie [120] uses the reflective middleware approach Gridkit [121] in combination with a domain-specific language for adapting component-based systems . Using the developed domain-specific languages (DSLs) named OpenCOM DSL [122] and Transition Diagram DSL, it is possible to specify the structural variability of components as well as the context variability. The OpenCOM DSL is described as "*Architecture Description Language (ADL) with generative capabilities*" [122]. This DSL is used for the generation of the component instances and configurations. Reconfigurations are represented using policies, which get evaluated in case of changes in the context models. These reconfiguration policies are generated by Genie automatically based on an instance of the Transition Diagram DSL. To summarize, Genie is generic considering all self-* properties, provides a decision engine, and also supports existing component-based systems. However, it is not use case-independent, as it focuses on component-based systems. Also, Genie does not support multiple languages and does not propose predefined interfaces. The authors specify the process a developer has to follow, and Genie supports the introduction of updates at runtime.

GRAF (Graph-based Runtime Adaptation Framework) is another model-based approach for developing self-adaptive systems [123]. Reflection-based runtime models are used for representing the managed resource. An adaptation manager senses changes in the runtime models and adapts the managed resource by changing them. GRAF uses code injection based on aspect-oriented programming for the connection between models and managed resources. For specification, GRAF uses the TGraphs tool [124] for graph-based modeling. The graphs are translated into rules, which get evaluated by a provided rule engine. GRAF is a generic approach, which can be used for all self-* properties, supports existing Java-based systems, and is use case-independent. Additionally, it provides predefined interfaces as well as a process for connecting a managed resource.

HAFLoop (Highly Adaptive Feedback control Loop) particularly aims at providing a reusable framework for adapting the elements of a feedback loop itself [125].

[2] `https://fesas.bwl.uni-mannheim.de`, accessed 2020-12-08

According to the authors, adapting the loop elements is, e.g., useful for reconfigurable monitoring [126]. HAFLoop is based on the FESAS adaptation logic template [9], which specifies the structure of a MAPE component independent of an actual implementation. The authors extended the template with capabilities for explicitly adapting a MAPE component's parameters and structure, e.g., for allowing self-improvement. Based on the extended template, the authors implemented a Java-based prototype. The approach is evaluated in a self-driving car scenario as well as in an IoT wireless network for adapting the monitoring component by, e.g., enabling or disabling monitoring capabilities in response to changes in the battery life. Overall, HAFLoop provides a generic framework for Java-based systems with a focus on the adaptation of the feedback loop. Like FESAS, it can be used for all self-* properties, supports existing systems, and is use case-independent. Further, interfaces are predefined, there is a process for connecting a managed resource to HAFLoop, and its source code is available[3].

Kinesthetics Extreme (KX) is a Java-based system using behavioral/architectural models, which focuses on the idea of adding autonomic behavior to legacy systems [127–131]. The reference architecture of KX contains four components: sensors, gauges, controllers, and effectors [132]. Publish/subscribe-based event busses between the different component types of the reference architecture constitute the communication facility inside KX. So-called probes are the managed resource-specific sensors forwarding their data using a proposed "Smart Events" XML format to gauges [129]. The gauges can preprocess the sensor data by applying filtering or aggregation techniques. It is also possible that probes rather send raw data, which does not follow the proposed XML format. In this case, gauges also can transform raw data to the XML format using probe- or managed resource-specific transformation modules [129]. As the authors also consider distributed managed resources, probes and gauges can also run distributedly. For controlling a system, the authors proposed a workflow engine called Workflakes [133]. Controllers are connected in a point-to-point fashion to specific effectors. Execution is achieved by the effectors deploying so-called Worklets, which are Java-based mobile software agents executed on the managed resource [129]. The agents run inside a Worklet Virtual Machine, which translates Worklet actions into managed resource-specific actions. Hence, each managed resource has not only to

[3] `https://github.com/edithzavala/loopa`, accessed 2020-12-08

provide effectors. It also executes the virtual machine providing access to its own internals [127]. With the generic workflow engine as the decision engine, possibly all self-* properties can be achieved. KX explicitly targets existing legacy systems and is use case-independent, as the different evaluations show [129, 130, 134]. It supports decentralized loops and the modification of the deployed KX components at runtime.

Malek *et al.* propose an architecture-driven framework targeting mobile applications [135]. Their approach describes a complete development process using a model-based specification and verification at design time, runtime analysis, and a middleware-based execution environment. The modeling approach called XTEAM is used to represent the structure and behavior of a system, to map architectural elements to actual hosts, and for the analysis capabilities, which are context-aware. As XTEAM is a meta-modeling approach, it also can be extended with custom concepts. As the technique for the analysis of the modeled concepts, mixed-integer linear programming is used. The proposed middleware [136] provides the runtime environment that can be used for adapting the system that runs inside of it. As the middleware is available as Java and C++ versions, this approach can be used with managed resources using these programming languages. However, a managed resource must be written explicitly for the middleware, which renders the approach unusable for existing systems. Besides, it supports all self-* capabilities, provides a ready-to-use decision engine, is use case-independent, provides interfaces and a development process, and is able to be modified at runtime.

MOSES (MOdel-based SElf-adaptation of SOA systems) is an approach aiming at adapting service-based systems for achieving certain QoS goals [137]. As MOSES targets systems using service-oriented architectures (SOA), it composes the available services and coordinates multiple services providing the same functionality. A linear programming-based optimization engine is used for planning adaptations, which combines the current service selection with the specified service-level agreement (SLA) goals for finding valid reconfigurations. Since the approach focuses on services, only a limited number of context attributes, consisting of response time, reliability, and cost, are monitored and used for adaptation decisions. A Java-based prototype of the system running in a centralized manner has been developed. Considering the comparison categories, MOSES only provides a decision

engine and can be used with existing systems. Other than that, the approach is tailored towards its specific use case of service-oriented software systems.

MUSIC is a development approach for engineering ubiquitous adaptive applications [12, 13]. The main goal is to provide developers with means to develop new applications providing possibilities for parameter and component adaptation in a component-based way. In order to achieve the adaptation at runtime, a middleware containing a feedback loop following the MAPE-K architecture monitors the context of the applications and adapts running applications accordingly. As a unique feature, there exists a master and a slave version of the middleware. Devices with low computational power, such as handhelds, can execute the slave version, allowing machines executing the master version to adapt the handheld. Hence, these low-powered devices are used as sensors and effectors. The variability of the managed resource is specified using a custom UML-based modeling notation. This modeling approach is also used for specifying the context, which should be monitored, as well as the QoS properties MUSIC should achieve. For avoiding oscillation, MUSIC suspends further adaptations after an adaptation for a short time [13]. The time delay can be changed at runtime. MUSIC also provides an Eclipse-based IDE called MUSIC Studio besides a prototype of MUSIC itself, which runs on top of the OSGi (formerly known as Open Services Gateway initiative) component framework for Java. Unfortunately, although the authors state that MUSIC has been developed as open source software, the MUSIC Studio and the source code of MUSIC itself are no longer available. MUSIC provided a reusable middleware with support for all self-* capabilities and a decision engine. It is not dependent on a single use case, features predefined interfaces and a development process, and enables runtime modifications.

Preisler *et al.* propose a middleware, including a description language-based approach for adapting new and existing systems [138]. The description language for coordination of the adaptation logic is based on XML. The authors use a component-based architecture, which includes the possibility to define dependencies in a service-oriented way. An application using the available distributed components is coordinated using coordination services consisting of a monitor interface connected to a component and an executor interface for adapting a component. As infrastructure and for allowing communication between the distributed components and coordinators, a service bus is applied. For implementing

the approach, the Java-based Jadex Active Component framework[4] is used. The approach does not limit its use considering self-* properties, provides a decision engine for the XML-based coordination language, supports existing systems, and does not focus on a specific use case. Additionally, the authors specify a set of development steps needed to use their approach.

Rainbow is a prominent architecture model-based approach [17, 18]. The goal of Rainbow is to provide a reusable framework for all different kinds of managed resources. The approach uses the Stitch language [99] for specifying the adaptation behavior. For this, the language provides support for specifying strategies consisting of multiple tactics. The language also provides possibilities for checking if an adaptation strategy has been executed successfully, including the possibility to set a time in milliseconds when this check should be performed. The managed resource is represented using the Acme language [48], which is an ADL. In order to model an architecture, the AcmeStudio [139] can be used. The architecture of Rainbow heavily uses the architecture model. Using the connected probes in the managed resource and gauges, which aggregate and preprocess raw probe data, Rainbow updates the instance of the architecture model. Then, the model is evaluated periodically according to the specified constraints, including non-functional properties. If a problem has been detected, a strategy for improving the problematic property is selected and executed via effectors. The connection between the feedback loop and the managed resource is provided using a translation infrastructure for mapping managed resource specifics to the architecture model and vice versa. Overall, Rainbow is a sophisticated approach allowing to target all self-* properties. It provides a model-based decision engine, supports existing systems, and is use case-independent. Additionally, it provides reusable interfaces, the code is available[5], and it has been compared to Zanshin [140] (see the following description of the approach) in [141]. The comparison of Rainbow with REACT as part of [11] is presented in Chapter 7 in this thesis.

REFRACT (REconfiguration-based FailuRe AvoidanCe Technique) is an approach extending Rainbow with specific components and algorithms aiming at failure avoidance in software systems [142]. In order to achieve failure avoidance in a distributed way, REFRACT proposes a MAPE loop on a central server and a

[4] `https://www.activecomponents.org`, accessed 2020-12-08
[5] `https://github.com/cmu-able/rainbow`, accessed 2020-12-08

MAPE loop on clients. The clients execute a software, representing the managed resource. In [142], the Firefox browser is used as managed resource. In case a failure is detected on a client executing REFRACT, the failing configuration of the managed resource, represented by a feature model, is sent to the server. The server tries to reproduce the problem, plans a so-called guard, e.g., limiting a value range of a parameter for avoiding the failure in the future, and distributes the guard to all clients. The clients with a problem can either wait for the server response or try out different configurations for finding a solution locally. Clients without a problem receive the new guards from the servers and can use them to proactively avoid failures in the first place. REFRACT only focuses on self-configuration and self-healing. It provides a decision engine, and it is capable of being added to existing systems. As the server changes the adaptive behavior on the clients, REFRACT supports runtime modifications of its feedback loop.

SASSY (Self-Architecting Software SYstems) is another approach dealing with the reconfiguration of service-oriented architectures [14]. For the specification of the reconfiguration behavior, a "visual activity-modeling language", based on the Business Process Modeling Notation [143], is used [14]. This model can be applied for automatically generating a so-called System Service Architecture (SSA), which represents a model@run.time and consists of a structural and behavioral view. Reconfigurations using the SSA are (re-) compositions of services, which are executed, e.g., based on QoS goals specified by the user. Accordingly, SASSY uses services registered in a repository for the composition. At runtime, the approach also supports changing requirements, which can be modeled for adjusting the adaptations of the managed resource. Summing up, SASSY is a model-based approach, which can be used for all self-management properties, provides a decision engine, supports existing systems, and is not focused on a single use case. Additionally, the adaptation decisions can be modified at runtime.

StarMX is another Java-based framework for developing self-adaptive systems [144]. Like FESAS, StarMX provides an infrastructure for developing adaptation logics, which are composed of multiple so-called processes providing MAPE capabilities. A process can be connected to anchor objects, i.e., sensors, effectors, or helping objects. By that, the developer of the processes is supported with many different services providing features such as logging and caching. The adaptation logic consisting of multiple processes can be triggered either periodically or in response

to specific events. For the communication and adaptation possibilities, StarMX uses the Java Management Extensions (JMX) technology. Decision making is done using rules and by incorporating a rule engine. However, the developer has to create all processes manually when using StarMX. Overall, StarMX can be used for all self-* properties, as it is a general framework. It can be connected to existing JavaEE-based systems and is use case-independent. The approach provides interfaces for connecting StarMX to a managed resource, specifies a development process, and the code of StarMX is available[6].

Tomforde proposes an Organic Computing-based approach for implementing SAS in a reusable way [25]. Architecture-wise, the observer-controller pattern is used, where a so-called system under observation and control (SuOC), referring to the managed resource, is managed by a layer 1 controller. The layer 1 controller directly contains a component for self-improvement. For the actual self-improvement approach, a layer 2 controller changes the layer 1 controller using machine learning techniques and simulations. Finally, a layer comprising layer 1 and 2 provides interfaces for monitoring and goal management for users as well as possibilities for coordination with other adaptation logic instances. Each individual feedback loop, however, runs always on one machine. As the observer-controller pattern itself also represents an architectural blueprint like the MAPE-K architecture, no techniques are specified to use by the pattern itself. In the presented implementation, reinforcement learning has been used on layer 1 [25]. On layer 2, evolutionary algorithms have been applied. The presented approach has been evaluated in many different use cases such as traffic control [145–147] or protocol adaptation of computer networks [148–150]. Tomforde's approach supports all self-* properties, provides a decision engine, supports existing systems, and is use case-independent. Additionally, interfaces between the different layers are predefined, there is a development process described, and by directly incorporating learning and a goal management interface for the user, runtime modifications are possible.

The final approach for engineering SASs presented in this section is *Zanshin* [140]. It is a control-based approach, which consists of the possibility to specify functional and non-functional requirements. Zanshin focuses on parameter adaptation, and its prototype is implemented using the OSGi component framework in Java. The requirements can be specified in a model-based way, using a meta-model approach

[6] `https://sourceforge.net/projects/starmx/`, accessed 2020-12-17

based on the Eclipse Modeling Framework[7]. Additionally, ECA rules can be used. The framework focuses on self-healing in the evaluation. However, there is no restriction regarding the other self-* properties. Zanshin provides a ready-to-use decision engine, explicitly supports existing systems, is use case-independent, and provides specified interfaces and a development process. Finally, the source code is available[8] and it has been compared to Rainbow in [141].

5.3. Autonomic Networking

With the emergence of Autonomic Computing proposed in [4], the idea came up to use Autonomic Computing principles in so-called Autonomic Communication [151]. As described in the survey of Dobson *et al.*, the Autonomic Computing goals of combining technology with business objectives for always having functioning systems with low administrative effort is suitable for addressing challenges of modern computer networks [151]. Hence, the fundamental goal of Autonomic Communication is bringing the self-* capabilities of SASs into the networks. More specifically, Dobson *et al.* define Autonomic Communication as follows [151]:

> *Autonomic communications seek to improve the ability of network and services to cope with unpredicted change, including changes in topology, load, task, the physical and logical characteristics of the networks that can be accessed, and so forth.*

The need for Autonomic Communication emerges from the high management complexity of networks, especially when setting up a new network. Accordingly, Autonomic Communication "*seeks to simplify the management of complex communication structures and reduce the need for manual intervention and management*" [151]. Dobson *et al.* mainly focus on decentralization as the central foundation for Autonomic Communication. Hence, networks should form so-called federations without a central instance and without any supervision by a human administrator. In Dobson's survey, for the most part different aspects and challenges of the state of the art are presented and discussed. This includes decentralized algorithms, modeling and handling of context, programming approaches, security and trust, as well as the evaluation of systems.

[7] `https://www.eclipse.org/modeling/emf/`, accessed 2020-12-17
[8] `https://github.com/sefms-disi-unitn/Zanshin`, accessed 2020-12-08

Based on the principles of Autonomic Communication, many approaches have been developed. Broadly, they can be grouped into the architectural categories hierarchical and flat [28]. These groups denote if there are possibilities of hierarchical control or if the Autonomic Networking system consists of nodes with equal rights cooperating in a peer-to-peer fashion. In the following, multiple approaches presented in the survey of Movahedi *et al.* of both categories are outlined [28].

5.3.1. Hierarchic Autonomic Networking Approaches

The Autonomic Internet (*AutoI*) project is a hierarchical, layer-based approach [152–154]. The overall idea of AutoI is to transform the Internet into self-managing virtual resources working end-to-end in heterogeneous infrastructures [153]. Applying this approach results in an overlay network with uniform control and service-based management capabilities. As part of the layered approach, first, all physical resources get virtualized using a virtualization layer on top of the physical networks. On this virtualization layer, the network and management services get instantiated. Next, a management layer is responsible for instantiating the actual MAPE-K-based decision loops in the network. Administrators control this layer by providing high-level policies. An ontology-based knowledge plane captures all knowledge from the other planes and distributes it. Finally, the orchestration plane handles conflicts and negotiates between multiple network entities. AutoI supports all self-* properties, provides a decision engine, is use case-independent, and contributes predefined interfaces. Additionally, it enables the deployment of decentralized feedback loops with the capability of runtime modifications.

Context-Aware MANETs (*CA-MANETs*) is the next policy-based approach focussing on the context-aware self-configuration of MANETs [155–157]. The hierarchic structure consisting of three tiers contains cluster nodes at the leaves of the hierarchy, cluster heads, as well as so-called manager nodes. All nodes have the possibility to enforce policies and to store different context information in a use case-specific context model [156]. The context information is gathered hierarchically and passed from the simple cluster nodes via the cluster heads to the manager nodes. This enables to enforce policies on the cluster as well as on the network level. CA-MANET is a use case specific approach providing a decision engine and decentralized feedback loops only.

DRAMA (Dynamic ReAddressing and Management for the Army) is an agent-based Autonomic Networking approach focussing on the management of Mobile Adhoc Networks (MANETs) in the US army [158–161]. DRAMA is a hierarchic policy-based approach with a global policy agent as well as clusters of nodes consisting of the cluster head being a domain policy agent. Every other member of the cluster is in the role of a local policy agent. In DRAMA, there is a focus on self-configuration of the MANET nodes [28] based on high-level policies set by the global policy agent, which is a centralized node [159]. A policy can specify QoS goals or rules for moving services to a different machine in case of bad performance. Hence, e.g., the routing protocol can be switched, or a DNS server can be moved from one node to another [159]. The hierarchy of the nodes and clusters is observed for monitoring as well, resulting in local policy agents reporting to their cluster head, which aggregates the data forwarding it to the global policy agent accordingly. DRAMA uses an adaptive middleware as the foundation [162]. It supports all self-* capabilities, provides a decision engine and interfaces, allows for decentralized feedback loops and runtime modifications.

Unity focuses on the management of cloud-based applications and the underlying resources in a multi-agent way [65,163,164]. The management possibilities include self-configuration, self-healing, and self-optimization. In this hierarchic approach, there are applications running on a server cluster controlled by a so-called resource arbiter. The resource arbiter allocates resources, i.e., servers, to applications. For specifying the goals of the system, Unity relies on a policy repository as well as on SLA goals. Unity has also been used in combination with utility functions [164]. Applications consist of an application manager controlling the allocated resources and constituting a role similar to a cluster head. All communication between the different applications for coordination is handled via the application managers. As shown in experimental evaluations, the central adaptations are the reallocation of resources as well as healing from breakdowns [163]. Every element of Unity is an autonomic element as defined in Autonomic Computing [4], meaning that every server and service manages itself autonomously. The combination of the provided services is used to achieve the specified SLAs cooperatively. Unity supports developers with a decision engine, decentralized feedback loops, and the possibility of runtime modifications.

5.3.2. Flat Autonomic Networking Approaches

The Autonomous Decentralized Management Architecture (*ADMA*) is a flat approach targeting self-configuration of MANETs [165]. Hence, the available nodes as part of the ADMA-enabled MANET reconfigure themselves in a peer-to-peer fashion. ADMA also adapts on the foundation of policies in the form of ECA rules. These rules are provided to at least one node of the MANET directly at the start of the system by an administrator. Then, ADMA distributes the policies using a custom-tailored protocol named Distributed Policy Management Protocol (*DPMP*) [166, 167]. This also enables setting up joining nodes by providing the available policies. Additionally, DPMP permits to distribute monitoring information between the MANET nodes. The nodes adapt themselves based on the policies and the available local and remote monitoring information stored in local repositories. Overall, ADMA provides a decision engine as well as decentralized feedback loops.

The Autonomic Networking Architecture (*ANA*) is a clean-slate project proposing a generic network architecture, which inherently provides possibilities for autonomic behavior [168, 169]. The ANA approach specifies interfaces and abstract entities. The most important abstract entities are so-called compartments, which represent a group of network entities, without defining how the compartment has to work internally. ANA promotes to arbitrarily chain network services, named functional blocks, together instead of enforcing a fixed layered architecture consisting of layers that depend on the others. For that, ANA specifies Information Dispatch Points (*IDP*), which represent address-agnostic connection points attached to functional blocks. The use of IDPs for connecting Information Channels, representing the actual connections, enables to transparently reconfigure the message flows. ANA itself does not provide any reconfiguration mechanism. ANA is generic and use case-independent and provides predefined interfaces. The source code is available[9].

The In-Network Management (*INM*) architecture is another clean slate framework as part of the 4WARD project for engineering the Internet of the future [170, 171]. The overall idea of the INM approach is to deploy distributed networking components as well as management capabilities in the network. These components

[9] `https://sourceforge.net/projects/ana/`, accessed 2020-12-08

follow the service-oriented architecture paradigm by providing different interfaces for management and accessing the service. INM specifies that a component can either have no integrated management at all, integrated management capabilities, which are separated from the functional logic, or inherent management, which is interwoven with the functional logic. For managing components without integrated or inherent management capabilities, specific management components only for managing the former can be created [170]. INM provides a runtime environment for executing the developed components. Multiple components can be composed for achieving complex behavior. A network operator, who can be considered as an administrator, can use a global management interface for providing high-level goals. The INM system provides feedback to the operator using real-time monitoring capabilities. INM supports all self-* properties and is use case-independent. Additionally, it features predefined interfaces as well as the possibility for decentralized feedback loops and runtime modifications.

Cognitive Networks constitute another flat approach for introducing self-* capabilities into networks [172, 173]. The term cognitive refers to artificial intelligence techniques, allowing the system to learn about the best adaptations continuously [172]. Cognitive Networks architecturally follow the idea of the three layers of Kramer and Magee [51] on the node level by having one layer for providing the high-level goals, one layer representing the adaptation logic, and the managed resource titled Software Adaptable Network (SAN). The approach makes sure that it works with partial global information for optimizing the behavior. For achieving this, there is a layer collecting and sharing status information between the available network components. Besides the learning aspect, a specific end-to-end perspective, including optimization along the complete path, delineates Cognitive Networking from other approaches. For specifying end-to-end goals, a Cognitive Specification Language (CSL) is employed. This language maps the goals to the actual mechanisms, which can be adapted. Cognitive networks are not limited concerning the self-* properties and are use case-independent. The approach has predefined interfaces and is able to execute decentralized feedback loops with support for runtime modifications.

5.3.3. Standardization of Autonomic Networking

Besides the presented Autonomic Networking approaches introduced in [28], currently, there is also some effort in standardizing Autonomic Networking principles in the US [174] and in Europe [175]. The current possibilities of autonomic behavior in single network protocols and the research gap of providing autonomic functions network-wide with, e.g., additional coordination and management possibilities are described in [176]. Based on these observations, RFC7575 [174] presents the summarized view of the Internet Research Task Force (IRTF) incorporating the insights of existing works such as [151], [177], and [28]. It contains general definitions and design goals for Autonomic Networking published for informational purposes. Hence, it is currently not yet an official standard [174, 176]. RFC7575 defines an Autonomic Node as a node without the need for any configuration, which can operate on any layer of the networking stack. As RFC7575 focuses on node-level autonomy, there is only a minimal dependency on central instances resulting in decentralization and distribution being fundamental. By that, "*if a problem can be solved in a distributed manner, it should not be centralized*" [174]. However, Autonomic Networking aims at coexisting with existing management capabilities, which might be centralized for administration purposes. Consequently, Autonomic Networking nodes obey their local autonomic default policies with the lowest priority and higher-level network policies with a medium priority. Management policies directly configuring the node, such as direct command-line configurations, simple network management protocol (SNMP), or software-defined networking (SDN) configurations, have the highest priority. The higher-level policies should be as abstract as possible, e.g., by not even exposing the version of the IP protocol used in the network. As feedback for administrators, the network as a whole should be able to provide aggregated reporting. Considering the overall infrastructure, there should be a common architecture for executing autonomic functions. Accordingly, there is the need for an overall control plane for communication and coordination between autonomic functions operating on the IP layer. Additionally, a life cycle represented by a state model should be used for reflecting the (deployment) state of autonomic functions in the network. Finally, the presented ideas are combined for proposing a reference model [178] for an autonomic node. A node should contain Autonomic Service Agents implementing autonomic behavior of specific services or functions. Besides the possibility to

adapt internally, external feedback loops should be able to influence the node as well. Next, an autonomic node should contain an internal self-aware knowledge component and a component for the discovery of other autonomic services and nodes in the network. The node gets completed with a frontend for administrators and external applications for monitoring the state of the network as well as the connection to the Autonomic Control Plane [179] for coordination, discovery, monitoring, and the communication of the feedback loops.

The European Telecommunications Standards Institute (ETSI) also has a process for standardizing Autonomic Networking [175, 180–182]. Their approach is called Generic Autonomic Network Architecture (*GANA*), which constitutes a reference model as well. GANA shall serve as an architecture for bringing self-management into networks in the form of a reference architecture. This reference architecture shall manage all kinds of resources, being "*protocols, stacks and mechanisms*" [175]. By that, GANA directly includes the concept of self-improvement [16] as learning should directly be integrated into the so-called decision engine [183]. GANA provides means for fully decentralized, hierarchical control-based [15], as well as centralized deployments. This enables for horizontal coordination, e.g., representing an end-to-end network path and hierarchical coordination. In the case of decentralized decision-making, the architecture also provides reference points for negotiation between decision engine instances along a network path. In any case, administrators should provide their goals using an ontology-based language for specifying business objectives. However, as GANA only provides a blueprint and not specific implementation details, it is not described which ontology or language would be suitable for this. The specified goals are contained in a Knowledge Plane (*KP*), which is a distributed system within the network. It provides the Overlay Network for Information Exchange (*ONIX*), enabling auto-discovery capabilities with possibilities to query the system actively, as well as a publish/subscribe-based interface. For tackling the heterogeneity of the different network devices, the KP also contains the Model-Based Translation Service (*MBTS*) translating from and to specific devices to the GANA architecture [175]. ETSI considers four architectural levels for the integration of autonomic functionality into networks in ascending order: protocol level, function level, node level, or network level. The protocol level consists of some of the already available protocols, such as OSPF [184]. In this case, the protocol itself is considered to provide some auto-

nomic behavior. GANA rather focuses on the other three levels [183]. A function such as routing is composed of multiple protocols and is managed by a decision engine constituting level 2 [175]. On the node level, GANA concentrates on security, fault management, configuration and discovery, as well as resilience and survivability [175, 183]. These properties are managed in GANA by a node-level decision engine. Finally, level 4 is the network level with the decision engines operating on a slower timescale, which corresponds to the idea of hierarchical control [15]. ONIX and MBTS are functional blocks on the network level themselves. The GANA developers are currently looking for use cases with ideas for applying the presented concepts [185]. Additionally, at some point in time, a UML-based GANA meta-model should be created for specification and deployment [183].

5.4. Discussion and Summary

This section discusses and summarizes the findings of the related work and the categorization of the approaches. Table 5.1 presents an overview of the categorization.

In the following, this section begins with the discussion of the SAS engineering approaches. Summing up the categorization, first, an approach that optimally assists system developers, supports all self-* properties [4] to be suitable for various use cases in communication systems. Second, the integration of a ready-to-use adaptation decision engine, which adapts the communication system based on rules, models, goals, or utilities (see Section 2.3) makes the approach useful for system developers without extensive knowledge about self-adaptive systems. Third, the support for existing systems is essential to integrate self-adaptivity into legacy systems. Fourth, a use case-independent approach is applicable to a wide range of communication systems. We observe that multiple approaches fulfill various of these requirements. Still, not all categories are fulfilled by a single approach. For example, FESAS [21] and HAFLoop [125] provide excellent developer support with reusable MAPE components but do not integrate a decision engine.

The resulting runtime environment of this thesis aims to support the system developer during the development process. Especially in the heterogeneous communication systems landscape, an approach is easy to use if it supports multiple

		Author / *System*	Capabilities: All Self-* Properties	Capabilities: Provides Decision Eng.	Capabilities: Supports ex. System	Capabilities: Use Case-Independent	Dev. Sup.: Multi-Language Support	Dev. Sup.: Predefined Interfaces	Dev. Sup.: Specified Dev. Process	Depl.: Decentralized Loop	Depl.: Runtime Modifications	Eval.: Code Available	Eval.: Comparison Available
SAS Engineering		*ActivFORMS(i)* [107–110]	●	●	●	●			●		●	●	
		Cetina [114]		●	●								
		EUREMA [116, 117]	●		●	●					●		
		FESAS [16]	●		●	●		●	●	●	●	●	
		Genie [120]	●	●	●				●		●		
		GRAF [123]	●	●	●	●		●	●		●		
		HAFLoop [125]	●		●	●		●	●	●	●	●	
		KX [131]	●	●	●	●				●	●		
		Malek [135]	●	●		●	●	●	●		●		
		MOSES [137]		●	●								
		MUSIC [12, 13]	●	●		●		●	●		●		
		Preisler [138]	●	●	●	●			●				
		Rainbow [17, 18]	●	●	●	●		●				●	●
		REFRACT [142]		●	●						●		
		SASSY [14]	●	●	●	●					●		
		StarMX [144]	●		●	●		●	●			●	
		Tomforde [25]	●	●	●	●		●	●		●		
		Zanshin [140]	●	●	●	●		●	●			●	●
Autonomic Networking	Hierarchic	*AutoI* [152–154]	●	●		●		●		●	●		
		CA-MANET [155–157]		●						●			
		DRAMA [158–161]	●	●				●		●	●		
		Unity [65, 163, 164]		●						●	●		
	Flat	*ADMA* [165]		●						●			
		ANA [169]				●		●				●	
		INM [170, 171]	●			●		●		●	●		
		Cognitive Networks [172, 173]	●			●		●		●	●		

Table 5.1.: Overview of related approaches for engineering SAS and in the field of Autonomic Networking, partly based on [11] (Depl. = Deployment, Dev. = Development, Eng. = Engine, Eval. = Evaluation, ex. = existing, Sup. = Support, ● = fulfilled).

programming languages, such as the approach by Malek *et al.* [135]. A vast majority of approaches relies on particular programming languages only, with Java being the most frequently used language. In addition, predefined interfaces, as introduced by the prominent Rainbow framework [17], allow connecting the managed resource easily to the adaptation logic, which is especially important for legacy systems. Rainbow, however, belongs to the approaches [14, 17, 114, 137, 142] that do not specify an easy-to-follow development process.

This thesis argues that an approach that is suitable for large and heterogeneous communication systems must support decentralized control with multiple feedback loops [15]. This typically also encompasses that one feedback loop itself can be separated into several distinct components that may run distributed. Most existing approaches are designed for centralized feedback loops only. As a running system might change over time in an unexpected way, it is helpful to adjust the behavior manually, apply self-improvement [16], or change the deployment at runtime. This holds true for communication systems in particular, where, e.g., new components or subsystems may join or leave the system at any time. In several related approaches [17, 114, 137, 138, 140, 144], the possibility to influence the system already ends with the deployment.

Ideally, the source code of the implementation is publicly available and well documented. This helps to foster further research and enables adoption by system developers in practice. Only a small subset of existing approaches [17, 21, 110, 125, 140, 144] provide an available implementation at present. Moreover, a comparative evaluation with other approaches highlights the merits of the particular approach and gives users guidance to select the proper approach for their respective communication system. Here, only Rainbow [17] and Zanshin [140] have been compared in [141].

Looking at the Autonomic Networking category in the table, we can see that all approaches require a developer to engineer completely new systems. There is no way to include existing communication systems in an Autonomic Networking approach. Also, only a subset of the approaches provide a decision engine or support all self-* properties. We can also observe that multiple approaches specifically target single use cases, with MANETs being a popular case.

Overall, the development support of the reviewed Autonomic Networking approaches is also rather low. There are some approaches providing predefined interfaces, but there is no approach supporting multiple programming languages or specifying a development process. All approaches except for ANA provide the possibility to execute a feedback loop in a decentralized way. Also, 5 out of 8 Autonomic Networking methods allow for modifying the deployment at runtime. In the evaluation category, only the code of the ANA approach is available.

In general, many problems of current, rather static networks and communication systems are planned to be addressed as part of the Autonomic Networking standardization processes. However, looking at the standardization efforts, currently, no implementations of the draft specifications are available yet. This leads to the fact that a classification using the presented categories is not possible.

The analysis of the related work shows that neither the SAS engineering nor the existing Autonomic Networking approaches provide all capabilities specified in the categories, which are based on the challenges defined in Chapter 4. Hence, this identified research gap, together with the motivation to provide a reusable and model-based runtime environment presented in Chapter 1, constitutes the foundation for the runtime environment presented as part of this thesis. The following chapter outlines the system design of REACT Core, which is the foundation for executing ready-to-use REACT Loop instantiations.

6. REACT Core: The Foundation for a Model-Based Runtime Environment for Adapting Communication Systems

The previous chapter presented and compared different approaches for adapting communication systems in the field of frameworks for engineering SASs and the Autonomic Networking approaches coming from the networking perspective. The chapter has shown that no approach fulfills all categories of the taxonomy presented in Section 5.1. This chapter presents the foundation for REACT, a model-based **R**untime **E**nvironment for **A**dapting **C**ommunication sys**T**ems. REACT focuses on providing all features presented as part of the taxonomy categories and aims at fulfilling all requirements presented in Chapter 4. In contrast to self-adaptation frameworks, which offer a standard way to build self-adaptive applications, we refer to REACT as a runtime environment, i.e., a platform that is additionally able to plan and execute adaptations based on user-specified adaptation behavior (see challenges in Chapter 4). Therefore, REACT can directly be applied without the need for system developers to know about SASs and their development.

Following the design science research methodology outlined in Chapter 3, this chapter presents the design and implementation of the foundation of this thesis' system, called REACT Core. REACT Core provides reusables services and structures for executing ready-to-use model-based feedback loops, called REACT Loops. The following sections present REACT Core's main design decisions, reusable components, and implementation details. Then, the optional context management module of REACT Core is presented, including its design, implementation, and a feasibility study. Finally, the last section outlines the development process for applying one of the REACT Loops using REACT Core for adding adaptive behavior to a communication system. This chapter is based on [11, 186][1].

[1] [11] and [186] are joint works with M. Breitbach, C. Krupitzer, M. Weckesser, C. Becker, B. Schmerl, and A. Schürr.

6.1. Architecture of REACT Core

This section introduces the foundation of REACT called REACT Core, which constitutes the infrastructure for the execution of model-based adaptations [11,186]. REACT Core provides multiple facilities for executing model-based adaptations using different models and planners based on (problem) solvers.

6.1.1. System Model

REACT Core proposes a framework for REACT Loops as well as interfaces for connecting managed resources. Potential managed resources in the communication systems domain are overlay networks such as peer-to-peer systems and underlay networks, e.g., in SDN scenarios. However, REACT Core and a REACT Loop can possibly be used in other non-network application domains as well. All REACT Loops follow the MAPE-K architecture. The MAPE components of a REACT Loop use information stored in the knowledge for reasoning. The loop receives sensor information from the managed resource as an input and determines the required adaptations as an output via interfaces.

Figure 6.1 shows the architecture of REACT consisting of REACT Core and generic MAPE components on top of a communication system using a UML-like notation. The interfaces, sensor component, and the knowledge service are generic, internal parts of REACT Core and independent of the use case. The dashed MAPE components are provided by a specific REACT Loop utilizing the foundations of REACT Core. All gray parts in Figure 6.1 are encapsulated in a ready-to-use fashion when applying a feedback loop instance and do not require any programming effort from the system developer. The white boxes represent the model knowledge and the effector implementation that have to be provided by the system developer.

REACT Core aims to be as generic as possible. This increases reusability while always providing the same generic facilities. Hence, the sensor and MAPE-K components can be used to instantiate different model-based REACT Loops. Various instances of the MAPE-K components and the sensor can be distributed on different machines, as the communication between the components is handled by REACT Core. Thus, we achieve high scalability and allow distributed deployments

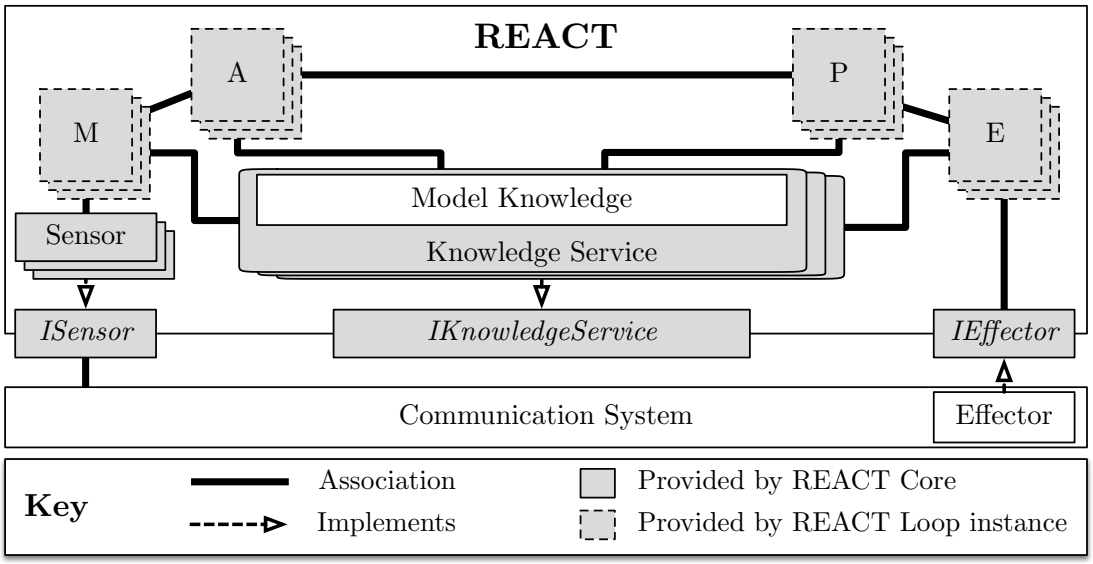


Figure 6.1.: REACT's architecture in a UML-like notation. It consists of one or multiple REACT Loops connected to (an) instance(s) of the knowledge service with model knowledge. The instance of REACT Loop uses the model knowledge to plan and execute adaptations. The managed resource connects to REACT via well-defined sensor and effector interfaces provided by REACT Core. The optional context manager is omitted here [11].

and decentralized control. Following this design, fully decentralized or hybrid patterns, as described in [15], are realizable.

At the beginning of the loop, the ready-to-use sensor of REACT Core implements the `ISensor` interface and simply forwards the received data to the subsequently connected monitor. Then, the sensor data is handled depending on the used REACT Loop. Finally, the result of the loop is handed over to an effector as part of the managed communication system implementing the `IEffector` interface. The knowledge component provides different types of knowledge, which are defined by the domains of the three provided REACT Loop and the corresponding IDL-based interfaces, which will be presented in Section 6.1.2. This design allows developers to easily extend the knowledge by adding new interface methods or implementing a completely custom knowledge component.

Each sensor and MAPE-K component is deployed and set up using key-value-based configurations. These configurations include information about which logic should be loaded, a name, as well as the names of a possible successor and the knowledge

service that should be used. Each MAPE-K component supports multiple ways to set up the actual communication between the loop elements. First, there is the possibility to set up static IP addresses and ports. This capability enables a fixed setup with the disadvantage of higher configuration effort. When IP address and port information are omitted, one of the two automatic set up procedures finds the feedback loop components without additional configuration in the local network. If the setup over the Internet should be automated, a dedicated registry has to be specified where each component registers on startup. In the local network, automatic setup is provided using multicasts. As runtime modification is a challenge that should be supported, REACT Core enables to change the key-value-based configuration at runtime. This enables to modify the deployment of a feedback loop in an easy-to-use and transparent way.

Considering the specification of the adaptive behavior, self-adaptive systems can use models, rules, goals, utility functions, or combinations of the former as decision criteria [95]. Regarding rules and policies, they can be interpreted by humans easily, and an adaptation logic can evaluate them quickly. However, they have a limited expressiveness, and it is difficult to keep track of many rules in the case of larger systems. As also presented in Section 2.3, goals on the other end of the scale are rather abstract. This means there has to be a mapping from high-level goals to low-level goals as well as actual adaptations. Hence, goal-based approaches often need multiple layers of specifications for defining the adaptive behavior (cf. Section 2.3). Utility functions are powerful, as they directly allow measuring how well a system performs. Again, a mapping to actual adaptations is needed here, and it is rather difficult to develop a utility function in the first place. As models provide a sufficient level of expressiveness while being easy to use for system developers, we select a model-based approach for specifying the behavior of REACT-based feedback loops. By creating the models at design time, the system developer tailors the feedback loop to the respective use case. Thus, the system developer is able to integrate self-adaptivity into the managed resource by only providing the models used as decision criteria. These models are then applied by one of the REACT Loop instances. Depending on the required complexity of the adaptation decisions and the corresponding choice of the applied REACT Loop, different models can be employed. For the specific different model types and REACT Loops, see the following Chapter 7.

6.1.2. Interfaces

For specifying the communication interfaces between REACT components and with REACT, IDL-based interfaces for the execution of Remote Procedure Calls (RPCs) are used. This enables to support multiple programming languages by generating high-level language-specific bindings. As shown in Figure 6.1, there are three external interfaces, which are usable by system developers. In order to connect REACT to the underlying communication system, REACT Core provides sensor and effector interfaces (`ISensor` and `IEffector`). For updating and interacting with the knowledge service, an `IKnowledgeSerice` interface exists. Beginning with the former two, they are shown in Listing 6.1. The sensor receives live context information from different parts of the communication system and forwards it to the feedback loop. This data is provided in a serializable format, such as JSON or XML. The system developer must provide the data periodically by calling the `receiveSensorData` method. At the other end, the effector implementing the `IEffector` interface as part of the managed resource receives the result of the feedback loop. First, such an effector is able to receive parameter changes via the *sendParameterChanges* method, and second, the *sendComponentChanges* method allows updating the composition of the managed resource. In both cases, it is the task of the system developer to execute the changes provided by the two methods.

```
1 interface IEffector {
2     void sendParameterChanges(ParameterChange p);
3     void sendComponentChanges(ComponentChange c);
4 }
5 interface ISensor {
6     void receiveSensorData(SensorData s);
7 }
```

Listing 6.1: `IEffector` and `ISensor` interfaces.

Looking at the interface of the knowledge service, the method `sendKnowledge`, which is exposed by the `IKnowledgeService` interface shown in Listing 6.2, can be used by system developers to set or update the specifications stored in a knowledge service instance at runtime. Updating knowledge at runtime may be necessary due to two reasons. First, complexity and uncertainty may lead to situations that

were not foreseeable at design time [187, 188]. Second, environmental changes may necessitate model changes. Thus, the `IKnowledgeService` interface allows, for instance, REACT to be connected to a self-improvement [16] module that continuously learns and improves the models used by a feedback loop instance. For fetching data from the knowledge service, the interface supports different knowledge types, as indicated with the comments in Listing 6.2. These types accommodate the three mentioned REACT Loop instances providing SAT, MILP, and CSP-based knowledge.

```
1 interface IKnowledgeService {
2     // Update Knowledge
3     void sendKnowledge(Knowledge k);
4
5     // Get Knowledge
6     // Specific getters for REACT Loops looking like this:
7     // Knowledge getKnowledge()
8     [...]
9 }
```

Listing 6.2: `IKnowledgeService` interface.

Summing up, the presented architecture provides a reusable structure for different REACT Loops. This allows the MAPE components to be instantiated with different model-based logics, while the communication between the components as well as between the adaptation logic and the managed resource is handled by REACT Core. In the following, implementation details of REACT Core omitting the optional context management module, for now, are presented.

6.2. Implementation

This section outlines the implementation details of REACT Core. First, this section presents how the provided runtime environment itself is implemented. This runtime environment is used for executing REACT Loops. The second section presents details about the communication facilities of REACT Core. This includes the communication inside of REACT as well as the communication between REACT and a connected managed resource.

6.2.1. Runtime Environment

One design decision to achieve reusability for the MAPE components is to provide a wrapper named `ALElement`. It can be reused for the different parts of a MAPE-based REACT Loop. Accordingly, this `ALElement` component wraps the actual logics, comparable to SAS frameworks such as FESAS [21] or HAFLoop [125]. This wrapper is responsible for the deployment of the logic on a machine and the communication between itself and a successor component, as well as the knowledge. By that, we perform a separation of concerns, allowing the ALElement component to focus on deployment and communication, while the logic can focus on handling the decision making.

For the internal communication between the MAPE components, an `IALElement` interface, shown in Listing 6.3, is used. The `ALElement` component implements this interface. The main purpose is to provide the `callLogic` method, which calls the logic the `ALElement` wraps. As a parameter a so-called `KnowledgeRecord` has to be provided containing meta-information along with the actual data. Additionally, besides the possibility to change a successor using the configuration, the interface supports changing the successor of an `ALElement` at runtime via an RPC.

```
1 interface IALElement{
2     void callLogic(KnowledgeRecord knowledgeRecord);
3     void setSuccessor(String successorString);
4 }
```

Listing 6.3: `IALElement` interface.

REACT and REACT Core are implemented in Java. As the design of REACT includes that the deployment is changeable at runtime, the implementation must support starting and stopping instances of the feedback loop arbitrarily. Thus, REACT uses OSGi [189, 190] in combination with iPOJO (inject Plain Old Java Objects) [191].

The OSGi standard describes a component and service platform for Java. OSGi adds the notion of components to the Java Virtual Machine. By that, it follows a component model and lifecycle aiming at modularity and extensibility. A component, which is called bundle in the OSGi context, combines multiple Java classes together like a Java Jar. Additionally, a manifest with meta-information

has to be provided for making a Jar a bundle. This meta-information is provided as part of a `MANIFEST.MF` file, which specifies information such as a bundle name, a description, a version, and which class is initially loaded when the bundle is starting. It can also be specified which packages or classes are provided and exposed and which other packages or classes are needed to start a bundle. Hence, OSGi adds the idea of visibility by explicitly exposing classes and importing other classes forming a system to manage dependencies. In order to provide this functionality, a bundle has to implement a publicly provided interface, which can be imported by other bundles. The OSGi runtime then has to make sure that a bundle importing an interface can only be started if another bundle providing the interface is available. For this, OSGi uses a bundle life cycle shown in Figure 6.2.

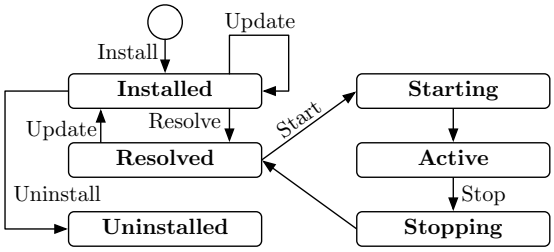


Figure 6.2.: OSGi Lifecycle [189, Section 3.2.5].

First, a bundle is *Installed* in the OSGi runtime environment, which also registers the bundle in an OSGi-managed service registry. This means that the bundle is loaded, and the OSGi runtime tries to resolve the needed dependencies. In case all needed classes are available in the runtime environment, the status changes to *Resolved*. Then, a bundle can be started, changing to the state *Starting*. In case there is an exception, it is possible that the bundle goes back to the state *Resolved* [189, Section 3.2.5]. Otherwise, it implicitly changes to the *Active* state. In case the bundle is stopped, a transition via the *Stopping* state back to *Resolved* takes place. If a bundle should be removed from the runtime environment, it has to go back to the *Installed* state, and then to *Uninstalled*[2]. In case a bundle instance gets updated at runtime, it has to be in the *Installed* state to make sure that potential changes in the required dependencies can be met.

This lifecycle enables to arbitrarily add, remove, start, and stop bundles at runtime without stopping the underlying Java Virtual Machine. More specifically,

[2]According to [189, Section 3.2.5], this has changed in version 4.2 of the OSGi specification. Before that, it was possible to directly transition from *Resolved* to *Uninstalled*.

the `Sensor`, `ALElement`, and `KnowledgeService` classes are bundles in REACT. Hence, this fulfills the initially mentioned requirement for changing the instances of a feedback loop at runtime. The OSGi standard also defines many additional services for, e.g., authentication, configuration management, and more. However, they are out of the scope of this thesis, as they are not explicitly needed in our case.

The OSGi Alliance only provides the OSGi standard, which has to be implemented as part of a specific implementation named OSGi framework. There are different OSGi frameworks available such as Apache Felix[3] or Eclipse Equinox[4]. In our case, we use Eclipse Equinox for executing REACT. However, as every runtime environment implements the same OSGi standard, portability between the different OSGi frameworks is provided. Hence, REACT can possibly also be executed in other OSGi frameworks.

For implementing REACT, iPOJO [191] is used. iPOJO provides a service-oriented platform on the foundation of OSGi. As the name suggests, the general idea is to inject POJOs with service handlers. These handlers manage the behavior of the object according to life cycle transitions. Additionally, iPOJO adds a distinction between a bundle and instances of that bundle. Hence, as with classes and objects, iPOJO enables to instantiate multiple instances of a bundle. Each instance can be set up with different properties. This is exactly the functionality that REACT needs to instantiate, e.g., multiple monitor instances in the same OSGi runtime.

The configuration of the instances is achieved using Apache Felix File Install[5]. This OSGi bundle allows observing the contents of a folder for automatically instantiating bundles based on its content. Each component is represented by a single key-value-based file setting the available properties of the according iPOJO component. Hence, the content of the observed folder directly represents the different bundle instances and their configuration. This enables to deploy instances of REACT's components with different configurations easily. As the OSGi runtime observes the folder also at runtime, this allows on-the-fly changes of the deployment. Considering REACT, the configurations are used to, e.g., specify a component's successor, which knowledge service should be used, or from

[3] `https://felix.apache.org/`, accessed 2020-12-08
[4] `https://www.eclipse.org/equinox/`, accessed 2020-12-08
[5] `http://felix.apache.org/documentation/subprojects/apache-felix-file-install.html`, accessed 2020-12-08

which file a knowledge service should load its specification. The connection to the successor can be specified manually, or it is specified which technique a component should use for automatic setup. The next section introduces the implementation of the communication as well as the different possibilities to set it up.

6.2.2. Communication

As presented in the previous sections, REACT Core provides different interfaces for the external and internal communication. The main goals concerning communication are supporting distributed deployments, allowing the use of different programming languages targeting heterogeneity, and having a low setup effort.

Considering the communication goals, this work identified four candidate technologies, which tackle the problem of the heterogeneity. Accordingly, this section briefly discusses the Common Object Request Broker Architecture (CORBA) [94], gRPC[6], Remote-OSGI (R-OSGi) [192], as well as ZeroC Ice [193]. Additional approaches such as Apache Thrift [194] or Jini [195] have been disregarded due to their early development stage, missing documentation, or the use of a Service-Object-Oriented-Architecture, which does not enable the use of multiple programming languages. CORBA represents a settled standard, while gRPC is rather recent [196]. R-OSGi aims at providing an RPC solution with focus on the OSGi standard. ZeroC Ice can be considered as the successor of CORBA, as former CORBA developers started its development [197].

CORBA is a standard defined by the Object Management Group (OMG) without providing a reference implementation. The CORBA standard defines a specification providing distributed objects between different programming languages. CORBA relies on an IDL for generating programming language bindings. However, a specific implementation of the CORBA standard does not have to support all possible language bindings. A developer, who wants to use CORBA, must select a commercial or open source implementation based on the development requirements. According to the official CORBA website[7], there is no open source implementation available that supports more than three language bindings. Most approaches support between one or two languages. Other disadvantages of CORBA are the

[6] `https://grpc.io/`, accessed 2020-12-08
[7] `https://www.corba.org/corbadownloads.htm`, accessed 2020-12-08

steep learning curve, complex application programming interfaces (APIs), and inconsistencies when using different languages [197].

Next, the relatively new gRPC framework also tries to provide a high-performance and universal open source RPC framework. gRPC was originally developed by Google in 2015 and uses Protocol Buffers (protobuf) for serialization. Protobuf provides an IDL as well for describing data structures and interfaces, which is used by gRPC for generating language-specific interfaces. gRPC does not allow distributed objects for focussing on microservice architectures. Although developed by Google, there is no professional support, and the documentation is rather basic.

R-OSGi uses proxies between multiple OSGi frameworks for transparently providing distributed objects [192]. The main problems here are that it is not (professionally) supported, and it has not been further developed since 2009[8]. Additionally, it is not suitable for non-OSGi software. Since REACT tries to target as many heterogeneous systems as possible, only supporting OSGi-based software decreases the broad applicability of the approach.

ZeroC Ice aims at providing an improved version of CORBA. Compared to CORBA, it is not only a standard but also an open source implementation. Ice also follows the approach of distributed objects. It provides out-of-the-box support for a variety of programming languages, has a low learning curve, and fixes problems of CORBA, such as the API complexity and inconsistencies [197]. These facts underline that there is a single company behind Ice and not the OMG as a standardization consortium. Hence, ZeroC provides a stable solution with a lot of documentation and support, including extensions for IDEs for generating language bindings from the IDL files. Accordingly, for achieving the aforementioned goals, REACT's interfaces are specified using ZeroC Ice's IDL.

Even though Ice provides the communication facilities themselves, the different components still have to find and connect to each other. REACT provides three possibilities, which have been briefly introduced in Section 6.1. First, there is the possibility to manually set up the communication between the feedback loop components using key-value properties. This includes fixed IP addresses and ports. As this option indicates a high setup effort, two other possible

[8] `https://sourceforge.net/projects/r-osgi/files/`, accessed 2020-12-08

ways are included in our approach. Second, for deployments over the Internet, an instance of the Consul[9] registry for automatic setup can be used. In this case, all feedback loop components register at a central Consul registry for setting up the communication. Finally, there is the possibility that feedback loop components find themselves automatically on the local network. This capability is implemented using *Multicast DNS* [198] and *DNS-Based Service Discovery* [199], also known as their implementation names such as Bonjour[10] or Avahi[11]. REACT specifically uses JmDNS[12] for publishing feedback loop components as services and searching for them in the local network. Using this technology only requires the system developer to specify names for the feedback loop chain elements, and the corresponding components find their successor or applicable knowledge service automatically in the local network.

The developer can decide which one of the three approaches should be used using the key-value-based configuration file. If there is no manual specification of the successor using IP address and port, as well as no specification of a registry server, a component automatically falls back to the default Multicast DNS-based approach.

6.3. Context Management Module

After the introduction of the fundamental parts of REACT Core, this section presents the optional context management module. Until now, the presented architecture of REACT omitted the context management module. This section presents REACT's context management capabilities for storing, exploiting, and distributing the internal and external context of a connected managed resource.

As presented in the previous sections, the knowledge service itself is only responsible for providing the model knowledge specifying the adaptation behavior. The context module aims at providing three additional features: context *storage*, *reminiscence*, and *distribution*. The storage saves context situations with the corresponding adaptation decisions in a database enabling to analyze the adaptation

[9] `https://www.consul.io/`, accessed 2020-12-08
[10] `https://developer.apple.com/bonjour/`, accessed 2020-12-08
[11] `https://www.avahi.org/`, accessed 2020-12-08
[12] `https://github.com/jmdns/jmdns`, accessed 2020-12-08

behavior over time. This storage is directly used by the reminiscence function, which checks if the current context has been encountered in the past. If this is the case, the feedback loop is able to omit the complex planning and can directly implement the previously planned adaptation in the managed resource. Finally, the distribution feature publishes (new) context situations and planned adaptations via a publish/subscribe system. This enables the integration of REACT into other systems and the use of the sensed contexts and planned adaptations in external systems, e.g., for a self-improvement software. Since REACT and REACT Core aim at being as lightweight as possible and a database is rather heavyweight and constitutes a central instance, the context module is optional and has to be enabled manually in the knowledge service configuration.

6.3.1. Architecture

Figure 6.3 shows the extended architecture of REACT, including the *Context Manager*. For addressing all requirements to handle context, REACT in combination with the context manager, supports the phases of the context life cycle consisting of context *acquisition*, *modeling*, *reasoning*, and *distribution* [44, 200]. REACT supports the acquisition phase by receiving the sensor data from the managed resource via the `ISensor` interface.

Storage is the first feature for managing and modeling context information. In this case, any internal or external context information of the managed resource sent to the sensor is stored here. In order to reason about the context, the according adaptation is stored for each context information. This allows not only to use this information for monitoring the behavior over time, but it also allows to query the storage for faster adaptation. This is the function of the *Reminiscence* module. This module allows the analyzer to check the existing context-adaptation pairs for the current context. In case the current context has already been stored, and an according adaptation has been planned and executed, the MAPE loop can skip the planning step and execute the previously planned adaptation directly. The *Reminiscence* enables to also specify a degree of similarity for numeric values. This allows for specifying, e.g., that the current context values can be a specific percentage off the already stored values. Hence, this capability is especially useful in the case of real numbers in the sensor information where

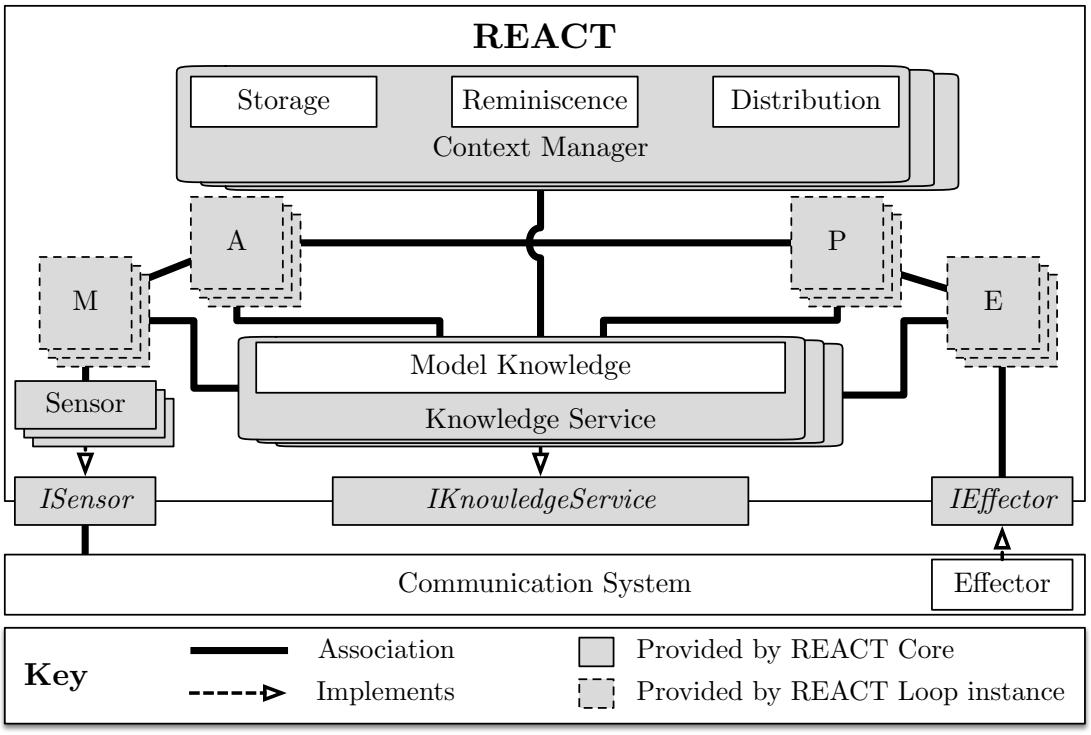


Figure 6.3.: Architecture REACT including the optional context manager module. It consists of a storage, reminiscence, and distribution functionality and is connected to the knowledge service.

even the smallest differences would directly indicate a different context state. The last module handles the context *Distribution*. The goal of this module is to distribute context information as well as corresponding adaptations from the storage to interested external software components. This functionality enables these software components to reason and learn about the context and adaptations over time. Technically a publish/subscribe-based system is used. This enables to notify interested (external) software components about new context information immediately. Single explicit queries resulting in a lookup in the storage can also be triggered via the publish/subscribe system. The looked-up results are then published via the publish/subscribe system as well.

6.3.2. Implementation

This section outlines the implementation details of the context manager by walking through its three components.

Storage: Beginning with the storage component, an SQL-based approach is applied. Considering performance, there is not a clear winner when comparing SQL and NoSQL-based databases [201–204]. Since the model-based specifications as part of the knowledge service already determine a schema, a schema-based SQL database is used. Hence, the storage module employs the MySQL[13] database management system, which is widely used. The database is divided into two types of tables. First, there is a context table representing all context attributes used by the developer as part of the model-based specification. Second, there is a table representing the according adaptations. Between both table types, context, and adaptation tables, a foreign key relation is established to map context situations to (planned) adaptations. The table structures are generated at the startup of the system based on the model-based specification. Furthermore, the storage component enables the execution of general operations on the database, such as inserting data and querying the already existing entries.

Reminiscence: The reminiscence module uses predefined value ranges for specifying how similar context parameters have to be, for mapping it to a previous context. Hence, the context module allows for exact definitions of how similar the respective values should be for each context parameter. For specifying the value ranges, a map structure is used, which can be set up using an additional method specified in the IDL specifying the `IKnowledgeService`. The IDL-based specification enables setting up the used similarity values as part of the reminiscence module and updating these values at runtime. As an example, a map entry [“attributeOne”, “+- 2”] specifies that the attribute with the name attributeOne is considered similar in a range of +/- 2. This only works for numeric attributes. In the case of Boolean or string attributes, the reminiscence only supports exact values. The reminiscence module uses the specified ranges and values at runtime to check the similarity. If a context is considered similar, the corresponding adaptations can directly be forwarded and executed, resulting in skipping planning and the mapping from adaptation actions to the managed resource as part of the executing step. In order to support this, the REACT Loops explicitly check the similarity using the context manager as part of the analyzer and forward a flag, including the previously planned adaptation as part of its internal communication. This functionality enables the adaptation logic to skip the rest of the loop.

[13] `https://www.mysql.com`, accessed 2020-12-08

Distribution: For the distribution of context changes and planned adaptations to external software components, the Message Queuing Telemetry Transport (MQTT) protocol is used, which provides a lightweight publish/subscribe solution. The use of MQTT results in a low coupling between the distribution module and interested external software components. This low coupling minimizes the changes needed by software developers to use the distributed data and provides a high degree of freedom in the implementation. Specifically, Eclipse Mosquitto[14] is the used broker, while Eclipse Paho[15] is used as the Java library for communicating with the broker. If the distribution is enabled via methods defined in the IDL-based interface of the context manager, it connects to an MQTT broker and publishes events. It is either possible to start a local MQTT broker or to specify an external one. Publishing the events allows external systems to receive contexts and adaptations, e.g., for learning new adaptations behavior using machine learning techniques [44]. REACT Core providing interfaces to access and update the knowledge of the knowledge component enables possible self-improvement.

6.3.3. Feasibility Study

In order to test the functionality of the context module, a feasibility study has been conducted. As a use case, an adaptive traffic light scenario is applied. In the setup, the adaptive traffic light senses the number of waiting cars on each lane at a crossing, the traffic density, the average speed of the cars, as well as the time a direction has a green traffic light. REACT adapts the green time of the different lanes at the crossing in response to the sensor data. Hence, the objective is to adapt the traffic light to irregular traffic flows, which occur over the time of a day. The evaluation uses the VSimRTI simulator [205] in combination with the Simulation of Urban Mobility (SUMO) [206] for specifically simulating the traffic. SUMO also allows implementing smart traffic lights at an intersection. The main goal of the context manager is to store past context situations and the corresponding adaptations to reduce the planning and execution time. Thus, as part of this feasibility study, the gains considering the reduced execution times are quantified. The other generic parts of REACT Core are evaluated in combination with the REACT Loop implementations as part of Chapter 7.

[14] `https://mosquitto.org/`, accessed 2020-12-08
[15] `https://www.eclipse.org/paho/`, accessed 2020-12-08

Table 6.1 shows the measured runtimes of the feasibility study. First, the analyzer has to map the sensor data to the model-based specification. After that, the context can be checked when applying the context management module. This can either result in the context being new or already known. If the context is new, it is added to the context storage, and the loop continues. Otherwise, the rest is skipped. Looking at the context checking results, they show only a small difference between the two measurements. The biggest gain in lowering the runtime can be achieved when the planner can be skipped. This is the case when a previously planned and stored adaptation can directly be forwarded. In combination with skipping most part of the executor's logic, this results in a decrease of around 63 ms in the feasibility study in the adaptive traffic light scenario compared to not using the context management module. Thus, the context module reduces the runtime of the used REACT Loop by applying the reminiscence feature and increases REACT's extensibility due to the distribution functionality. As the context module is as generic as possible, we state that it is applicable for all available REACT Loops.

	Analyzer		**Planner**	**Executor**	**Total**
	Map	**Check Context**			
Run	5.41 ms	3.26 ms	58.69 ms	2.43 ms	69.79 ms
Skip	5.41 ms	3.64 ms	0.04 ms	0.06ms	9.15 ms
Without CM	5.41 ms	0 ms	58.69 ms	2.43 ms	66.53 ms

Table 6.1.: Results of the feasibility study for evaluating the context manager. The monitoring component is omitted as it does not utilize the context manager. CM: context module.

Discussion and Related Work

The feasibility study shows an advantage in the execution time in the case similar context situations are encountered at runtime. When taking the results of Table 6.1 into account, it is possible to calculate the percentage the application of the context manager improves the average execution time. It is beneficial to use the context manager if it is possible to skip the planning and execution in 6 % of the cases. Specifically in the evaluation, 81 % of the time, the context was already known resulting in skipping. However, it is noteworthy that the execution frequency of the loop, beginning with the sensing as well as the context complexity and specified similarity settings, have a large impact on these results.

Considering related work in the field of context management, there are many works in the area of pervasive and context-aware computing, such as Aura [207], CARISMA [208], Gaia [209], or PROACTIVE [210]. The interested reader is referred to [211], [212] for an overview of context-aware systems. The mentioned example approaches follow different methods for storing and using context, reaching from model-based approaches in the case of CARISMA [208] to ontology-based approaches in the case of Gaia [209]. Looking at other overviews of related work, [44] and [211] show a shift from rules towards ontology-based approaches for reasoning. However, as REACT's goal is to provide an approach that is applicable in all kinds of different use cases, a fixed ontology explicitly for modeling the context is not suitable. The current implementation of the context storage on the SQL database's foundation does not impose limitations on the data format. Obviously, in this case, the context module has to take care of querying the database in a suitable way. Overall, the model-based specifications already provide a structure for generating tables for solving the structural data storage without an additional schema. Finally, looking at the distribution capabilities, there exist both explicit query- and subscription-based approaches [44]. In order to provide a highly flexible solution to developers, the current implementation of the context module provides both approaches as well. This enables to request single queries but also to get notified in case context information changes. In future work, specific reasoning techniques from the high number of related works could be integrated directly into the context module. Additionally, proactivity methods, like the ones presented in [210], could improve the adaptation decisions and, as a consequence, the managed resource's performance further.

6.4. Development Process

Finally, as mentioned in the introduction of this chapter, we propose a development process for using REACT as presented in [11]. The process helps system developers in applying REACT as part of new or existing systems. It is as generic as possible and can, therefore, be applied with any one of the REACT Loops. An overview of the development process is depicted in Figure 6.4, consisting of the three development steps: 1) Modeling, 2) Connecting, and 3) Configuring.

1. Modeling		2. Connecting		3. Configuring	
1.1	Problem Space	**2.1**	Effector Implementation	**3.1**	Key-Value-Based Configuration Files
1.2	Solution Space	**2.2**	Sensor Implementation		

Figure 6.4.: Proposed development process for applying REACT using a provided REACT Loop [11].

In the first step, the system developer creates models specifying the *Problem Space* (*1.1*) as well as the *Solution Space* (*1.2*). This step specifies the adaptation behavior of the deployed feedback loop instance. Separating the models is largely inspired by works in the SPL community [213] and models@run.time [105] research, which distinguishes between problem and solution space. According to [213], the problem space consists of the "stakeholder needs and desired features", while the solution space represents the "architecture and components of the technical solution". The separation into problem and solution space allows REACT to reuse the same problem space specification for different communication systems. For instance, a shared repository with reusable specifications could offer other system developers the possibility to readily use an already available specification for their system. In this case, they could only specify the solution space representing the managed resource and reuse an existing problem space specification. Depending on the used REACT Loop, different modeling possibilities are used in this step. Additionally, it is possible to apply model checking techniques to ensure that the specifications provide a certain degree of correctness. Depending on the modeling technique, it is also possible to test the behavior of a specification using example values at design time.

After the modeling part, system developers connect REACT to the managed resource as part of the connecting step. First, they implement the effector interface to the managed resource (*2.1*), which was presented in Section 6.1. As it is common in the field of self-adaptive systems to have parameter and architectural adaptation [6], the interface encompasses methods for each adaptation type. Currently, REACT always sends complete instances of the solution space to the managed resource instead of only the difference between the current and new configurations.

After this step, the system developer implements the sending of sensor information from the managed resource to REACT (*2.2*). As shown in Section 6.1, the `ISensor` interface specifies the method implemented in REACT's sensor component instances for receiving sensor information. Therefore, the managed resource calls the sensor function periodically to trigger REACT. After these implementation steps, the interfaces can be used with REACT and any kind of specification, i.e., arbitrary problem and solution space specifications. Consequently, they can be reused if the specification changes.

Finally, the system developer or administrator provides REACT with a minimal set of configuration information (*3.1*). This configuration step enables a distributed deployment of REACT's components. One configuration file for each MAPE component allows configuring the component type, a unique identifier, the successor component, and the corresponding knowledge component. Additionally, the context manager can be enabled and configured. REACT then creates and deploys a respective component for each key/value-based configuration file.

7. Feedback Loop Instantiations

The previous chapter introduced REACT Core, which provides a reusable framework for REACT Loops. In this chapter, different specific REACT Loop instantiations are presented. Section 7.1 begins with the SAT REACT Loop using context feature models, which get transformed to Boolean expressions. Section 7.2 presents the MILP REACT Loop, which increases the expressiveness of the used context feature modeling approach and introduces non-functional goals. The final REACT Loop in Section 7.3 uses context feature models, which get transformed into CSPs, and solves these problems using constraint programming (CP). Finally, Section 7.4 presents a feasibility study comparing and combining different REACT Loops using a common use case.

Each section that presents a REACT Loop instance introduces the problem domain, the modeling approach, as well as an overview of the respective architectures and implementations. This includes the evaluation of each instance in different use cases. The use cases consist of examples from distributed and edge computing systems [214], topology control in wireless sensor networks (WSNs) [215], cloud server management [216], and (wireless) software-defined networks [217]. Each section finishes with a presentation of the evaluation results as well as a discussion.

7.1. SAT-Based Feedback Loop

The first REACT Loop is a SAT-based approach presented in [104][1] and [218]. In the following, this section introduces satisfiability problems and how the approach transforms context feature models into these problems. Then, this section outlines the architecture and corresponding implementation as well as the evaluation and discussion.

[1] [104] is joint work with C. Krupitzer, M. Weckesser, and C. Becker.

7.1.1. Satisfiability Problems

Petke defines (Boolean) satisfiability problems as follows [219, p. 15]:

> *The problem of deciding whether there is a variable assignment that satisfies a propositional formula is called the Boolean satisfiability problem (SAT).*

The logical specification of a SAT problem consists of multiple clauses in conjunctive normal form of a formula [220], [221, p. 253 ff.]. Each clause consists of one or multiple literals, while a literal is a "*propositional variable or its negation*" [220, p. 124]. Thus, each clause represents some constraints. All clauses, which are disjunctions for themselves, are conjunctively connected with each other. If there is a possible assignment of Boolean values to all literals satisfying all sentences, there exists a so-called model for this setting. According to Russell and Norvig, satisfiability of a sentence is defined as follows [221, p. 250]:

> *A sentence is satisfiable if true in, or satisfied by, some model.*

This Boolean satisfiability problem can be solved by SAT solvers. A SAT solver accepts clauses as input and creates an assignment for the available variables for this input if there is any [221, p. 271 f.]. A widely used input and output format for SAT solvers is the DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) conjunctive normal form (CNF) format [222]. An example in conjunctive normal form is given in Formula 7.1. In this case, A or B as well as C have to be assigned to *true* for the whole formula evaluating to *true*.

$$(A \vee B) \wedge C \tag{7.1}$$

7.1.2. SAT-Based Context-Aware Feature Modeling Approach

This work uses the context feature modeling approach introduced in Section 2.4 for the problem space. The approach augments generic feature model diagrams with a context branch in addition to the system features. Moreover, the approach uses feature attributes with support for integer and real values, feature instance cardinalities, as well as group type cardinalities for specifying constraints in the model. Cross-tree constraints are used between context features, feature attributes,

and system features for specifying the reconfiguration behavior. They can either require or exclude a feature. At runtime, the selection of context feature attributes represents the current state of the running software.

For finding valid configurations at runtime, the context feature model is mapped to a Boolean representation that is interpretable by a SAT solver. Each feature represents a literal, which can either have the value true or false. Non-numeric cross-tree constraints can also be translated into a Boolean representation. Since a SAT solver only works with Boolean problem definitions, the feature attribute value ranges are modeled as enumerations, with the specified ranges either being true or false. Hence, each attribute with its value range is translated into multiple so-called feature attribute items (FAI). A FAI represents an enumeration item characterizing a range of values of the corresponding attribute. The translation of the complete context feature model results in a respective DIMACS CNF [222] representation of the model. Due to the restriction to Boolean variables, features can only be present once or not at all. An explicit model of the solution space is not part of the SAT-based REACT Loop.

7.1.3. SAT-Based Architecture

The architecture of the SAT-based REACT Loop as part of REACT is depicted in Figure 7.1. As an example, an entire run through the feedback loop is described using the data center use case presented in [9]. It describes a self-managing data center that starts additional servers given a high workload. Accordingly, it reduces or maintains the number of servers in low workload situations. Also, it is possible to redistribute virtual machines over all physical servers for better resource utilization. Figure 7.2 shows the big picture of the data flow through the MAPE components, including the context feature model of the example, which is part of the knowledge component. For illustration, the monitor only observes the workload of the servers in this simplified example. The system features consist of a startup policy and a keep policy for server management. Additionally, the VM management possibilities include a redistribution and a stay policy. In this example, one feedback loop manages three data center areas.

As shown in Figure 7.1, the idea is to augment a MAPE-K cycle-based adaptation logic with a CFM inside the knowledge component. The knowledge contains

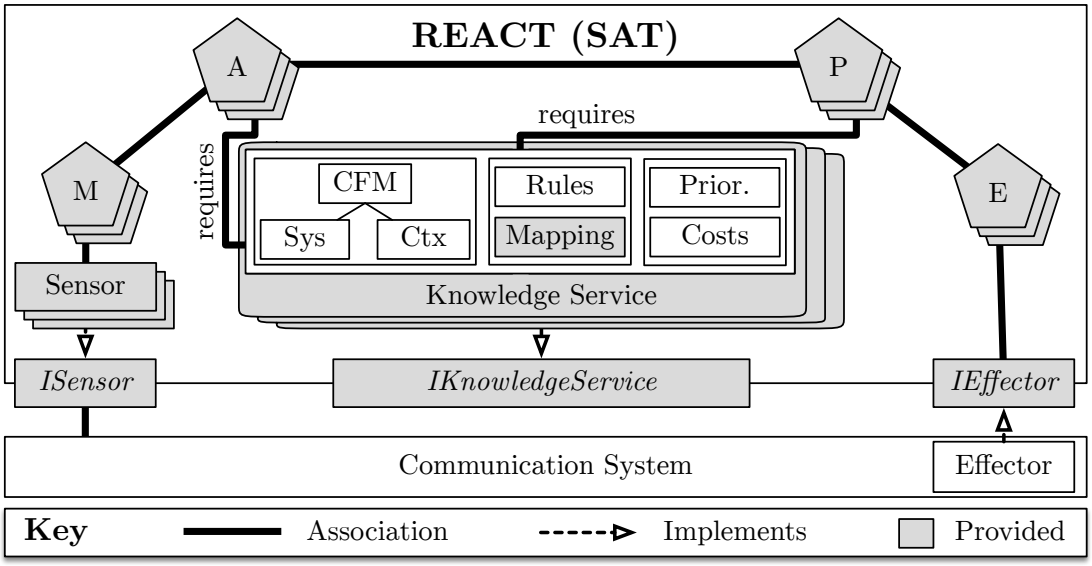


Figure 7.1.: Architecture of REACT using the SAT-based REACT Loop [104,218].

all elements that are needed for reasoning in the adaptation logic. The context feature model is only one part of the knowledge besides the (feature attribute) rules, the priorities, the costs, and the automatically generated mapping. The complete meta-model of the SAT-based knowledge can be seen in Figure A.1 in the appendix. Features can be either system or context features. As introduced in Section 2.4.3, each feature can have two types of cardinalities: a feature instance cardinality and a group type cardinality. Also, the SAT-based REACT Loop uses the extended feature model approach presented in the same section, which enables features to also have attributes.

Rules for relating sensor data to context feature attributes are part of the knowledge as well. Feature attribute rules represent the rules for matching context sensor values to actual feature attributes and FAIs. A rule specifies the name for the sensor input for matching it to an attribute. Hence, the rules provide a matching method for determining the attribute item that is represented by some context value. For the example in Figure 7.2, the attribute workload would have the rules: (1) (Workload<0.6) mapped to the "<60%" FAI and (2) (Workload>=0.6) mapped to the ">=60%" FAI.

Additionally, it is possible to specify priorities and costs for each feature of the feature model. Priorities and costs reside inside the knowledge component as well, and they are used for conflict resolution as well as for the selection of one

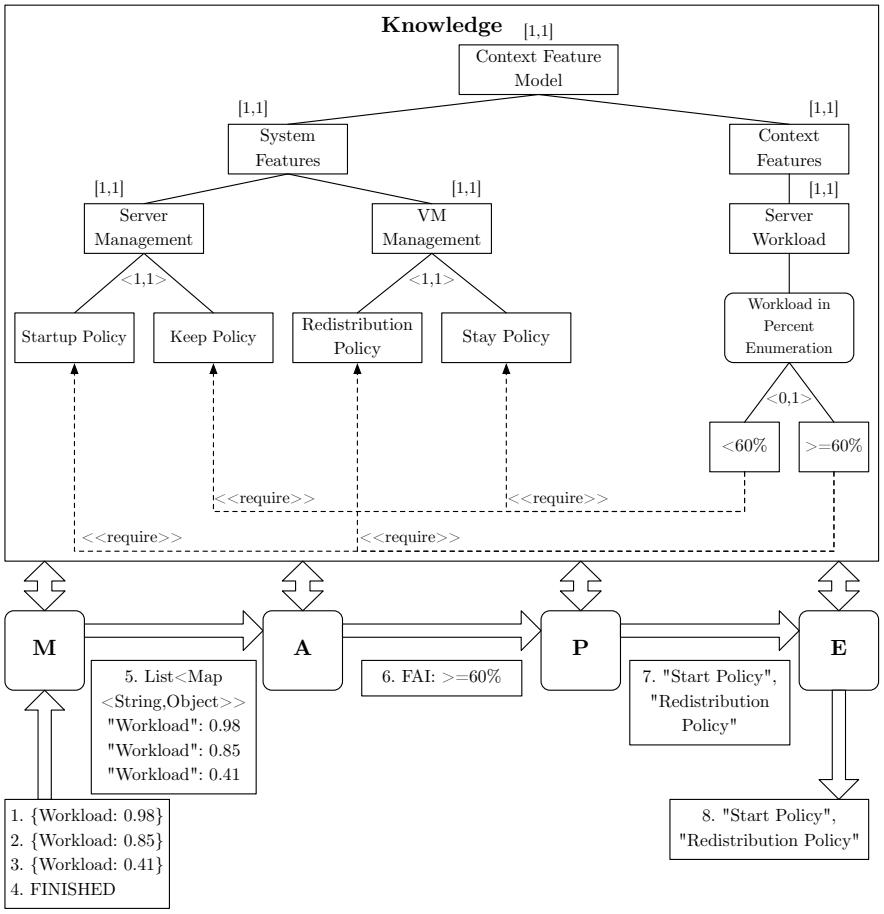


Figure 7.2.: Data flow between the MAPE components in the SAT-based REACT Loop. The knowledge component omits the rules, priorities, and costs in this figure. FAI: Feature Attribute Item, WL: Workload. Feature instance cardinalities are denoted with square brackets. Group type cardinalities are denoted with angle brackets [104, 218].

configuration, given multiple configurations are possible. This enables the support of incomplete or erroneous specifications of system developers to a certain degree. Priorities and costs should be present for all system features, which are part of a feature group. A priority is a number stating the importance of a system feature inside its corresponding feature group. A cost value referring to a system feature states the estimated cost to activate the respective system feature in comparison to other system features of the same feature group.

Finally, the SAT mapping is created automatically by the system directly when it starts. This facilitates the knowledge component to return the corresponding

(SAT) literal given a feature or feature attribute. In the following, the complete data flow through the adaptation logic is illustrated by using Figure 7.2.

The monitoring component receives the sensor data, prepares it, and passes it to the analyzer component. The preparation includes interpretation of serialized input like an XML or JSON string in order to create plain data objects for working with the sensor data. The resulting data objects can be used more easily by the analyzer component than a raw string. The adaptation logic is able to receive and process partial sensor data until the managed resource sends a message indicating the end of data. Thus, the monitor gets sensor information not as one package but in fragments. Mapping this to the data center example, the monitor receives sensor information about each of the three data center areas' workload. Then, as shown in Figure 7.2, this information is followed by a keyword for stating that the monitor should forward the average of the received values to the analyzer. Another possibility is to forward the data periodically.

The analyzer selects the corresponding FAI, while one FAI represents the range feature attribute in the implementation. First, the analyzer creates the average of all entries in order to create one single sensor value representing the average system state. Each average system value is used to map its value to context feature attributes representing the system's context state. With minor customization, developers can specify other aggregation functions than the average. In our case, after the creation of the average values, the rules representing the relationships of context information, their names, and context feature attributes are used. Matching the actual values to context feature attributes requires the rules stating the value range of each context feature attribute. The approach supports partial knowledge, meaning that not for all context feature attributes data must be present. Even without full knowledge, the system is capable of finding a valid configuration. The resulting attributes, which are selected according to the context information, are passed to the planner component.

The workflow of the planner is shown in Figure 7.3. As this planner uses a SAT solver, the result of the mapping process at the beginning is a logical representation of the context information in CNF. The mapper uses the (SAT) mapping of the knowledge component. These mappings state which feature or feature attribute is mapped to which literal for representation inside the SAT solver. Each context feature attribute generated by the analyzer is mapped to its literal

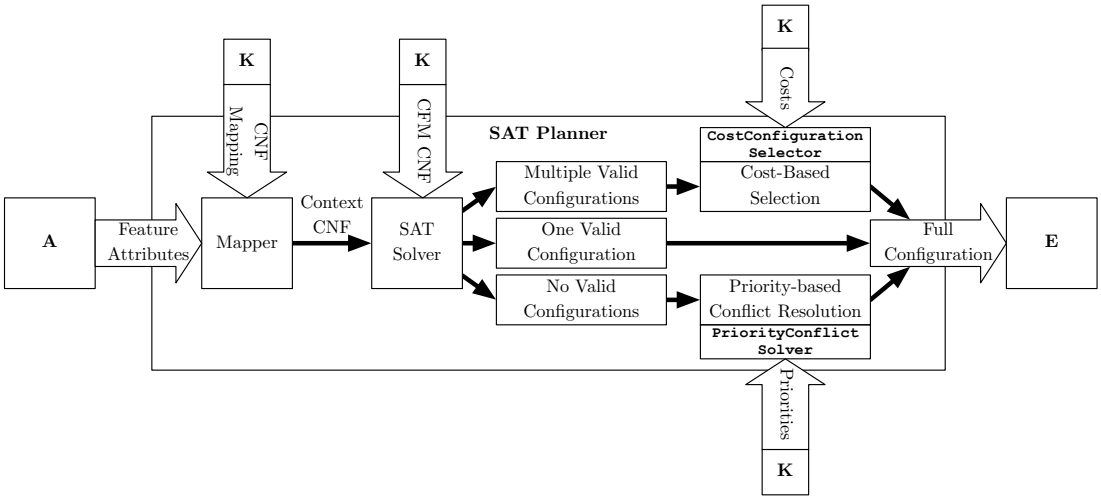


Figure 7.3.: Internal workflow of the SAT-based planner component utilizing a *Mapper*, *SAT Solver*, a `PriorityConflictSolver`, and a `CostConfigurationSelector` [104, 218].

representation resulting in logical clauses in CNF. The result is a DIMACS CNF representation of the entire context information. The CFM—which is available in CNF representation as well—is used in conjunction with the CNF representation of the context information as input for the solver. The solver's task is to determine if valid configurations exist and to output all possible configurations. Now, there are three distinct cases, which are depicted in Figure 7.3. First, if there is only one valid configuration, the planner is finished as there are no other configuration options it can choose from. Second, in the case of multiple possible configurations, the additional cost information residing in the knowledge component is used. The costs contain a numeric cost value for each system feature as part of a feature group inside the CFM. Using this, the planner then selects the configuration with the lowest cost using the `CostConfigurationSelector`. Third, if no valid configuration is found, there is a conflict in the combination of the CNF of the CFM and the context. A conflict occurs if unforeseen context situations arise, which were not anticipated at design time or in case of an erroneous specification. The planner tries to solve the conflict for still allowing to plan an adaptation. Based on the context situation, the conflict can, e.g., result in the need to select multiple features in the same group, which have an XOR relationship. In this case, the priority information is used. The `PriorityConflictSolver`, as part of the planner, determines the priority of conflicting system features in relation to

other system features in their respective feature groups. Then, the feature with the highest priority for each conflicting feature group is selected. This allows the system to also handle unknown or conflicting context states at runtime to a certain extend. Finally, the planner's result is a complete list of system features the managed resource should (de-)activate. Based on the input representing an overall high load, in the example of Figure 7.2, the planner selects the startup policy and the redistribution policy. The selected configuration is sent to the execution component, which forwards the configuration to the corresponding managed resource. This ends one complete cycle through the adaptation logic. In the following, implementation details of all MAPE components with reference to the example are explained.

7.1.4. Implementation of the SAT-Based Feedback Loop

Monitoring: The monitoring component receives multiple raw strings as sensor input. It converts the raw JSON data to Java objects and adds it to an array list. In the example in Figure 7.2, three JSON strings are stored in a `List<Map<String,Object>>` object. Each string represents the current status of the workload of one data center. Passing this array list to the analyzing component triggers the analyzer.

Analyzing: The analyzing component further inspects and handles the values and entries coming from the monitor. In the current implementation, only the average of the monitoring values can be created by the analyzer. Referring to the example in Figure 7.2, the analyzer creates the average workload. This is done for every variable over all items in the list. The resulting value is mapped to the actual feature attributes items representing the context situation of these averaged values. In the example, only the FAI $>=60\%$ is selected. Then the analyzer sends the information about the FAIs to the planning component.

Planning: The planner component uses *Sat4J* as SAT solver [223]. First, using the (SAT) mapping, the planner maps context feature attributes to their literal representation. Next, it checks if there is a model using the given context information. At this point, the solver already has loaded the CNF of the CFM. If there is no model, the conflicting features are determined. The conflicting features are passed to the `PriorityConflictSolver` class, which identifies the system

features responsible for the conflict. For each feature group with conflicts, the feature with the highest priority is selected. Adding the resolved feature selection as well as the remaining conflict-free context information to the solver ends the conflict resolution. If the model is directly free of conflicts, the context information is added to the solver. If multiple models are valid, the `CostConfigurationSelector` class uses the costs assigned to the system features. For every feature group, the feature with the lowest cost is selected. The planner sends a list with all selected system features to the executor.

Execution: The execution forwards the feature selection towards the managed resource. As shown in Figure 7.2, the feature selection is a string with the system feature names separated by a comma. The managed resource maps the system feature names to actual adaptations.

7.1.5. Evaluation of the SAT-Based Feedback Loop

This section outlines the evaluation of the SAT-based REACT Loop. First, we present the use case of the evaluation. Second, we pose the evaluation question and describe the corresponding evaluation scenario. Third, we describe and discuss the results of the evaluation.

Use Case Description

For the following use case, we look at a distributed computing system. Such a system consists of computational resources, clients, and resource managers for scheduling. The goal of the resource managers is to answer resource requests from clients and to schedule this request to the available resources. This enables to offload computations from a client to a resource provider. One problem in distributed computing systems is the changing system context resulting from, e.g., the network churn. If too many resource providers and clients join, the resource managers can get overloaded. Also, due to the heterogeneity of the devices and due to the device context, such as the current network connection, static scheduling can result in low performance. Hence, in this setting, REACT using the SAT-based REACT Loop is applied for adapting the number of available resource managers and their scheduling behavior.

The specific use case of the evaluation is the management of the Tasklet system [214]. The goal of the Tasklet system is to provide a middleware for distributed

computing on heterogeneous devices. Therefore, three entities are available: resource providers, resource consumers, and resource brokers. Providing resources means to offer a Tasklet virtual machine (TVM). Resource consumers send a Tasklet consisting of code and data to providers for remote execution. Additionally, local execution is possible. Each resource provider registers at a broker while brokers themselves form a peer-to-peer overlay network. Consumers send resource requests to a broker, which then searches for a suitable resource provider. A consumer may specify different levels of non-functional requirements called Quality of Computation [214] for a Tasklet, e.g., reliability, speed, or security. This may limit the set of possible providers. In this evaluation, it is possible for consumers to require a high-performance provider for a Tasklet. After the request of the consumer, the broker returns the information for connecting to a provider. The consumer uses this knowledge to directly send a Tasklet to this provider. The provider executes the Tasklet in a TVM. When the computation is finished, the provider directly passes the result to the consumer. Additionally, the consumer informs the broker about the used provider and the successful computation. If a computation fails, the consumer also informs the broker about this. The broker uses this information to calculate reputation values for each provider based on the successful and unsuccessful computations.

Each entity in the network runs the Tasklet middleware that handles the construction of Tasklets, their execution, and distribution. An overview of an example overlay network topology is shown in Figure 7.4. It shows the inner broker overlay network, the connection between providers (P), consumers (C), and brokers, as well as the execution of Tasklets. The figure also shows the possibility that an entity can be provider and consumer at the same time (denoted with P/C). For more details on the Tasklet system, the interested reader is referred to [214].

We simulated the Tasklet system using the simulator presented in [224]. Additionally, we added a broker manager entity, which stores a list of all brokers, monitors them, and adapts their behavior by changing their configurations according to the results of the adaptation logic. Besides changing the configuration of brokers, it can start and stop brokers, as well as redistribute entities connected to them. As the simulator is discrete, we synchronized the simulation with the adaptation logic through the broker manager. Hence, the broker manager transmits all monitored data at the end of a simulation step to the adaptation logic and pauses the simula-

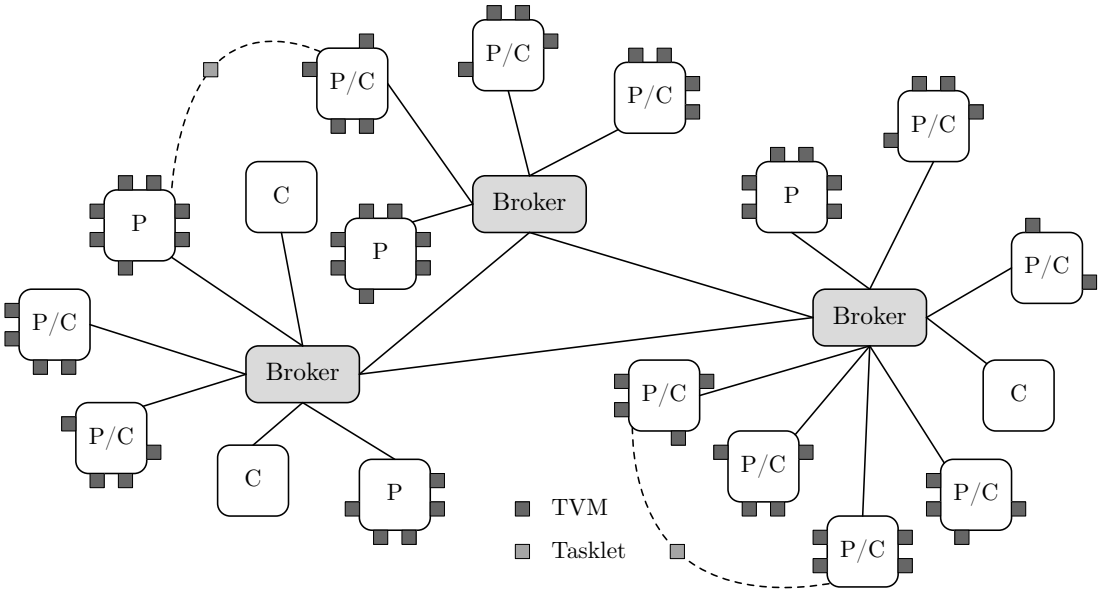


Figure 7.4.: Overview of a Tasklet overlay network topology [214]. P: Provider, C: Consumer, TVM: Tasklet Virtual Machine.

tion. After a run of the REACT Loop, the broker manager adapts the simulated system to the new system configuration. Afterward, it starts the next simulation step. In the case of a real system, adaptations would occur continuously. Thus, the simulated system does not represent this aspect of the real world.

Evaluation Settings

To evaluate the effectiveness of the SAT-based REACT Loop, we define one evaluation question (EQ):

EQ: Does the adaptation logic improve the system performance, and how does the number of nodes in the Tasklet network influence it?

The specific evaluation setting is outlined in the following. In order to adapt the settings of the brokers using the adaptation logic, the next step is to identify system features for the management system as well as context information that is needed to plan the selection of the system features. First, the broker manager is enabled to start new brokers in case the load of the brokers is too high. Next, in order to minimize the number of provider requests to the brokers, cache lists are used. These lists are distributed at regular intervals from the brokers to the consumers, which can use the lists to directly send Tasklets to the providers that are part of the list. If all providers on the list are offline or do not agree on

computing another Tasklet, the consumer has to contact the broker. The lists can be configured at the broker in terms of the distribution interval, the length meaning the number of providers on the lists, and the order of the lists' entries. The cache list interval can be set to slow, standard, or fast. When there is a high fluctuation in the network, the interval for the distribution of the cache lists are changed accordingly between these intervals. The cache list length can be short or long. Thus, when the fluctuation is high, the cache list length is increased to the long length resulting in more providers being on the list. Hence, the consumer has more providers for establishing a possible connection before a request to a broker has to be sent. If the fluctuation is low, the short list length is activated accordingly. The cache list content can either be generic, location-based, or specialized. Generic means the list is ordered according to the reputation of the providers in descending order. The location-based cache list puts providers that are in the same geographic region as the consumer at the beginning of the list, minimizing latency. As part of the simulation, there are three regions for simulating latencies. Each entity is located in one of the three regions. Finally, the specialized cache list puts high-performance providers at the top of the list. This results in a faster execution time when the percentage of high-performance TVM requests is high. For further reference, the complete CFM can be seen in Figure A.2 in the appendix.

For identifying improvements of the adaptation logic, multiple evaluation scenarios with and without the adaptation logic are compared. Each entity is randomly assigned to a distinct geographic region imposing latencies. This evaluation is executed on an Intel Xeon E5345 with 8 cores, Windows Server 2008 Standard, and 6 GB RAM using Oracle Java 8 Update 121. Each run has been executed 30 times. In the first setting, 1000 Providers, 500 consumers, and two brokers are the start configuration. In the following setting, we use 5000 providers, 2500 consumers, and ten brokers. Additionally, the system behavior with different ratios between providers and consumers is evaluated. Thus, we also evaluate the 1:1 and 1:2 ratios resulting in 5000 providers and consumers in setting 3, and 2500 providers and 5000 consumers in setting 4. The evaluation always stops after 25 000 finished Tasklets.

Evaluation Results

In the following, the aggregated results of the evaluation are outlined. We measured two variables in the Tasklet system: the average latency (in milliseconds) imposed by the entity's regional positions and the average load of the brokers in the Tasklet system. The load is quantified using the number of connected providers and a predefined capacity of each broker. This capacity C is defined as $C = \#Providers/\#Brokers$, which is calculated at the startup of a broker. For both latency and load, the adaptation logic triggers a system feature adaptation if they exceed a threshold. We tested different distributions between providers and consumers, and each setting has been executed 30 times. As a result, the simulated system using the adaptation logic has, on average, a 43% lower latency, and the brokers have on average 45% less load than the simulated system without the adaptation logic.

Table 7.1 shows the aggregated measurements per run. The 2500/5000 setting was not able to finish without the SAT-based REACT Loop. Every run ended in an OutOfMemory exception. We also tested it on an r4.2xlarge instance from Amazon EC2 with 61 GB of RAM. Even on this machine, we were not able to finish a single run in this configuration without the SAT-based REACT Loop. One reason is that the overall simulated load in the Tasklet system is high. The ratio between providers and consumers in this setting increases the load even further. This results in a high number of steps for finishing the predefined 25 000 Tasklets. The other results are similar in all four settings indicating that the adaptation logic is always able to fulfill the adaptation goals.

Setting (#P/#C)	Av. latency no AL	Av. latency AL	Av. load no AL	Av. load AL
1000/500	18.38 ms	10.67 ms	100 %	54.70 %
5000/2500	18.35 ms	10.52 ms	100 %	55.36 %
5000/5000	19.12 ms	11.65 ms	100 %	55.27 %
2500/5000	-	11.22 ms	-	55.29 %

Table 7.1.: Aggregated latency and load on the brokers in the first scenario. P: Providers, C: Consumers, Av.: average, AL: Adaptation Logic [104, 218].

Table 7.2 shows the needed simulation steps in the different settings for finishing the 25 000 Tasklets. Due to the increasing load imposed by the different provider/consumer ratios, the number of steps rises with the settings. Again,

in the 2500/5000 setting simulating without using the SAT-based REACT Loop ends in an OutOfMemory exception. The results show that the adaptation logic improves the performance also in this impossible setting. Other than that, it is noteworthy that the adaptation logic decreases the performance in the 1000/500 setting.

Setting (#P/#C)	Steps no AL	Steps AL	Change
1000/500	967.67	1288.03	+33 %
5000/2500	1599.87	1244.70	-22 %
5000/5000	2205.33	1110.07	-50 %
2500/5000	-	1301.57	-

Table 7.2.: Aggregated results of the average steps needed for the first evaluation question. P: Providers, C: Consumers, AL: Adaptation Logic [104,218].

7.1.6. Discussion of the SAT-Based Feedback Loop

This section discusses the evaluation presented in the previous section. When looking at Figure 7.1, the adaptation logic reduces the average latency as well as the average load of the brokers in all settings. In the first three settings, the latency could be reduced on average by 41.3 %. Respectively the load could be reduced by 44.89 % on average. These results show that the modeled behavior was successfully enforced in the Tasklet system.

The most interesting change is the number of simulation steps needed by the simulated system in the four settings. Table 7.2 shows them, including the improvement between the aggregated values of the runs without and with the SAT-based REACT Loop. Again, there are no comparison values for the last setting. In the 1000/500 scenario, the overhead of the adaptation logic, including the broker management system's policies, results in a worse performance. A reason for this might be that brokers that get stopped are directly stopped without further notice. As a single broker has a higher percentage of the total number of providers in the 1000/500 setting compared to the other ones, a stopping broker triggers more reconnects to other brokers and increases the number of failing Tasklet requests. The larger the setting gets, the higher the improvement regarding the number of saved simulation steps is. The third setting, which is the largest setting

regarding the number of the entities, even needs 50 % fewer simulation steps employing the SAT-based REACT Loop.

The SAT-based implementation already shows the potential of REACT. As the expressiveness of SAT problems is limited to Boolean expressions, the following section uses a MILP solver. Accordingly, the following section presents the MILP-based REACT Loop.

7.2. MILP-Based Feedback Loop

This section presents the MILP-based REACT Loop, which is based on [225][2]. Hence, in the following, the MILP problem domain is introduced. Then, the feature modeling approach as well as the overall architecture and implementation of the loop are outlined. Finally, the feedback loop is evaluated and discussed.

7.2.1. Mixed-Integer Linear Programming Problems

In this feedback loop, the configuration problem for planning adaptations is formulated as a mathematical optimization problem [226]. Therefore, optimal solutions can be found using mathematical solvers. The problem consists of linear inequalities, as well as a set of decision variables, which can have integer or real values. Additionally, an objective function stating if a decision variable should be maximized or minimized is added.

7.2.2. MILP-Based Context-Aware Feature Modeling Approach

For the instantiation of a MILP-based feedback loop, the first change to the SAT-based approach is the explicit differentiation between problem and solution space, as introduced in Chapter 6 and presented in, e.g., [177] and the models@run.time approach [105]. This leads to the separation of two models decoupling the specification of the reconfiguration behavior from the definition of the managed resource and its architecture. The separation has the advantage that problem space specifications can be reused with different managed resources. A disadvantage

[2] [225] is joint work with M. Weckesser, R. Speith (né Kluge), M. Matthé, A. Schürr, and C. Becker.

besides creating two models instead of one is that this approach needs an additional mapping between problem and solution space.

A CFM-based specification is used for the problem space. The capabilities of the modeling approach are the same as for the SAT-based feedback loop. It is possible to transform the CFM of this approach either to a SAT or to a MILP problem definition. Hence, either DIMACS CNF or a mathematical problem formulation is generated from the CFM. In this approach, the system features and system feature attributes are decision variables. As we target the MILP problem domain, attributes can have integer and real values without the need for using enumerations. The structure and consistency properties of the feature model can also be transformed into mathematical constraints [225]. If the SAT solver is chosen, this limits the expressiveness again. Accordingly, no integer or real values can be used directly when employing a SAT solver.

For the solution space, there are different approaches available such as ADLs or DSLs, which are used in Rainbow [17] or Genie [120], respectively. We selected UML class diagrams for representing the solution space, as they typically do not require a developer to learn a new meta-model or DSL. Additionally, UML diagrams can also be generated from existing source code, which can possibly reduce the development time when integrating REACT. Hence, a system developer not only has to provide the CFM representing the solution space but also a UML class diagram representing the architecture of the managed resource.

Finally, an explicit mapping relating CFM and UML class diagram is needed. In our approach, this mapping is represented using a text file matching features and attributes to classes and properties in a line-by-line manner.

7.2.3. MILP-Based Architecture

Figure 7.5 shows an overview of the MILP-based instantiation. The knowledge is divided into the specification of functional and non-functional constraints. Looking at the functional constraints first, the knowledge consists of the CFM, a mapping, as well as a class diagram. This is due to the fact that the problem and solution space are separated. The functional constraints imposed by these parts can be used either with a SAT or MILP solver.

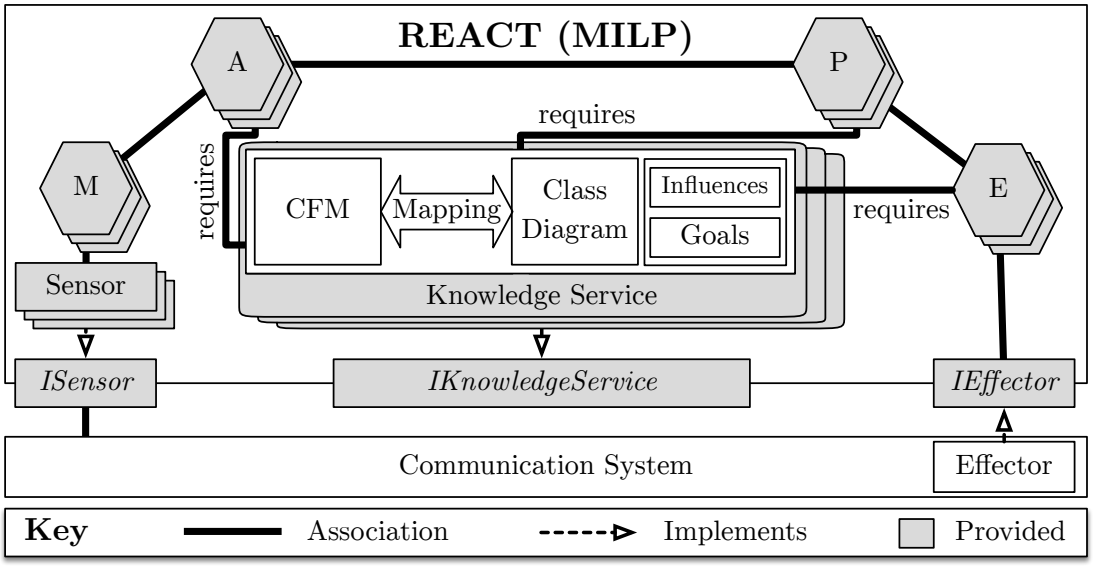


Figure 7.5.: Architecture of REACT using the MILP-based REACT Loop. The knowledge consists of a CFM representing the problem space, a class diagram representing the solution space, a mapping between both, and non-functional (performance) goals and (performance) influences [225].

Supporting non-functional constraints is another advantage over a SAT-only-based approach. Information about the effect of a feature or attribute on non-functional metrics are represented by performance-influence models [227] and are only supported when using a MILP solver. They represent the influence of system configurations on non-functional properties. In the MILP-based REACT Loop, the performance-influence models are represented by regression formulas. Each non-functional property has its own formula. There are two possibilities for providing information about performance influences. First, an expert can assume certain influences and build up the formulas by hand. Second, the performance influences are learned offline at design time. By using the information of performance influences, it is possible to optimize the planned configuration towards specific goals. The performance goals are a list of the available non-functional properties with weights attached to them. This way, it is possible to specify a specific trade-off, e.g., between performance and costs. Hence, this additional information optionally provides the possibility to optimize the adaptations further if a MILP solver is applied.

Looking at a run of the MAPE components in Figure 7.6, the monitor preprocesses the context data. This context can, e.g., be JSON or XML. Based on the preprocessed context, the analyzer, first, creates an instance of the class diagram. In this instance of the class diagram, a specific environment component contains all properties representing the context of the managed resource. Using the mapping, the context subtree of the CFM is instantiated. This results in a partial CFM configuration where only the context part is instantiated. The planner creates the mathematical optimization problem or the DIMACS CNF when using a SAT solver, respectively. This means it transforms the constraints imposed by the CFM itself as well as the context-based constraints to a MILP or SAT problem. Then, it uses a MILP or SAT solver for deciding about the system features and attributes. Hence, the result of the planner is an optimal system configuration. Finally, the executor maps this full configuration to the class diagram resulting in a class diagram instance. The class diagram instance represents the architecture and properties of the managed resource in its adapted state. This model is passed to the effector, which has to implement the changes in the managed resource.

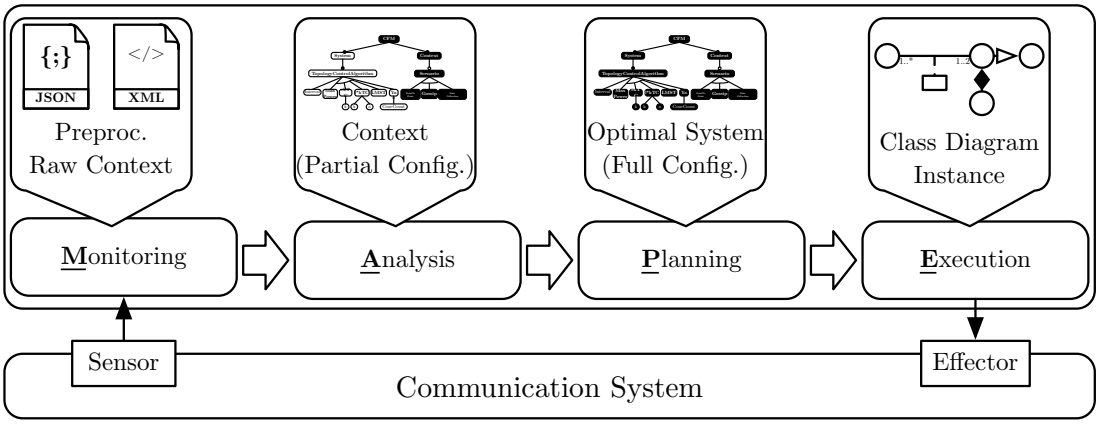


Figure 7.6.: MAPE activities in the MILP-based REACT Loop. Preproc.: Preprocessed [225].

7.2.4. Implementation of the MILP-Based Feedback Loop

This section outlines the implementation details of the MILP-based feedback loop. It is also implemented using Java. As the approach is able to work with SAT and MILP solvers, the planner supports MiniSAT [228] as well as IBM's

CPLEX Optimizer[3]. The CFM-based specification is implemented using an enhanced version of CardyGAn [229]. CardyGAn provides tool support for the specification and validation of cardinality-based context feature models. The approach proposes a domain-specific language, which employs Eclipse Xtext [230] for the implementation. CardyGAn provides an Eclipse plugin for creating CFMs using a textual editor with syntax highlighting and checking, advanced anomaly detection, and the possibility to generate sample instances of the model [229]. The possibility of (multiple) feature instances is omitted. The class diagram is a UML2 class diagram representing the managed resource. REACT parses the UML class diagram as an XML file complying with the *UML 2 Abstract Syntax Metamodel* by the OMG. Due to this standardized format, the system developer can create the XML file manually or use a graphical editor that offers an export in this format, such as Papyrus[4]. For the mapping between problem and solution space, a Java properties file is employed. This file matches attributes and features in the problem space to properties and classes in the class diagram. Going from the functional constraint to the non-functional constraints, the performance goals are also expressed using a Java property file containing the name of a non-functional property as key, and the corresponding weight as value. As final element of the knowledge, there are the performance influences. In order to learn the performance influences offline, we used the tool SPL Conqueror [231]. SPL Conqueror uses a CFM in combination with previously generated data containing different CFM configurations and their corresponding values of the available non-functional properties. It then uses machine learning techniques for learning the influence of the different feature and parameter selections on the non-functional properties. It outputs a performance-influence model in the form of a regression equation for each metric. REACT directly supports the output files of SPL Conqueror, which eases the application of the MILP-based REACT Loop in combination with the learned performance-influence models.

7.2.5. Evaluation of the MILP-Based Feedback Loop

This section presents the result of evaluating the MILP-based approach with respect to effectiveness, efficiency, and applicability using adaptive WSNs as an

[3] `http://www.ibm.com/analytics/cplex-optimizer`, accessed 2020-12-08
[4] `https://www.eclipse.org/papyrus/`, accessed 2020-12-08

evaluation scenario. REACT Core's context management module has not been used in this evaluation, as the effectiveness and efficiency of the loop itself are evaluated. As the context module often skips the loop, it hinders the measurement of the feedback loop's actual runtime.

Use Case Description

The goal of the use case is to optimize the latency in a WSN, as low latencies, e.g., help all nodes to have a consistent, up-to-date view of the network. A WSN consists of sensor nodes trying to detect specific events, such as high levels of water in a flood warning system [101]. As each node can reach a certain number of nodes in the neighborhood, the nodes form a topology. This topology influences the performance of the network, e.g., in terms of latency and robustness. The more nodes can be reached by using a higher sending power, the higher the energy consumption. Using topology control algorithms [232] enables exploiting these properties by changing the topology balancing energy consumption and performance goals, such as latency in our case.

In this evaluation, the adaptation logic configures a topology control algorithm in the WSN. Depending on the current topology as well as on the mobility and the mode, different algorithms and parameters yield the lowest latency and battery drain. The complete feature model is shown in Figure 7.7. The system features include algorithms with and without additional attributes. In total, there are five algorithm features, five attributes—being either integer- and real-valued—and two features determining whether UDP or TCP is used [225]. The context of the WSN is represented by different scenarios. First, there is the possibility that all WSN nodes send data to a single base station in a many-to-one fashion called *DataCollection* scenario. Otherwise, there is one-to-one communication, either deterministic in a *PointToPoint* scenario or probabilistic in a *Gossip* scenario. Additionally, nodes can be mobile, which is represented by a *Mobility Speed* attribute. The context also represents the world size and topology density using integer attributes. Finally, the WSN can either be in normal or emergency mode. Depending on the objective of the WSN, the emergency mode is enabled in the case of flooding or a fire, which indicates the nodes to send messages as fast and with the highest sending power as possible. The rules stated at the top left of the figure shows cross-tree constraints of the WSN setting.

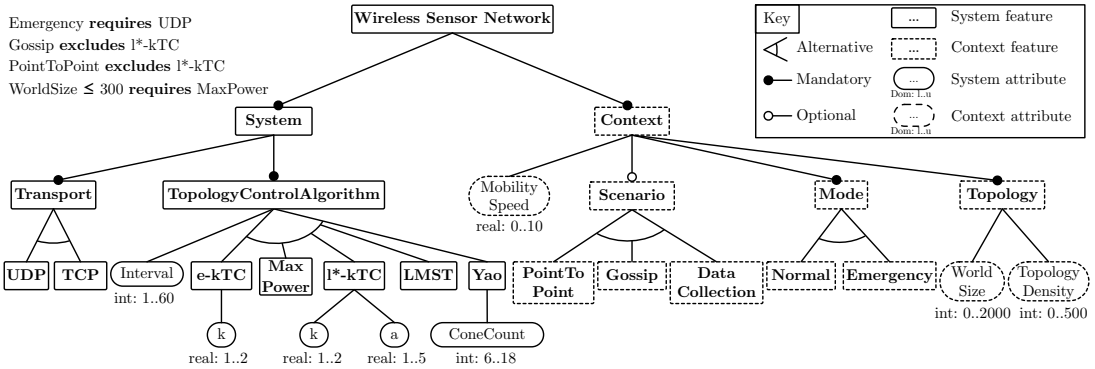


Figure 7.7.: CFM of the wireless sensor network use case [225].

Evaluation Settings

We state the following evaluation questions:

EQ1–Effectiveness: How much does our approach improve the system performance with respect to the desired performance goal in terms of non-functional properties? What is the influence of the performance-influence model size on the same system performance?

EQ2–Efficiency: How does the size of the performance-influence model affect the runtime of the planning component?

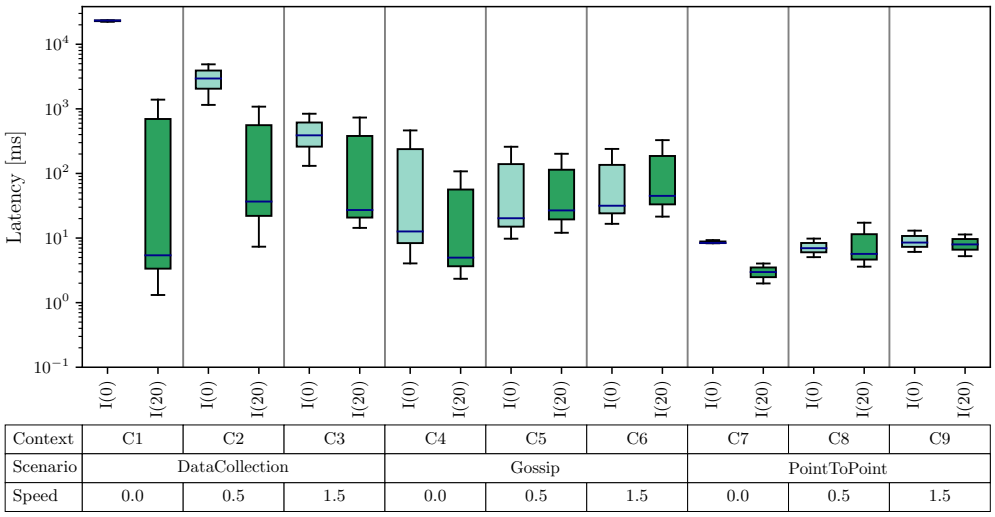
EQ3–Applicability: Is the approach applicable to communication systems in general?

In our case, we translated the CardyGAn model into a representation that is compatible with SPL Conqueror and created data points with configurations and values of the non-functional properties using a Java WSN simulator based on the Simonstrator platform [233]. The value ranges of the configurations are determined by the applied CFM. All measurements were run on a 64-bit Windows 7 workstation, equipped with an Intel i7-2600 CPU ($2 \times 3.7\,\text{GHz}$) and 8 GB of RAM, with 2 GB being assigned to the simulator. For creating the training data, we executed 6125 simulation runs representing a fixed configuration in terms of system and context features. Using each configuration, we measured the mean latency of the network in the simulation. When evaluating the approach, for each run, 100 nodes were distributed randomly onto a square region. Then, either the data collection node in case of a data collection scenario or a random node is selected

as the planner node. The planner decision is distributed to all other nodes in case of a reconfiguration. After a reconfiguration, the simulation proceeds with the execution of the selected topology control algorithm. The first two evaluation questions are answered quantitatively, while *EQ3* will be discussed qualitatively.

Evaluation Results

This section outlines the evaluation results aiming at answering the first two quantitative evaluation questions by presenting an overview of the respective measurements. *EQ3* is discussed qualitatively in Section 7.2.6.



Context	C1	C2	C3	C4	C5	C6	C7	C8	C9
Scenario	DataCollection			Gossip			PointToPoint		
Speed	0.0	0.5	1.5	0.0	0.5	1.5	0.0	0.5	1.5

Figure 7.8.: Latency vs. context class for I(0) and I(20) [225].

Looking at *EQ1*, the main goal is that the system performance using the adaptation logic is better than a MILP-based baseline system without performance influences creating random valid configurations. First, we compare the effectiveness, and then we look at the trade-off between performance and training cost.

Figure 7.8 shows all possible context combinations of the *Scenario* and the *Mobility Speed* attribute on the x-axis named C1 to C9. All boxplots in this thesis follow the convention that (i) the blue horizontal line indicates the median value, (ii) the lower and upper caps mark the 25 %- and 75 %-percentile, and (iii) the whiskers enclose all values within the 1.5 inter-quartile range. The y-axis represents the measured latency on a logarithmic scale. For each context, the boxplots show the MILP-based REACT Loop working without performance influences on the

left (I(0), light green) and the same loop with 20 training iterations on the right (I(20), dark green). Accordingly, the baseline system I(0) generates valid configuration. However, in this case, the planner cannot optimize the result towards non-functional goals. As we can see in the plots, using the trained MILP-based REACT Loop results in lower median latencies compared to the baseline system in seven of nine context classes. In the other two context classes as part of the Gossip scenario, the median latency is lower using the baseline system.

Taking the training cost for achieving this effectiveness into account, Figure 7.9 shows the performance in the representative context class C2 (i.e., DataCollection and MobilitySpeed = 0.5 m/s) with different numbers of iterations. We can see that the performance stabilizes after ten iterations, and in the case of 35 iterations, the performance decreases again.

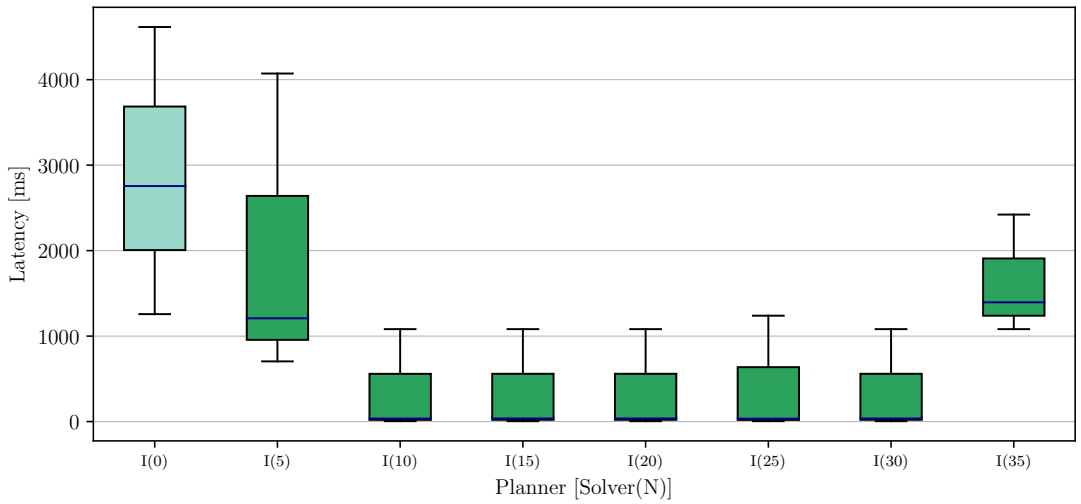


Figure 7.9.: Latency vs. training cost for context class C2 [225].

Next, as part of *EQ2*, this section looks at the scalability of the approach considering varying sizes of the used performance-influence models. Figure 7.10 presents an overview of the mean planning duration with a changing number of iterations resulting in smaller or larger performance-influence models. In this case, we have two baseline values: M(0) representing a MiniSAT-based measurement as well as I(0), which, again, is the baseline system without a performance-influence model. As the first M(0) represents MiniSAT, it cannot handle, e.g., integer or real values. For this measurement, a test data set with 1215 simulation runs is used to explore combinations of different context features and attribute values. In this case, each

one of the nine planner settings shown in the figure is executed in the different combinations. The 1215 simulation runs correspond to 135 different context settings times the shown nine planner settings. The use of MiniSAT naturally results in the shortest planning duration, while the use of performance-influence models directly increases the planning time compared to I(0).

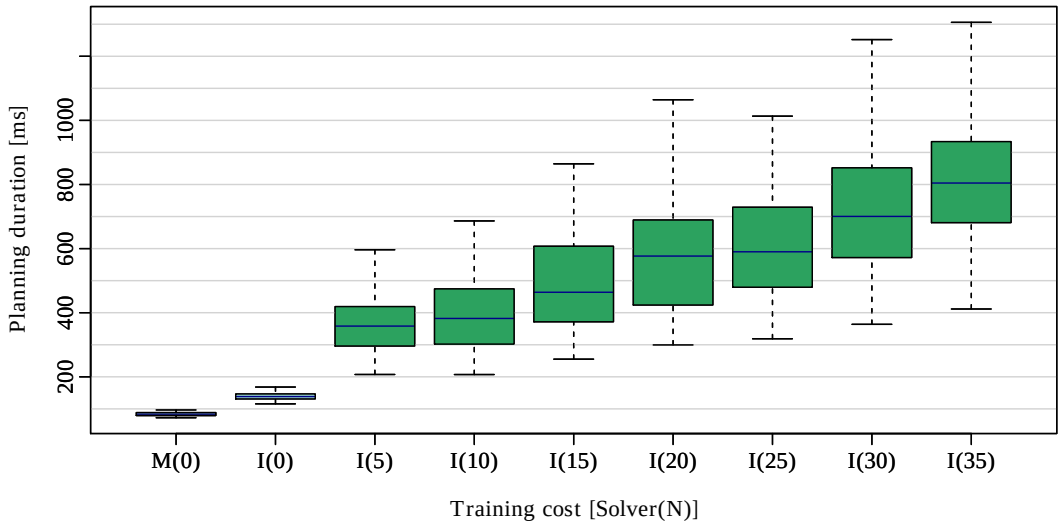


Figure 7.10.: Planning duration vs. training cost [225].

7.2.6. Discussion of the MILP-Based Feedback Loop

In this section, we discuss the presented evaluation results. This includes an analysis of the three evaluation questions and their results, as well as an overall discussion of the MILP-based REACT Loop.

Considering the evaluation of the effectiveness as part of *EQ1* and Figure 7.8, by looking at the median values, we conclude that the MILP-based feedback loop using performance-influence models considerably improves the latency in most contexts. Looking at Figure 7.9, we can see that our approach also overfits the data in the case of 35 iterations. This means it still must be manually determined how many learning iterations should be taken into account, as when applying general machine learning techniques. Additionally, by looking at the other data points, we observe that the approach is robust in its performance between 10 and 30 iterations. This robustness shows that there exists a larger range for the

number of iterations in which the results are similarly optimal. Accordingly, this fact simplifies to find the optimal number of iterations for the system developer.

Considering *EQ2*, Figure 7.10 shows a dependency between the learned performance-influence models and the planning duration. Looking at the baseline setups, the MiniSAT-based planner needs approximately 40 % less time compared to the MILP solver. As expected, handling real and integer values directly increases the planning time compared to a SAT solver. If performance-influence models are used, adding more and increasingly complex constraints to the solver also increases the planning duration. The results indicate that systems that need faster adaptations should rely on either a small number of training iterations, omit them, or use a SAT-based solution. Hence, according to the use case, there is a tradeoff between planning duration and optimality considering non-functional goals.

Evaluating the applicability as part of *EQ3*, as the adaptation behavior of the MILP-based REACT Loop is also completely dependent on the information of the knowledge component, we state that it is a reusable approach [225]. The learning method is also generic and can be used with other simulation environments or real systems. Additionally, updating the learned performance-influence models at runtime is possible in theory. As we have seen in Chapter 6, REACT Core already provides the needed facilities to update the knowledge at runtime. Accordingly, if an external toolchain for online learning using SPL Conqueror is available, changing the performance-influence models at runtime in an instance of REACT is enabled with a single API call. However, employing an online learning feedback loop is out of scope of this work. Finally, the MILP-based REACT Loop is inherently applicable to more managed resources than the SAT-based REACT Loop due to the means to plan numeric attributes.

Summing up, one problem of the MILP-based REACT Loop is the missing multi-instantiation of features defined in the CFM. In general, it is possible to define that features can occur multiple times in a system configuration. However, the encoding of the MILP problem presented as part of [225] cannot handle this multi-instantiation. When performance-influence models should be used, a simulation for generating data before runtime or enough data from a system at runtime is needed. Depending on the size of the problem space, this requires many of data points for SPL Conqueror to work. As a third and final REACT Loop, this thesis proposes a constraint programming-based REACT Loop in the following section.

7.3. CP-Based Feedback Loop

The last REACT Loop uses constraint programming (CP) internally for planning adaptations. This section is based on [11] and [186][5].

7.3.1. Constraint Satisfaction Problems

First, we define the class of CSPs. A CSP consists of a set of variables, a domain for each variable, as well as a set of constraints restricting the values of the variables [234, 235]. A feasible solution of a CSP is "*an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied*" [234]. If there is no such assignment, the problem is unsatisfiable.

Until here, this only describes a problem of satisfiability or feasibility. In order to optimize the result turning the CSP into an optimization problem, an objective function must be added [234]. For solving a CSP, including optimization goals from an objective function, CP is used. In general, there are the possibilities of finding any solution, or an optimal or near-optimal solution given these additional objectives. The objective function adds the notion of minimizing or maximizing a variable given the constraints of the problem. This section describes a CP-based REACT Loop. For a detailed comparison between mathematical and constraint programming, the reader is referred to [235].

7.3.2. CP-Based Context-Aware Feature Modeling Approach

Again, the already presented CFM-based modeling is applied here. In this case, the CFM gets transformed into a CSP. As with the MILP-based approach, attributes can have integer and real values. It is also possible to employ multi-objective optimization. As an addition to the MILP-based transformation, it is possible to have multiple instances of a feature present in a configuration. This allows for more complex specifications, as the expressiveness of the CFMs is increased.

[5] [11] and [186] are joint works with M. Breitbach, C. Krupitzer, M. Weckesser, C. Becker, B. Schmerl, and A. Schürr.

7.3.3. CP-Based Architecture

Figure 7.11 presents the architecture of the CP-based approach. The CP-based REACT Loop requires two models for separating problem and solution space:

1) The **adaptation options specification**, which is an explicit representation of valid reconfiguration options. It thus describes the problem space with a structural modeling language, including constraints.

2) The **target system specification** models the architecture of the managed resource, i.e., the solution space. After solving a problem in the problem space, the CP-based REACT Loop maps the result to the solution space according to the target system specification.

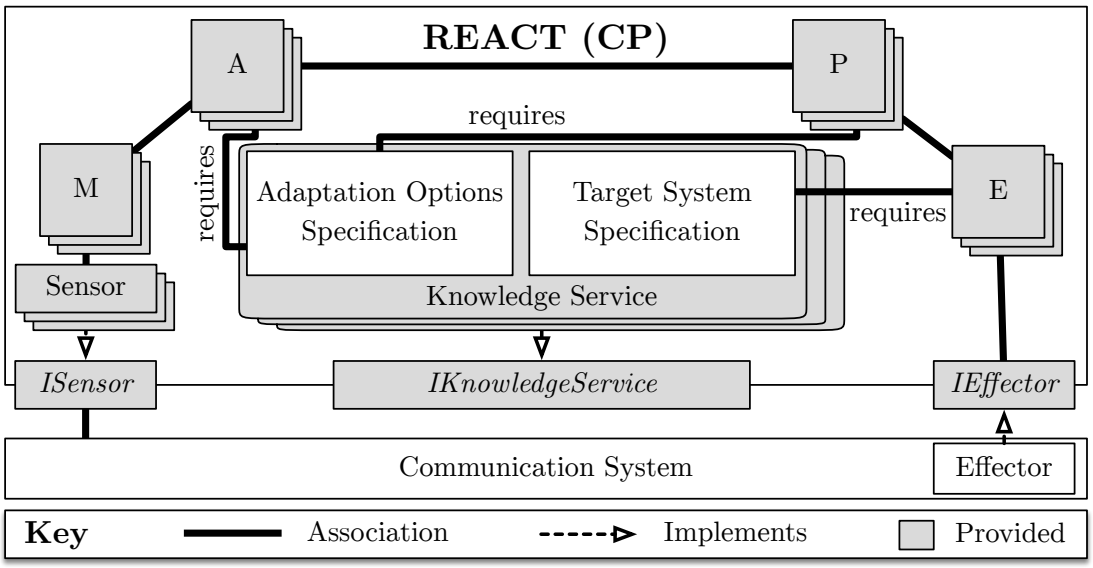


Figure 7.11.: Architecture of REACT using the CP-based REACT Loop [11].

The CP-based REACT Loop uses the live sensor data provided by the communication system together with the adaptation options specification to adapt the system to the desired target state. The feedback loop instance is reusable since it works with arbitrary adaptation options specifications and target system specifications. It enables multiple instances of features and does not need an explicit mapping model. For the mapping, the CP-based REACT Loop uses an automatic mapping by feature/class names and by attribute/property names.

Going through the loop, the sensor receives raw input such as JSON or XML. The monitor preprocesses this data by transforming it into higher-level objects.The

monitor instance passes the higher-level objects to the analyzer. The analyzer uses the adaptation options specification for mapping the sensor data to the specification. Hence, it creates model instances from the sensor data. These model instances are directly passed to the planner. There, the adaptation options specification gets combined with the model instances from the analyzer. As these model instances represent the context state of the managed resource, in combination with the adaptation options specification, this represents the CP. The planner now finds a solution using the CP solver. This solution is a (completed) model instance, which has to be mapped to the target system specification, which represents the managed resource. Contrary to the MILP case, the CP-based REACT Loop uses an implicit name-by-name-based mapping following a convention-over-configuration [236] approach. This mapping is the responsibility of the executor. The instance of the target system specification is then transferred to the effector. Finally, it is again the responsibility of the managed resource to implement the changes.

7.3.4. Implementation of the CP-Based Feedback Loop

This section outlines the implementation details of the CP-based feedback loop. The section includes a description of the modeling capabilities using the modeling language Clafer [237] and an outline of the functionality of the feedback loop.

Modeling Capabilities

The essential parts of the CP-based REACT Loop are the used models of the adaptation behavior (adaptation options specification) and of the managed resource (target system specification). The system developer provides these models at design time and may update them at runtime. The CP-based REACT Loop uses the models at runtime to adapt the managed resource. Specifically, the feedback loop supports adaptation options specifications in the structural specification language Clafer (**cla**ss, **fe**ature, **r**eference) [237]. There are multiple reasons to use Clafer. First, it is a well-established approach applied in different domains [237, 238], which is available as an open source project and extensively documented. Second, Clafer provides lightweight modeling capabilities with just a minimal set of concepts. Thus, Clafer makes modeling accessible to users from different domains without extensive modeling experience. Third, Clafer provides

model verification and validation [239]. By using Clafer, the CP-based feedback loop offers the possibility for advanced analysis as presented in [225]. Thus, the system developer is enabled to minimize modeling errors in the Clafer specifications. Clafer specifications not only can be translated into CSPs but also into SAT and satisfiability modulo theories (SMT) [240] problems [241]. We decided to use Clafer in combination with its CSP-based backend.

A Clafer-based model is created using a single type of element named Clafer [237]. A Clafer represents a type, an attribute, a relationship, an instance, or a combination of these. Each Clafer has a name and is either top-level or nested under other Clafers. Nesting is expressed using indentation. We illustrate Clafer's basic modeling capabilities with the following use case from a cloud server management scenario, where a system developer uses REACT with the CP-based feedback loop to implement adaptive behavior. Based on the context dimensions (i) number of running servers, (ii) total number of servers, and (iii) average response time, REACT launches additional servers adaptively if required. The launch of an additional server happens if the average response time exceeds a threshold value (here 75, representing 75 ms) and additional servers are available.

Listing 7.1 shows an exemplary adaptation options specification in Clafer for this use case. Line 1 contains a (top-level) Clafer named `ServerLauncher` that describes that an additional cloud server should be started. Clafers may have instance cardinalities, while the default instance cardinality is 1. By adding `0..1` to Line 1, we specify that model instances are valid with either none or only one `ServerLauncher` Clafer. Clafers may be abstract. An abstract Clafer "*aggregates commonalities*" [238] like a class in object-oriented programming. Hence, a Clafer can inherit from an abstract Clafer and use abstract Clafers like a type. Lines 2-5 describe an abstract entity of type *Context* with integer attributes. A solution to this problem space requires to have precisely one instance of this Clafer with all attributes set. Lines 6 and 7 define the auxiliary Clafers `ExtraServers` and `HighRT` that state whether it is possible to start an additional server and whether the response time is high. In addition, a Clafer model may contain constraints in brackets. Lines 8-9 specify constraints that set the auxiliary Clafers `ExtraServers` and `HighRT` according to the context. Line 10 is the adaptation rule stating that the `ServerLauncher` Clafer should be present in a model instance if the response time is high and more servers are available.

```
1  ServerLauncher 0..1
2  abstract Context 1
3      servers -> integer 1
4      maxServers -> integer 1
5      responseTime -> integer 1
6  ExtraServers 0..1
7  HighRT 0..1
8  [
9    if Context.servers < Context.maxServers then one ExtraServers else no
         ExtraServers
10   if Context.responseTime >= 75 then one HighRT else no HighRT
11   if HighRT && ExtraServers then one ServerLauncher else no ServerLauncher
12 ]
```

Listing 7.1: Adaptation options specification in Clafer for the self-adaptive cloud server management case [11].

The CP-based REACT Loop uses separate models for the adaptation behavior, which is modeled in Clafer, and the managed resource. This induces the need for a mapping from the problem space to the solution space, which represents the managed resource. The CP-based REACT Loop uses the target system specification, which the system developer provides in UML as class diagrams, as in the case of the MILP-based REACT Loop. As mentioned, in many cases, a UML model of a managed resource might already exist and could be ready to use as a target system specification for REACT decreasing the development effort. In addition, an automated creation of a UML model from source code might also reduce the time for modeling. In the cloud server management example with its adaptation options specification in Listing 7.3.4, the simplest UML model only contains a single class named `ServerLauncher`. The classes as part of the UML model indicate if they should be present in the managed resource or not.

Feedback Loop

The previous section described the modeling of the adaptation options specification in Clafer and the target system specification in UML. Now, we show how REACT leverages these use case dependent models to achieve self-adaptivity using the CP-based REACT Loop. Figure 7.12 shows the behavior of the CP-based REACT Loop in the aforementioned cloud server management example. The

feedback loop starts when new sensor information is received via the sensor interface in JSON format. In the example, this sensor data ❶ is context information about the cloud system. The received information is handed over to the monitor.

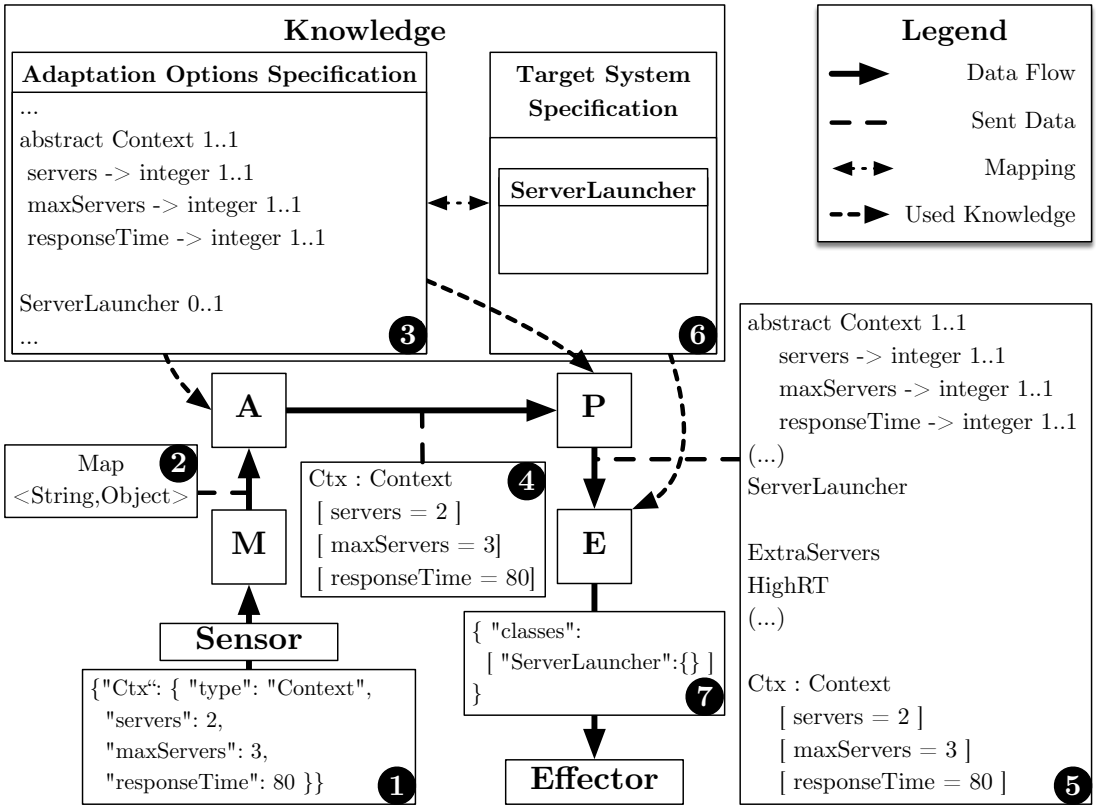


Figure 7.12.: An adaptation cycle of REACT using the CP-based REACT Loop in the cloud server management example. The analyzer maps the JSON-based sensor information to the adaptation options specification in Clafer. The planner evaluates the model and finds a valid instance. Here, it adds a `ServerLauncher` Clafer as starting a new server is desired. The effector maps the plan to the target system specification in UML and transfers the adaptation to the managed resource [11].

The implementation of the CP-based monitor allows system developers to choose from multiple monitoring strategies. In the *default* strategy, the monitor parses the raw JSON data and hands it to the analyzer as a map ❷. The loop also offers an *aggregation* strategy that additionally aggregates information from multiple sensors and a *windowing* strategy that applies a sliding window approach to the sensor values. The `IMonitoringStrategy` interface shown in Listing 7.2 further enables to create, share, and integrate custom monitoring strategies.

```
1 public interface IMonitoringStrategy {
2   public Map<String, Object> handleSensorJSON(String sensorData);
3 }
```

Listing 7.2: IMonitoringStrategy interface.

The analyzer fetches the adaptation options specification ❸ from the knowledge service. It uses the abstract Clafers specified in the adaptation options specification to create concrete Clafers from the monitoring data. To achieve this mapping, the original sensor data contains `type` attributes. The CP-based REACT Loop uses these `type` attributes to map the monitoring data objects to the correct abstract Clafers in the adaptation options specification. In the exemplary case, the type has the value `Context` and REACT, therefore, maps it to the `Context` Clafer in the adaptation options specification ❸. The concrete Clafers are then forwarded to the planning component ❹.

The planner merges the generated Clafers with the adaptation options specification to the problem specification. Thus, this specification contains the global constraints of the adaptation options specification and the current constraints imposed by the sensor data. Now, the planner solves this CP using the Java-based library Chocosolver [242]. Hence, the solver finds a model instance ❺ that satisfies all constraints. In the exemplary case, this model instance would either contain or not contain the `ServerLauncher` Clafer, which constitutes the adaptation decision.

The planning result in the form of concrete Clafers is then passed to the executor, which maps the Clafers to the target system specification ❻. Then, the CP-based REACT Loop maps the Clafers by name to the classes or parameters of the UML model and creates a UML instance. In the example, the created `ServerLauncher` Clafer (note the missing 0..1 cardinality in ❺) is mapped to the class `ServerLauncher` of the target system specification. The executor transforms this UML instance to a language-independent representation. Finally, the executor passes this representation via the effector interface ❼ to the managed resource, where adaptations will take place. The CP-based feedback loop works with arbitrary adaptation options specifications and target system specifications and is thus applicable to a wide range of scenarios.

7.3.5. Evaluation of the CP-Based Feedback Loop

This section evaluates the implementation of REACT employing the CP-based feedback loop. The REACT Core's context module has not been used as part of this evaluation for the same reason as in the MILP case. As we want to measure the effectiveness and efficiency of the loop itself, skipping runs of the loop when using the context module does not enable us to measure the runtimes easily. First, we compare the system with Rainbow, a well-known and frequently applied framework for model-based adaptation. For doing so, we implemented the simulation-based SEAMS exemplar SWIM (Simulator for Web Infrastructure and Management) [216], which represents a cloud system. Second, this section presents the application of REACT using the CP-based REACT Loop in an emulated communication system in the field of SDN.

In our first experiment, we compare REACT using the CP-based feedback loop with the well-known Rainbow framework [17] in terms of development effort, performance, and features. The second experiment aims at the application of REACT using the CP-based REACT Loop in a real-world SDN use case. We try to answer the following evaluation questions, while *EQ1-EQ3* are tackled in the first experiment and *EQ4* in the second one.

EQ1–Development Effort: How does REACT using the CP-based REACT Loop compare to Rainbow in terms of development effort?

EQ2–Performance: How does REACT using the CP-based REACT Loop compare to Rainbow in terms of performance?

EQ3–Capabilities: How does REACT using the CP-based REACT Loop and Rainbow differ in terms of capabilities?

EQ4–Real-World Effectiveness: Can REACT using the CP-based REACT Loop be implemented and used effectively in a real-world communication system?

Cloud Server Management

In our first experiment, we compare REACT using the CP-based feedback loop with the well-known Rainbow framework [17] in terms of development effort, performance, and capabilities. Accordingly, the following two paragraphs introduce the Rainbow framework in more detail.

The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems, with components implementing each aspect of the MAPE-K loop. Probes are used to extract information from the managed resource that update the model via gauges, which abstract and aggregate low-level information to detect architecture-relevant events and properties. This separation means that the same code for Rainbow can be used across multiple deployments of the system by only changing probes (and effectors). Evaluators check for satisfaction of constraints and properties in the model and trigger adaptation if any problems are found, (e.g., the response time falls below some threshold or the cost of deployment becomes too high). The adaptation manager, on receiving the adaptation trigger, chooses the "best" adaption plan to execute, and passes it on to the strategy executor, which executes the strategy on the managed resource via effectors.

The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as reflected in the models. The underlying decision making model is based on decision theory and utility [18]; varying the utility preferences allows the system developer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [99], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the managed resource with variable execution time. A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the managed resource. As a framework, Rainbow can be customized to support self-adaptation for a wide variety of system types. Furthermore, the flexibility of the framework has enabled not only the multi-object trade-off selection of strategies among competing objectives that is embodied in Stitch, but has also supported research into online adaptation planning [243], predictive proactive adaptation [244], and human-machine cooperation [245].

In this experiment, REACT using the CP-based REACT Loop and Rainbow adapt a cloud server deployment providing a web application. This experiment uses SWIM [216], which offers a reproducible way for evaluating adaptation logics in a web server environment. It is a simulation environment based on OMNeT++ [246]. The SWIM exemplar consists of multiple simulated web servers

connected to a round-robin load balancer. The load balancer distributes simulated requests, and the corresponding server simulates the execution. Each web server response may contain optional content (e.g., advertisements), which increases the response time but also leads to additional revenue for the web site operator. This optional content is represented using a so-called dimmer value, which states in percent how many requests contain the optional content. The overall goal of the system is thus continuously reaching a fixed response time goal, maximizing the revenue with the optional content, and minimizing the cost for the servers. Accordingly, there are two ways of adapting the running system: 1) Adding or removing servers, and 2) controlling the percentage of responses with optional content. We use the "1998 World Cup Web Site Access Logs" trace provided by SWIM for the comparison.

In accordance with [18] and [247], we measure the required source lines of code (SLOC) for implementing the SWIM use case with REACT using the CP-based feedback loop and Rainbow. The SLOC comprise the specification files and the interface implementation for connecting the respective approach to SWIM. Further, we measure the cycle time for executing an adaptation in REACT using the CP-based REACT Loop and Rainbow as well as the processing time of each MAPE activity. We conduct ten evaluation runs each for REACT and Rainbow on a machine equipped with an Intel Core i7-8700k and 32GB of RAM. Both approaches have been executed in Docker[6] containers. For better comparability, REACT and Rainbow perform similar adaptations, leading to the same response times and simulated costs for the web site operator in SWIM.

Looking at *EQ1*—the development effort—two metrics influence the system developer's experience: the lines of code required to achieve self-adaptivity and the number of different programming languages, tools, and technologies she needs to be familiar with. Both metrics apply to i) specifying the adaptive behavior and ii) implementing the interfaces to SWIM. Table 7.3 shows the SLOC for the specification of the adaptive behavior.

We observe that specifying the adaptive behavior with REACT requires considerably fewer SLOC. The system developer has to write 152 SLOC in 2 files with clear responsibilities. To achieve the same behavior with Rainbow, the system developer has to write 593 SLOC in 6 different files using various languages. Next,

[6] `https://www.docker.com/`, accessed 2020-12-18

we assess the development effort for the interface implementation. In Table 7.4, we observe that REACT requires 200 SLOC and Rainbow requires 204 SLOC. However, REACT requires fewer (configuration) files for setting up the connection. In addition, due to its language-independent interfaces, system developers can use their preferred language.

Rainbow			**REACT**		
Artifact	**SLOC**	**Language**	**Artifact**	**SLOC**	**Language**
Strategies and tactics	113	Stitch	Adaptation options specification	123	Clafer
Utilities	55	YAML			
Architecture Model	261	YAML	Target system specification	38	XML
	128	ACME			
	25	DTD			
	11	XML			
Total	**593**		**Total**	**152**	

Table 7.3.: SLOC measurements of the modeling in Rainbow and REACT using the CP-based REACT Loop [11].

Rainbow			**REACT**		
Artifact	**SLOC**	**Language**	**Artifact**	**SLOC**	**Language**
Probes	91	Perl	Interfaces	200	Python
	68	YAML			
Effectors	9	Bash			
	25	YAML			
Utility Files	11	Bash			
Total	**204**		**Total**	**200**	

Table 7.4.: SLOC measurements of the interface implementations of Rainbow and REACT [11].

For answering *EQ2*—looking at the performance—Figure 7.13 presents the average runtimes per MAPE activity as well as their average sum. We observe large differences in the different phases as well as in the average total time of the feedback loop. REACT using the CP-based REACT Loop considerably outperforms Rainbow in the monitoring and analyzing phase. However, the planning phase of the CP-based feedback loop needs more time compared to Rainbow. Still, looking at the total time, the average total runtime of the REACT's feedback loop using the CP approach is considerably lower. In total, the average adaptation cycle execution when using the CP-based REACT Loop requires 84 ms in comparison to 216 ms in Rainbow.

Finally, EQ3 considering the capabilities is answered qualitatively. Rainbow is based on architecture models that enable a more in-depth analysis and a less complex planning phase as a result. In addition, it works utility-based with

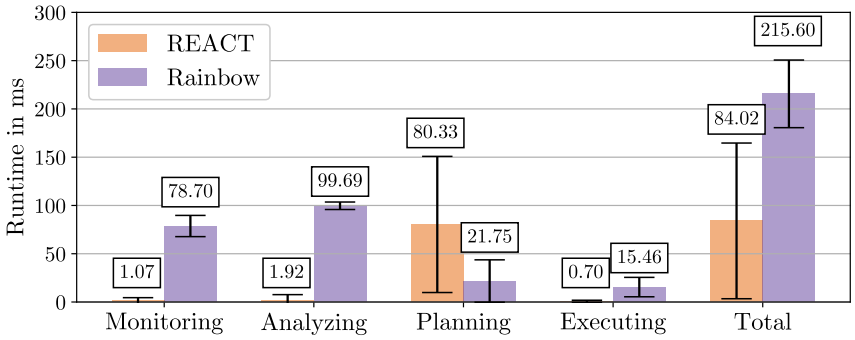


Figure 7.13.: Average run times of the MAPE activities of REACT using the CP-based REACT Loop and Rainbow [11].

the possibility to weight optimization goals. This enables system developers to specify precisely how vital a non-functional goal should be. Currently, Rainbow itself is deployed centrally and requires a higher runtime in comparison with REACT and the CP-based REACT Loop in the use case. The specification of the adaptive behavior requires more files in different file formats compared to REACT using the CP-based REACT Loop. On the other side, REACT offers runtime modifications, which enables to update the results of the adaptation logic at runtime. Also, REACT provides means for setting up deployments following decentralized control, and it directly supports multiple programming languages due to the used IDL. However, it is not possible to weight goals in the CP-based REACT Loop.

SDN-Based Wifi Handover

In the second experiment, we show REACT's focus on communication systems in a real-world SDN-based use case adding adaptive behavior to an underlay network. Sensor information from two distributed hosts is pushed to a decentralized adaptation logic following the *regional planning* pattern [15].

In this scenario, a car receives a live stream from a streaming server via a wireless network connection (see Figure 7.14). With each handover between the wireless network towers along the road, the user in the car experiences packet loss. The goal is to improve the quality of experience by minimizing the packet loss during the handover. SDN "*is a paradigm where a central software program, called a controller, dictates the overall network behavior*" [248]. The controller manages a set of controllable switches. These switches deal with incoming packets according

to flow rules. A flow rule can, for instance, forward a packet to a specific port, change or add packet headers, or implement firewall functionality by rejecting a packet. The SDN controller offers an API that allows system developers to write applications for the controller. In our case, we apply these capabilities by monitoring and adapting the flow rules with REACT for seamless handovers. A specific adaptation means that there should be flow rules for the current wireless tower, as well as flow rules duplicating the streaming traffic to the next tower. This duplication should only take place when the car is going to leave the radio range of the first tower soon. Achieving this behavior continuously requires a recurring adaptation of the flow rules.

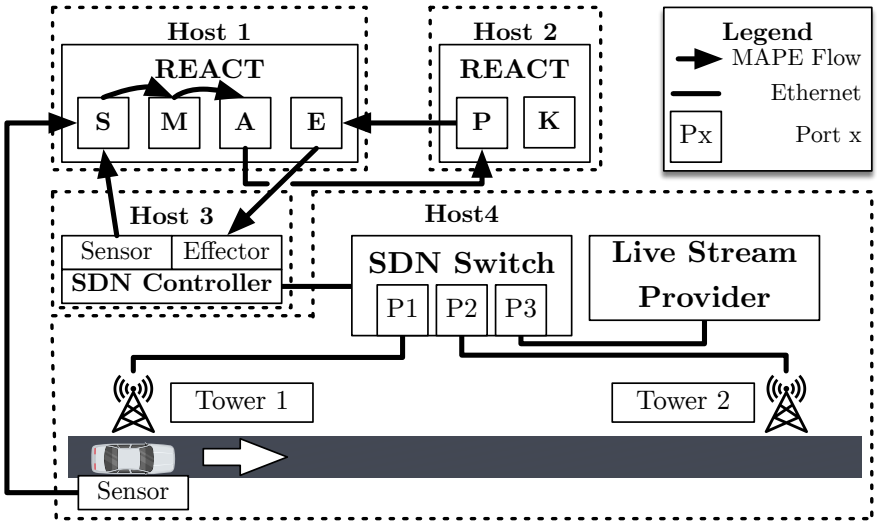


Figure 7.14.: SDN handover setup with two access points in a live streaming scenario with multiple sensors. The SDN sensor sends topology information while the car sends its distance to the currently connected radio tower. REACT creates flow rules for adaptively duplicating the live stream traffic [11].

REACT receives sensor data from two sources. First, the sensor SDN application sends the host location, which contains addressing information as well as the currently connected network tower, to REACT. An additional sensor in the car sends the distance to the currently connected access point to REACT every second to minimize the network traffic duplication. We apply the built-in *aggregation* monitoring strategy of the CP-based monitor to combine the data for reasoning. The MAPE components are distributed according to the regional planning pattern [15]. Monitor, analyzer, and executor are deployed on a separate

machine (Host 1). A powerful and stable resource runs the computationally intensive planner and the knowledge service (Host 2).

We use the ONOS [249] SDN controller in this evaluation, which runs on another separate machine (Host 3). The network was emulated with Mininet-Wifi [217] on a fourth machine (Host 4). In a pre-test, we used the VLC player[7] for streaming a 4K video. However, for better controllability and reproducibility of the experiment, we run Iperf[8] in UDP mode with 25 Mbit/s, the bandwidth recommendation of Netflix for 4K video streams[9]. The ethernet connections have a bandwidth of 100 Mbit/s. We emulated four access points, one moving wireless node as the car, and a static host representing the live stream provider.

We compare the self-adaptive handover with REACT to ONOS' reactive forwarding application. The reactive forwarding application deploys flow rules on switches if a host connects to another. In this case, the corresponding switches would request the controller to decide how the packets should be handled. The reactive forwarding application subscribes to a corresponding event and deploys flow rules handling these packets on the switches. We measure the packet loss with REACT and ONOS' reactive forwarding in 30 runs each.

As shown in Figure 7.15, self-adaptivity with REACT reduces the packet loss considerably. The aggregated mean packet loss of the overall simulation time improves from 4.87 % in the reactive forwarding case to 0.48 % with REACT.

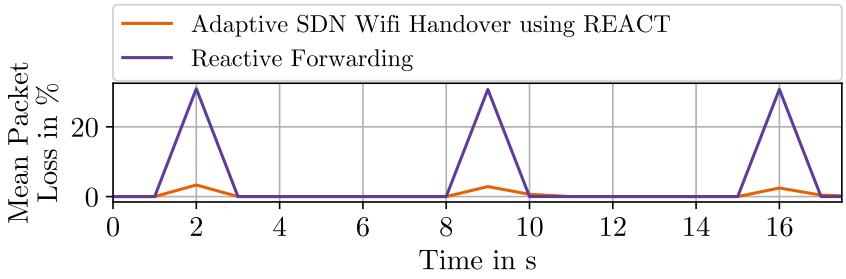


Figure 7.15.: Average packet loss in % over time. The duplication flows are added as soon as the access point is about to switch [11].

[7] `https://www.videolan.org/vlc/`, accessed 2020-12-18
[8] `https://iperf.fr/`, accessed 2020-12-08
[9] `https://help.netflix.com/en/node/306`, accessed 2020-12-18

7.3.6. Discussion of the CP-Based Feedback Loop

This section discusses the evaluation, including the four evaluation questions presented in the previous section. For this, the structure follows the four evaluation questions *EQ1* to *EQ4*.

Comparing the development effort in terms of the modeling and interface implementation, Table 7.3 and 7.4 show considerable differences. In case a system developer wants to minimize the effort to adapt a system, REACT with the CP-based feedback loop should be chosen. It needs fewer different files for specifying the adaptation behavior and the interfaces. Another important point is that the IDL-based interfaces of REACT Core enable the utilization of a large number of different programming languages.

Next, discussing the performance measurements presented in Figure 7.13, we can answer *EQ2*. First, since Rainbow holds an exact architecture model of the managed resource, it updates the model when new sensor data is available, periodically checks for problems, including an analysis where the problem is located in the model, and triggers an adaptation. Thus, this design choice bears a more complex analysis of the managed resource's architecture at the cost of slower adaptation. The total execution time of an adaptation cycle in REACT using the CP-based REACT Loop is determined to a high degree by the planner component. This is not surprising, as the planner executes Chocosolver to find a valid model instance. Clafer itself scales well with increasing problem size even with models of several thousand Clafers [241, p. 84]. In Rainbow, the complex problem analysis in the monitoring and analyzing component accelerates planning. The planner only uses the utility function and expected outcomes for selecting one of the specified strategies instead of running a solver. Taking the total time of the feedback loops into account, we argue that REACT using the CP-based approach is well-applicable in scenarios where fast adaptation is required.

EQ3 has been evaluated qualitatively. As described, REACT using the CP-based feedback loop and Rainbow show different strengths and weaknesses. Hence, depending on the use case, a system developer has to choose one of them. A system developer who has to decide which approach to use has to consider specific requirements, such as the need for distributed deployments or weighted performance goals. As a system developer also has to apply the chosen approach,

the missing development process in Rainbow's case might be an obstacle when picking it up. Rainbow has its strengths in more in-depth analysis using its architecture model and a less complex planning phase as a result. Considering the timing aspect, not only the time to execute one run of a feedback loop is important, but also the frequency of adaptations. If the frequency is low and no quick decision is needed, a slower adaptation logic as in the case of Rainbow does not impose a drawback. REACT, however, offers runtime modifications of the adaptation behavior, decentralized control, and multi-language support. Accordingly, if there is the need for weighted optimization and a central deployment without too strict timing requirements, Rainbow is a good choice. If there is no need for weighted optimization and the requirement for decentralized deployments and fast execution, REACT is a good candidate. Determining these requirements allows the system developer to make the right choice.

By looking at the results shown in Figure 7.15, we observe that REACT using the CP-based REACT Loop can be applied effectively in a real-world communication system (*RQ4*). In addition, REACT makes it possible to efficiently change the behavior of the SDN controller by changing the adaptation options specification. It further enables porting the specified behavior to different SDN controllers by only implementing the effector interface and sending sensor data accordingly. Thus, we achieve portability of the specified behavior, which is not available in SDN in general, where each SDN controller needs specific SDN applications with different interfaces to the controller for applying a particular behavior in the network.

This finishes the presentation of the third and last ready-to-use REACT Loop as part of this thesis. Depending on the (modeling) requirements of a system developer targeting a communication system, one of the presented implementation has to be selected. In order to help in the decision by looking at the planning durations, in the following, the MILP- and CSP-based REACT Loops are compared directly against each other in a single use case. Additionally, the next section briefly examines potential improvements when combining multiple REACT Loops using different modeling approaches in parallel. For both objectives, a feasibility study is conducted.

7.4. Comparison and Combination of Feedback Loops

The previous sections presented different feedback loop instantiations called REACT Loops. This section presents the results of a feasibility study comparing and combining the MILP-based with the CP-based REACT Loop. The goal of this feasibility study is to answer the following evaluation questions:

EQ1–Comparison: What are the differences and similarities between the planners of the feedback loop instances considering speed, effectiveness, and expressiveness of the specification?

EQ2–Combination: How effective is the combination of multiple REACT Loops with different modeling capabilities and planning techniques?

For answering the two evaluation questions, we again chose the SWIM [216] use case, which we also used in Section 7.3. As solvers, SAT4J [223], IBM CPLEX[10], as well as Chocosolver [242] have been used. In this evaluation, the MILP-based REACT Loop using CardyGAn [229] presented in Section 7.2 is used in combination with SAT4J and CPLEX, The CP-based REACT Loop presented in the previous section is combined with Chocosolver again. The context module of REACT Core has not been used as part of this comparison as the objective of this evaluation is to measure the runtimes of the loops themselves. For the simulations a machine using Windows 10 in combination with an AMD Ryzen 3700X CPU with 16 GB of RAM has been used. SWIM has been executed using Docker with two CPU cores and 2 GB of memory.

In order to answer the question about planning speed posed in *EQ1*, a feature model, which can be solved by a SAT, MILP, and CSP solver is used. This first approach makes sure that the different possibilities in specifying a problem do not have an influence on the results. The CFM for this case is depicted in Figure 7.16. As shown, the response time is separated into multiple enumeration values. Additionally, there are system features with either one or two servers as parameters and the dimmer value can be changed using fixed steps. Again, servers induce costs, and the dimmer value represents the percentage of how many requests contain advertisements creating revenue and increasing the load on the servers. The specification enables to use this setting with all loop types.

[10] `http://www.ibm.com/analytics/cplex-optimizer`, accessed 2020-12-08

In the scenario, the 30 minute ClarkNet [250] trace provided with SWIM is used. Every run has been repeated 20 times, and the context of the system has been fetched every 10 seconds. We measured the average planning time, the standard deviation of the planning time, as well as the average utility. The utility is provided by SWIM. The embedded utility function takes the costs as well as the response time of the servers into account. Hence, it represents the quality of the reconfigurations. A C++-based layer implementing REACT's interfaces is used for communication between SWIM and REACT.

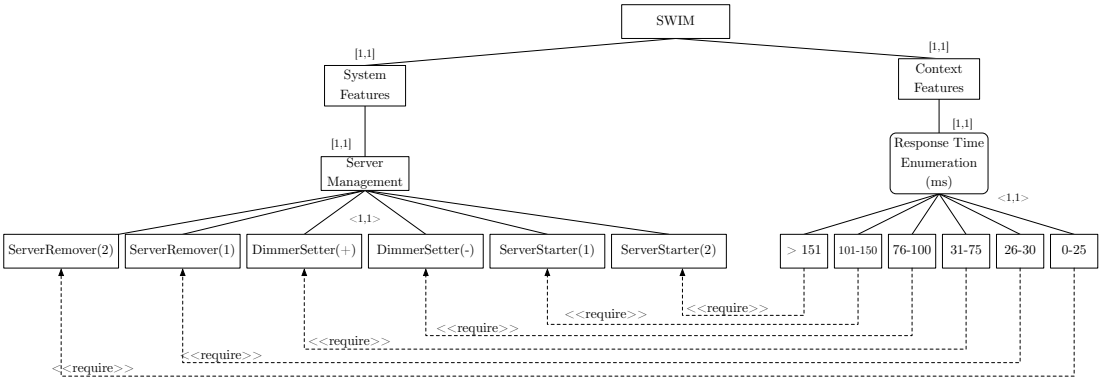


Figure 7.16.: CFM for the SWIM case used in the comparison for answering *EQ1*.

Table 7.5 shows the aggregated results of the evaluation runs. By looking at the planning time in the table, we observe that the planner using SAT4J needs the lowest amount of time. It is 10 % faster than the MILP-based planner and 73 % faster than the Chocosolver-based planner. Taking the utility values into account, the results show that they are close. However, the SAT4J-based runs have a slightly higher utility on average. The measured standard deviation of the utility values is 97.6. The utility difference is explainable with the faster adaptations, so the system reacts more promptly to changes in the load induced by the trace.

	Time average (total)	**Time St.Dev. (total)**	**Utility average (total)**
SAT4J	13.53	2.05	2505.30
CPLEX	15.00	4.77	2341.60
Chocosolver	49.45	3.57	2331.50

Table 7.5.: Total average planning time in ms and utility per run using simplified specifications. St.Dev.: Standard deviation.

Considering the expressiveness of the three solvers, this used CFM represents the lowest common denominator. Hence, the MILP- and CP-based solutions are not able to, e.g., plan attribute values directly. This leads to a larger CFM due to additional enumerations and duplicate system features differentiating only in terms of parameters.

For taking the different levels of expressiveness into account, in a second setup, each solver gets a specification using all solver features. In the SAT case, this does not change anything. However, the MILP- and CP-baser planners will be able to directly plan the absolute dimmer value and number of servers to start or to stop. In this case, factors approximating the revenue are used for particularly planning the dimmer value more precisely. The complete CardyGAn and Clafer specifications of the SAT problem, the simplified MILP, and Clafer counterparts, as well as the full MILP and Clafer specifications, can be found in Appendix B.

Table 7.6 shows the aggregated results of the different planners exploiting the expressiveness of the solvers. We can see that CPLEX needs considerably more time compared to the simplified specification used before. Looking at Chocosolver, it only needs about 5.5 ms more time using the full specification. For a better overview, Figure 7.17 shows a boxplot with all solvers, including the simplified runs of CPLEX and Chocosolver. The very first measurement in each run of the evaluations can be considered as warm-up phase and has been removed as outlier. The biggest changes can be seen in the average utility value. When using the full potential of the MILP- and CP-based solvers in terms of expressiveness, the utility increased considerably. CPLEX and Chocosolver perform similarly when comparing the utility.

	Time average (total)	**Time St.Dev. (total)**	**Utility average (total)**
SAT4J	13.53	2.05	2505.30
CPLEX	33.88	10.35	4059.50
Chocosolver	55.04	5.67	4009.10

Table 7.6.: Total average planning time in ms and utility per run using complex specifications. St.Dev.: Standard deviation.

Answering *EQ1*, we conclude that in the SWIM use case the specific planning of the attributes when using CPLEX or Chocosolver can increase the utility considerably. However, this is a tradeoff between planning time and utility. In cases where

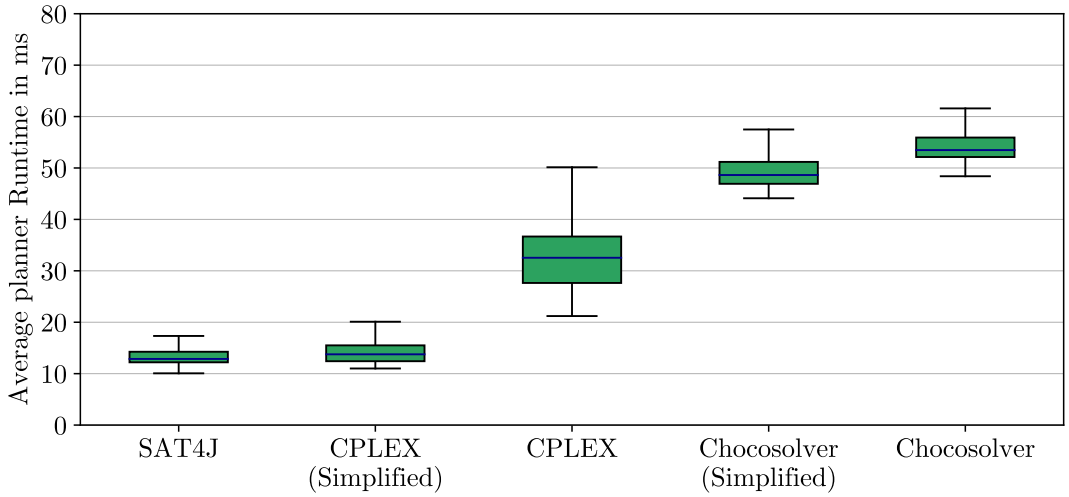


Figure 7.17.: Average planning times using different solvers and settings.

fast adaptations are needed, employing SAT4J or a SAT-based specification in combination with CPLEX is possibly the better choice.

With regard to *EQ2* and based on the results of *EQ1*, one idea is to combine multiple solvers. So, the approach is to use parallel feedback loops with the SAT solver directly adapting the system while the second feedback loops using MILP- or CP-based solvers are still running.

For answering *EQ2*, we use all three feedback loops in parallel with a coordinator component in front of and behind the loops. The coordinator starts the cycles at the same time and collects the results. Apart from the collection, it also handles conflicts between the different configurations. Hence, it implements a correction mechanism in case, e.g., the SAT solver had a different plan than the MILP-based feedback loop. In this feasibility study, the loops get assigned a priority value for handling conflicts. In this evaluation, the order is ascending from the SAT-based, over the MILP-based, to the CP-based loop. The same metrics as before are measured, including the number of corrections. As described in Section 2.1, it is possible to plan parametric or compositional adaptations. In the SWIM use case, changing the dimmer attribute in the feature model represents a parametric change, while changing the number of available servers is compositional. Hence, for each correction, it is logged, what kind of adaptation had to be corrected.

Figure 7.18 shows the measured results in the parallel execution of the feedback loops. We can see the number of corrections, with information about what type of correction had been executed and the number of adaptations without corrections. Additionally, cases where no adaptation was needed are shown. The figure reveals that 98 corrections have been compositional, while four where parametric. This shows that if the SAT solver is also planning the number of available servers, it often gets overruled by the MILP or CP solution. The average utility decreases from the value of 4059.50, which is the value when using the MILP-based feedback loop alone, to 2913.05 on average when using the parallel feedback loops.

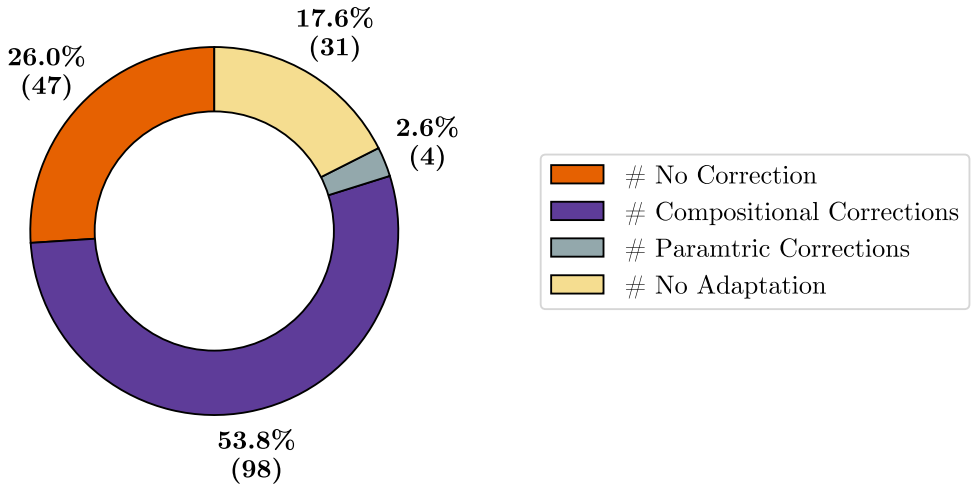


Figure 7.18.: Corrections when applying parallel feedback loops with priorities.

Consequently, as a second approach, the SAT solver only plans the dimmer value resulting in parametric adaptation, while server changes are planned by the MILP and CP solvers only. Accordingly, the CFM in Figure 7.16 only increases or decreases the dimmer value in response to the current response time. This setup should result in a lower number of total corrections. Figure 7.19 presents the measured number of corrections. As we can see, there are no compositional corrections anymore decreasing the overall number of corrections. However, as the SAT solver is only able to plan the dimmer in this setting, the number of parameter corrections increases accordingly. Additionally, when running in this combination, the overall average utility increases to 3907, which is 3.74 % lower compared to the MILP-based feedback loop. This hints at a problem with this combination. When two solvers are planning different parts of a system, conflicts

can easily occur. Solver 1 could increase the dimmer value and additionally start new servers to handle the additional load, while Solver 2 could do exactly the opposites. When taking the parametric increase of the dimmer value from the first solver and the compositional plan to decrease the number of servers, this possibly leads to bad results in the managed resource. Hence, this aspect is important to take into account when conducting future research.

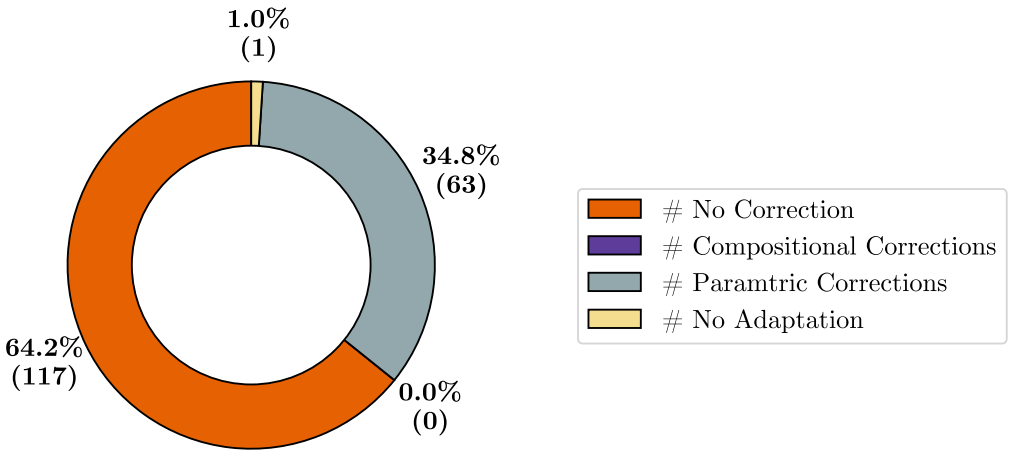


Figure 7.19.: Corrections when the SAT solver only plans the dimmer value.

Discussing these results considering *EQ2*, we do not see a performance gain when combining multiple feedback loops in the SWIM case. Still, the feasibility study shows the potential when combining multiple feedback loops with different planning capabilities together. The possibilities of taking these capabilities into account will be further discussed in Chapter 10, outlining future work.

8. Visualization of REACT

As described in Section 4.1, where the different stakeholders of a model-based runtime environment are outlined, administrators and system developers require a way to monitor and check the behavior of a SAS. For providing a corresponding solution, this section presents two visualization approaches in combination with REACT. Section 8.1 presents CoalaViz providing traceability capabilities, which can increase the understanding of the adaptation behavior at development time and runtime. This traceability can be used by system developers for debugging purposes during development as well as by administrators for monitoring the adaptation behavior. Based on this first approach, Section 8.2 presents EnTrace providing enhanced traceability incorporating influences from human computer interaction and explainable artificial intelligence. EnTrace improves CoalaViz in terms of architecture and capabilities. Both approaches close the loop when applying REACT after specification, implementation, and deployment for tracing the behavior of the SAS.

8.1. CoalaViz: Traceability of Adaptation Decisions

In order to provide a first solution for the problem of traceability, the section presents CoalaViz. This section is based on [251] and [252][1].

When applying REACT using one of the presented feedback loops, the resulting reconfiguration decisions cannot be traced back to the current system configuration, the contextual parameters, or the system performance, which all contribute to the reconfiguration decision. Hence, we propose CoalaViz, a novel tool for demonstrating the reconfiguration behavior of self-adaptive communication systems [251]. CoalaViz offers the following insights into the reconfiguration decisions: (i) The current system state can be investigated using a configurable graph-based

[1] [251] and [252] are joint works with M. Weckesser, R. Speith (né Kluge), J. Edinger, M. Luthra, R. Klose, C. Becker and A. Schürr.

network view (e.g., the overlay and underlay network of a distributed system). (ii) The entire configuration space and the currently active configuration of the system can be inspected using a feature-model view. (iii) The current system performance in terms of non-functional properties is shown as one or more metric plots. (iv) The priorities of the optionally available performance goals can be inspected and adjusted at runtime. CoalaViz is designed to be a standalone tool with clear technical interfaces for each of the described visualization components. These interfaces simplify the use of CoalaViz in different evaluation scenarios (e.g., as part of simulations and testbeds) and use cases. When looking at related works, already available tools are either tailored towards single specific use cases (e.g., [253–255]) or so generic (e.g., [256]) that they need a lot of work from potential users.

8.1.1. Use Cases and Challenges

Three pervasive communication systems are used to illustrate the challenges that lead to the architecture for the new system. These three systems consider the Tasklet distributed computing system [214], WSNs, as well as complex event processing (CEP).

An introduction to the Tasklet system has already been provided as part of Section 7.1.5. In short, the Tasklet system is a context-aware computational offloading middleware [214]. For the scheduling decision, the broker takes context information into account, for example, to avoid failures or to meet deadlines of tasks. To avoid failures, it selects a scheduling algorithm that most accurately predicts the availability of the providers, which enter and leave the system dynamically [224]. Accordingly, adaptively changing the scheduling algorithm can help to improve the system performance.

WSNs and topology control have also been already briefly introduced as part of Section 7.2.5. In short, a wireless sensor network (WSN) consists of dozens to hundreds of cheap, battery-powered, resource-constrained sensor devices (called *motes*) that collectively serve a particular purpose (e.g., environmental monitoring) [257]. A modern mote provides numerous configuration options to adjust the WSN to the current system context (e.g., mobility pattern and robustness requirements). *Topology control* is a technique to address non-functional system

goals (e.g., the energy consumption) of a WSN by thinning out the number of visible neighbors on the link layer. A topology control algorithm presents the resulting *virtual topology*, which is a subgraph view of the physical neighborhood, to the network layer. The sparsity of the virtual topology comes at the cost of decreased robustness and/or higher latency. Each of the numerous topology control algorithms that have been proposed in the literature provides a different trade-off between energy consumption, robustness, and latency. In the context of IoT, WSNs are used in safety- and security-critical scenarios (e.g., e-health, intrusion detection). Therefore, reconfiguring the topology control algorithm of a mote periodically is required to meet the safety, security, and performance requirements [215].

Complex Event Processing (CEP) deals with processing continuous streams of data from devices (*producers*) to derive meaningful events for the end-users (*consumers*). *Complex events* are highly relevant for applications in the context of IoT (e.g., weather monitoring using WSNs). A complex event can be expressed as continuous query that is registered with the CEP engine. The CEP query is composed of *logical* units called *operators*. The CEP system processes the query in a distributed manner by placing the operators on the devices in the network (e.g., *motes* in a WSN). *Operator placement* is a mechanism that places operators based on the non-functional requirements posed by the consumers. Therefore, a CEP system exposes an underlay view, consisting of connected consumers, producers, and brokers, and an overlay placement view, better known as an *operator graph*. The operator placement should be such that it fulfills the non-functional requirements of the *consumers*. However, the *consumers* may have distinct and conflicting non-functional requirements depending on the current environmental conditions (context), e.g., when the operators are placed on mobile devices or cloud resources. In particular, one non-functional requirement of IoT applications is to deliver complex events in minimum *response time*, but also at a low cost in terms of overhead for mobile devices (e.g., measured as the number of messages exchanged). These distinct conflicting requirements are hard to be fulfilled using *one* operator placement mechanism, but requires a runtime reconfiguration of the CEP system with *multiple* operator placement mechanisms.

Resulting from the three described use cases, three challenges are identified for CoalaViz. These challenges also fit the requirements of the stakeholders as well as the description of the different specific requirements in Chapter 4.

C1 Traceability: How does the adaptation logic come up with an adaptation decision based on the current system state? With the assumption that adaptation decisions depend on the system context, the current system context needs to be presented in an understandable way. Additionally, for tracing an adaptation from one state to another, the current system state shall be visualized. As the state of a communication system can typically be represented by a graph, effects like entity churn or node movements can comprehensibly be visualized. In doing so, nodes can form both an overlay and an underlay network, resulting in different changing topologies, respectively. Besides the context and the system state, also non-functional requirements, such as execution speed, fairness or response time, may change during runtime. Hence, non-functional performance metrics shall be traceable as well. Additionally, as the goals of a system might change at runtime, CoalaViz must show which goals are pursued at any point in time. Challenge C1 maps to functional requirement R_F8 (runtime monitoring and modifications).

C2 Extensibility: While the three presented use cases address important and well-known challenges, they may still be considered just a problem subset in the field of adaptive communication systems. Hence, an important challenge is to develop a tool that exposes clear extension points for supporting new use cases. In doing so, we also take into account that each use case may well exhibit its individual non-functional system goals. This ensures that different scenarios and simulation tools can be combined with it. Challenge C2 also maps to requirement $R_{NF}6$ for providing a solution with high extensibility.

C3 Responsiveness: CoalaViz shall address the pictured system's dynamics by providing a decent level of responsiveness. We note that communication systems have several degrees of dynamics at different levels of the communication protocol stack, e.g., movement or changing communication connections. Nevertheless, the performance metrics shall be monitored continuously and reliably and with low delay on all levels. The high responsiveness shall assist system developers in maintaining a clear and comprehensible picture of the overall system and its adaptation decisions. Challenge C3 also maps to requirement $R_{NF}3$ for providing a solution with a high overall performance.

8.1.2. Design and Implementation

Figure 8.1 illustrates the architecture of CoalaViz, which consists of its backend and frontend. The connection between the REACT-based SAS, which should be monitored, as well as CoalaViz are depicted in Figure 8.2. Backend components process the events that originate from the SAS and arrive via one or more event streams (A), and notify the corresponding frontend components ((B)...(E)). The major event types are shown as broad black arrows in Figure 8.2. All information that is visualized in the frontend components originates from these received events. This completely event-based workflow enables CoalaViz to replay events even in the absence of a running system. A JSON- and socket-based interface allows CoalaViz to receive events independently of the programming language and type of managed resource (e.g., simulators or actual devices). Each frontend component can be exchanged individually, e.g., to show a logical expression instead of the graphical feature modeling view. Figure 8.1 indicates that the backend consists of components for each view that appropriately translate events for the actual frontend implementation. The frontend forms a dashboard and consists of a graph-based network view (B), a metric view showing the reported non-functional property values of the system (C), a combined CFM and configuration view (B), and the performance goals control panel, which shows the weighted performance goals and enables to update the weights interactively (E). CoalaViz interacts with the managed resource and the feedback loop of the SAS. To establish compatibility with CoalaViz, we exemplary extended the MILP-based feedback loop to emit events about (i) available performance goals with default weights during initialization for the panel (E), (ii) the CFM of the adaptation logic for the view (D), and (iii) new system configurations whenever the planner produces a new reconfiguration decision, which are also visualized by (D). Conversely, CoalaViz informs the adaptation logic, when the user modifies the performance goal weights.

The managed resource (e.g., a simulation in OMNeT++) sends events about (i) modifications of the network state, such as node or edge additions, removals, or property modifications, which are visualized by (B), and (ii) new metric values as triple of metric name, timestamp, and metric value, as visualized by (C). The

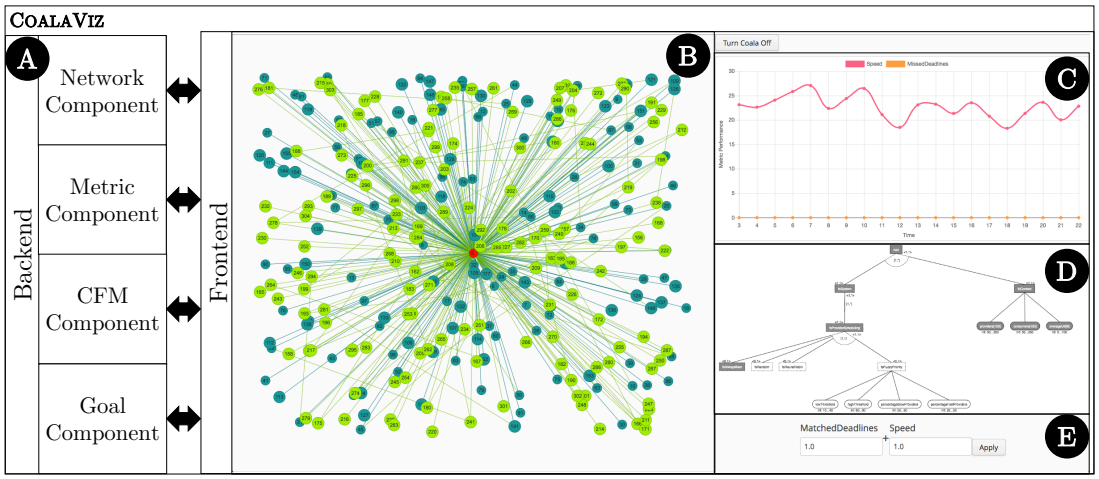


Figure 8.1.: Architecture of CoalaViz [251, 252].

network view interprets optional node and edge properties as rendering hints (e.g., node fill and border color, edge color and stroke, textual node, and edge labels).

Network Component: The network component processes the network model change events from the managed resource. It supports events for adding and modifying nodes and edges that represent the network state. The network frontend view (B) shows a graph that represents the current state of the network. The position of each node is communicated by the managed resource. CoalaViz maintains the graph structure based on the event stream and visualizes it in the network view. A user of CoalaViz can set the colors of nodes and edges or the thickness of edges for showing different weights. For example, this provides the capability that the graph can represent overlay or underlay networks. The color of a node can represent the different device or connection types.

Metric Component: The metric component processes events with new metric values. Each such event provides the metric name, a numeric value, and a timestamp of the data point. The *metric view* (C) shows the evolution of one or more metric values in a combined x-y-plot. The x-axis shows the time (according to the timestamp values), and the y-axis shows the value per metric.

CFM Component: The CFM component receives events about the model and the context or system configurations and notifies the *CFM view* (D) accordingly. The model is typically received only once when the system starts. At runtime, the CFM component gets the system context from the managed resource and the

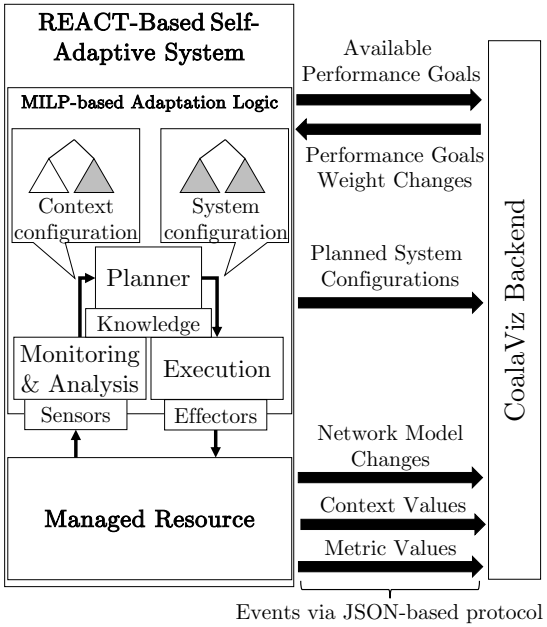


Figure 8.2.: Connection of a REACT instance using the MILP-based REACT Loop to CoalaViz [251, 252].

planned system configurations from the AL. The CFM view shows the configuration options of the system as attributed feature diagram together with the currently selected features and their attribute values. The view provides an aggregated, centralized perspective of the system compared to the detailed network view. This allows monitoring the reconfiguration decisions of the adaptation logic according to context changes.

Goal Component: The goal component receives the optionally available non-functional performance goals from the adaptation logic and sends events about changed weights back to the AL. The corresponding view component is the *performance goal control panel* (E). It shows the available performance goals with weights for each goal. In combination with the network, metric, and CFM views, the goal view allows assessing how well and how quickly the adaptation of the SAS meets a defined system goal. The user may also adjust the weights of performance goals at runtime to explore how the SAS reacts. Finally, this component allows exploring the reconfiguration behavior of an adaptation logic and the resulting system states under changing performance goals.

8.1.3. Evaluation

This section evaluates CoalaViz concerning the three identified challenges Traceability (C1), Extensibility (C2), and Responsiveness (C3). Traceability and responsiveness are evaluated qualitatively given the capabilities and architecture of CoalaViz. Responsiveness is evaluated quantitatively by running CoalaViz with a JSON event stream on a laptop with an i5-5257U CPU and 8GB of memory.

C1 Traceability

C1 is concerned with the traceability of different aspects of the inspected self-adaptive communication system. This includes the system state of the network, metrics, the reconfiguration space, including the current configuration, and the performance goals. The system state of the network can be viewed using the network component and the connected network view (B). It shows nodes and edges and can be styled to show different node or edge types. Thus, a changing network topology can be tracked visually. The metric view shows the current value of multiple non-functional metrics as well as the history of each value (C). Concerning the context and the system configuration, CoalaViz is able to show CFMs, including the current system configuration in its CFM view (D). The performance goal panel shows the non-functional system goals and their weights and enables to change these weights at runtime (E). This allows the assessment of the influence of the adjusted goal on the adaptation decisions.

In summary, we qualitatively evaluated the traceability of CoalaViz for the three use cases presented earlier and found that (A), (B), (C), (D) and (E) contribute in analyzing the system and context configuration, especially since adaptations affect the performance of the system tremendously. For instance, using CoalaViz, we can monitor how adaptations influence throughput and deadline misses in Tasklet, energy consumption in WSNs and response time in CEP. Summarizing, the implemented views of CoalaViz provide continuous insights into different aspects of a self-adaptive communication system and make adaptation decisions traceable.

C2 Extensibility

The goal of the second challenge was to make sure that our solution is easily extensible and changeable. CoalaViz is a web application using the Vaadin

framework[2]. This allowed us to implement the backend code in Java, while for the frontend, standard JavaScript libraries could be used. The JSON-based protocol makes sure that CoalaViz can easily be made compatible with different adaptation logics and managed resources. Accordingly, CoalaViz could be applied with all three use cases although the Tasklet system is simulated using OMNeT++ [246], the WSN scenario uses Simonstrator [233], and the CEP case uses a custom implementation [258]. The view components, in particular, are abstractions in the backend for using different views in the frontend. These components translate incoming events by calling corresponding JavaScript methods of the frontend. For the network view, we use VisJS, while the metric view is implemented using ChartJS[2]. The CFM view is a customized implementation, while the goal view consists of standard UI elements of Vaadin. Due to the modular design, views can easily be exchanged.

C3 Responsiveness

For measuring the responsiveness, we log the timestamps right after the socket on the sending side is flushed and when the JavaScript code for changing a view was executed. As a first step, the responsiveness of the network, metric, and CFM views are evaluated separately with artificial data. In all three cases, we simulate JSON requests with events for 5 minutes of real-time. For evaluating the network view, we use a Poisson distribution with an average arrival time of 1 event per second. The Poisson distribution is a commonly used model to describe inter-arrival times of incoming or departing data entities. By that, we add nodes and edges randomly to the system. In the case of the metric and CFM view, we send one event per second, as metrics and context changes typically happen on a more regular basis. Finally, we send two metric values at once each second and a random configuration / context, respectively. The results of the three evaluation runs are depicted in Figure 8.3.

The figure shows that the network and metric views, which are implemented using standard open source components, perform better than the custom-built CFM view. In fact, the CFM view is rendered as an image, which makes it slower compared to the JavaScript views. Additionally, the whole image is (re-) rendered

[2] `https://vaadin.com/`, `http://visjs.org/`, `https://www.chartjs.org/`, accessed 2020-12-08

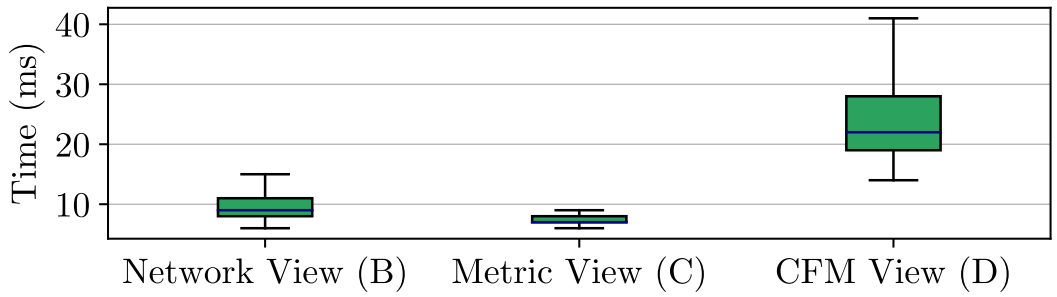


Figure 8.3.: Responsiveness per view with artificial data [251].

even for small changes. As this is a low load scenario, this shows the best possible performance for each view.

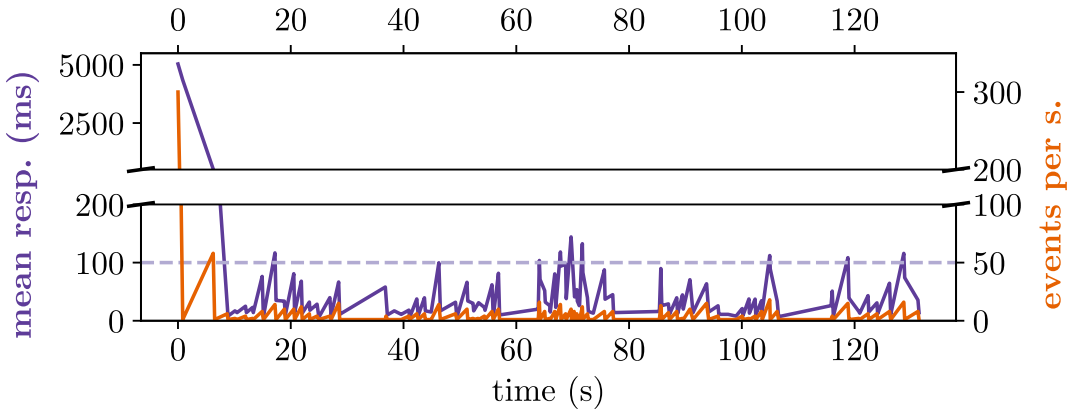


Figure 8.4.: Mean of the responsiveness over time in WSN case [251].

Figure 8.4 shows the playback of one hour simulated time in the Simonstrator in the WSN case. Here, all different events in the three views are measured. The stream resulted in 2.5 minutes of runtime in CoalaViz. We observe that the first 302 events have a high latency as in this period, initial nodes and edges are added to the system. After this warmup phase, we end up with a median of 55 ms for the responsiveness. In UI research, 100 ms are considered as a limit for an instant reaction [259]. The dashed line in the figure indicates this value. Most reactions in the UI can be considered responsive. Thus, we consider CoalaViz to be sufficiently responsive to provide traceability given this condensed event stream and the hardware the evaluation was executed on.

CoalaViz provides a modular, web-based, and reusable visualization platform that enables to trace the reconfiguration behavior of self-adaptive communication systems while interactively adjusting the system's optimization goals. In combination with REACT it helps system developers to check the specified behavior for debugging purposes. Additionally, administrators can use CoalaViz to monitor and influence the deployed self-adaptive communication system at runtime. CoalaViz has been evaluated in combination with the MILP-based REACT Loop presented in Section 7.2. However, CoalaViz is also generic for simplifying the integration of other REACT Loops.

8.2. EnTrace: Enhanced Traceability of Adaptation Decisions

By using CoalaViz, as described in the previous section, we gain first experiences with traceability of self-adaptive systems. One shortcoming of CoalaViz is the missing integration of foundations from artificial intelligence research, as understanding and tracing the behavior of machine learning techniques is a major research focus named explainable artificial intelligence. Additionally, we identified limitations of the Vaadin-based implementation, resulting in problematic performance in larger settings as well as complex customization options of the dashboard items themselves. Accordingly, we build upon the first results, draw inspiration from artificial intelligence research, as well as aim at using a better-performing and customizable implementation.

The challenge of understanding black-box algorithms is a focus in many works referring to the mentioned term explainable artificial intelligence (XAI) [260]. Further, several approaches for visualizing machine learning algorithms with the goal of making them interactive and explorable (e.g., in [261]) exist. We coin the term *enhanced traceability* in self-adaptive systems as an understanding of traceability that is inspired by knowledge from XAI, data visualization, and human-computer interaction. This section is based on [262][3].

In the following, we propose EnTrace—a reusable and open source platform that provides enhanced traceability capabilities for self-adaptive communication

[3] [262] is joint work with M. Breitbach, and C. Becker.

systems. EnTrace is an interactive tool that visualizes the current state of a self-adaptive communication system, including its network topology and CFM specification as well as the progression of non-functional goals over time. EnTrace offers a customizable dashboard with multiple views. We design EnTrace to be easy to use by developers and administrators without extensive knowledge in SAS development. Therefore, we ensure that developers and administrators can easily connect EnTrace to an existing system and customize the visualization according to their preferences. Since many modern adaptive systems consist of distributed components and leverage decentralized control [15], EnTrace is able to show monitoring data from multiple, distributed hosts. In addition, EnTrace provides a seamless user experience with high responsiveness. We offer EnTrace as an open source project[4] and may thus contribute to the application of adaptive systems in practice, which is still considered a major challenge [110]. In a quantitative and qualitative evaluation, we show EnTrace's benefits and compare it to CoalaViz presented in the previous section.

8.2.1. Definition of Enhanced Traceability

We define *enhanced traceability* as techniques, methods, and concepts used to *visualize* the states and decisions of a system, as well as making them *explainable* in an *interactive* format. In contrast to software traceability [263], which is concerned with, e.g., traceability of requirements and test cases, we rather focus on enhanced traceability from an XAI perspective. The overall goal of a tool providing enhanced traceability is to allow system designers and administrators to make informed decisions about its development, deployment, and use by offering transparency about its inner behavior. In the following, this section briefly introduces the foundational concepts of XAI, data visualization, and human-computer interaction that shape our understanding of enhanced traceability.

XAI makes black boxes transparent resulting in white boxes [260]. It is a combination of several topics, including transparency, causality, bias, fairness, and safety [260]. By approaching XAI, interpretability should be achieved, which is "*the ability to explain or to present in understandable terms to a human*" [264]. The authors state that interpretability is required in systems with incompleteness

[4]Available here: `https://github.com/martinpfannemueller/EnTrace`

in domains such as scientific understanding, safety, and ethics. In these domains, it is difficult to formalize problems resulting in the need for a human in the loop. Hence, interpretability helps to understand reasoning processes in incomplete problem spaces [264].

Data visualization is an important part of enhanced traceability. Here, especially the concept of information overload must be taken into account [265]. The broad types of visualizations—tables and graphs—should be chosen according to the task. Tables show data in a symbolic way with precise values, while graphs are better suited for spatial information [266]. In addition, visualizations should (i) avoid clutter [267], (ii) tell stories if possible [268], and (iii) be easily understandable and memorable [269].

Lastly, by combining data visualization techniques and foundations from human-computer interaction, the human should be supported with possibilities of interactivity [270]. This results in concepts such as zooming, filtering, details-on-demand, relate, history, and extract for interactively exploring data [271]. Human-computer interaction research also shows that (simple) animations can help to improve graphical perception [272]. Dashboards are a common way to present different kinds of data simultaneously [273]. Dashboards can be categorized into *visual* dashboard elements showing static displays of information and *functional* dashboard elements, including interactivity.

Looking at related works, the previous approach CoalaViz has several shortcomings. First, it is difficult to apply to self-adaptive systems that do not use the REACT- and MILP-based adaptation logic. Second, it does not support developers optimally since it has not been publicly available and requires the implementation of a central data collection in the managed resource, which may be cumbersome in distributed self-adaptive systems. Third, it has only limited usability due to, e.g., low responsiveness under high load.

In addition to our experiences with *CoalaViz*, we draw inspiration for the design of EnTrace from artificial intelligence research, where explainability and interpretability are a key focus. Kahng *et al.* introduce *ActiVis*—"*an interactive visualization system for interpreting large-scale deep learning models*" [261]. It provides multiple views displaying various graphs and other visualizations in a dashboard. Users are able to interactively zoom the visualized elements and re-

trieve additional information from hovers. *DeepVID* is a "*Deep learning approach to Visually Interpret and Diagnose DNN [deep neural network] models*" developed by Wang *et al.* [274]. It is applied to visual image classifiers enabling users to interactively navigate through the feature space of an image classifier. Again, it follows a dashboard approach showing different graphs and visualizations. There are multiple other approaches such as *GAN Lab* [275] for exploring Generative Adversarial Networks in the field of deep learning, *RetainVis* [276], *ReVACNN* [277], and the *TensorFlow Graph Visualizer* [278] related to general neural networks and *Seq2Seq2-Vis* [279] visualizing neural sequence-to-sequence models.

All works except for *RetainVis* are web-based implementations using JavaScript, while the *GAN Lab* authors explicitly mention the cross-platform aspect of using JavaScript for development. Interactivity is the main focus in all these related works allowing one to zoom in and explore graphs and visualizations in detail. Except for *GAN Lab* and *ReVACNN*, hovers have been used to show additional information to the user. All approaches follow a dashboard-style user interface with multiple views showing graphical representations of major parts of the underlying system.

Based on the observations so far, we propose EnTrace—a reusable platform for enhanced traceability in self-adaptive communication systems. First, the following describes the challenges that shape EnTrace's features. Second, we present EnTrace's design. Third, we briefly give implementation details about the publicly available tool.

8.2.2. Challenges

Compared to CoalaViz, the first challenge of EnTrace is to increase the monitoring/traceability possibilities. Second, as CoalaViz uses a single socket connection to the managed resource, which is problematic in the case of distributed deployments, EnTrace should support distributed and decentralized systems. Third, as CoalaViz showed an improvable performance in some cases, EnTrace focuses on a higher performance for monitoring even larger systems. Accordingly, EnTrace's design is shaped by the three challenges (i) *enhanced traceability*, (ii) *support for distributed and decentralized systems*, and (iii) *responsiveness*.

Enhanced traceability extends the general idea of traceability with *explainability*, *visualization*, and *interactivity* as introduced in the previous section. Explainability ensures that adaptations are visible, traceable, understandable, and ultimately explainable. Hence, this leads to trustworthiness and transparency. EnTrace should help humans to easily understand explanations [260]. Consequently, the overall complexity should be as low as possible, and only elements increasing explainability should be incorporated. Second, the ideas from visualization show that data must be presented in an understandable form. As the visualization of software and algorithms is one of the most complex categories to visualize [280], a clean interface without unnecessary information is required [267]. Hence, the interface should be human-centered [270]. The representation of the data should be memorable facilitating understanding [268, 269]. Last, as literature shows, interactivity is needed in addition to visualization concepts. Accordingly, EnTrace should provide "*overview first, zoom and filter, then details on demand*" [271]. Additionally, animations and a dashboard approach support interactivity [267,272].

EnTrace should offer **support for distributed and decentralized systems**. These systems typically follow distribution patterns, as described in [15]. Accordingly, EnTrace should be able to receive information from different parts of a system. Without support for distributed data collection, the system developer faces the additional obstacle to collect all status information within the SAS at one point, put it together in messages, and manually send it via a socket as in CoalaViz [251]. Ideally, EnTrace should, therefore, make it possible to easily connect to different system parts and collect status information from various sources.

Responsiveness is a challenge, as only a responsive tool can trace system dynamics correctly in time. As soon as something happens in the monitored system, EnTrace should show the corresponding change. Conceptually, for achieving responsiveness, the implemented solution should focus on high performance and low overhead. EnTrace should be considerably more responsive than CoalaViz, as also systems with large network topologies and various metrics should be supported, which led to a bad user experience during peak times in CoalaViz.

8.2.3. System Design

EnTrace offers a dashboard to provide enhanced traceability to system designers, developers, and administrators who want to verify and observe the behavior of a SAS. EnTrace's dashboard is usable in the production and the operations phase of the software development process presented in Section 4.1. It has a customizable two-column grid layout with multiple views. Thus, Users can create a visual representation according to their preferences. Figure 8.5 shows a screenshot of EnTrace's dashboard with the same WSN use case we used in CoalaViz. Interactivity is realized by incorporating zoom, filter, and drill-down capabilities. All dashboard elements provide hovers to provide additional information on demand. EnTrace's views do not only visualize the managed resource's state but also enable users to perform changes in the system configuration or the system goals and directly observe the impact of these changes. Hence, the user is also able to interact with the SAS instead of just observing it.

The dashboard contains a *network view*, a *metric view*, a *configuration view*, a *performance view*, a *state view*, and an *event view*[5]. The *network view* shows the network topology of the self-adaptive communication system using nodes and edges. Users of EnTrace are able to customize the view to their needs. Nodes and edges can be configured with different colors and additional properties such as weights. In the exemplary WSN use case in Figure 8.5, the network view displays the network topology of the WSN. The user selected a special color for the sink node of the WSN to better distinguish it from the sensor nodes. The *metric view* plots multiple, configurable non-functional goals of the managed resource, such as latency or fairness over time. The *configuration view* depicts the CFM of the managed resource—if available—including the current configuration and attribute values representing the system state. In addition to Boolean features, it is also possible to show available attributes for representing parameter or context values. The CFM view enables users to set an attribute in the system or "freeze" a feature so that it remains unchanged by adaptations. Users can thus explore the behavior of the adaptive system in specific scenarios directly via the dashboard with this interactive feature. The *performance view* makes it possible to weight non-functional goals for influencing the adaptation decisions. This allows putting

[5]Network, metric, configuration, and performance views were already part of CoalaViz [251], but in a less customizable and interactive form.

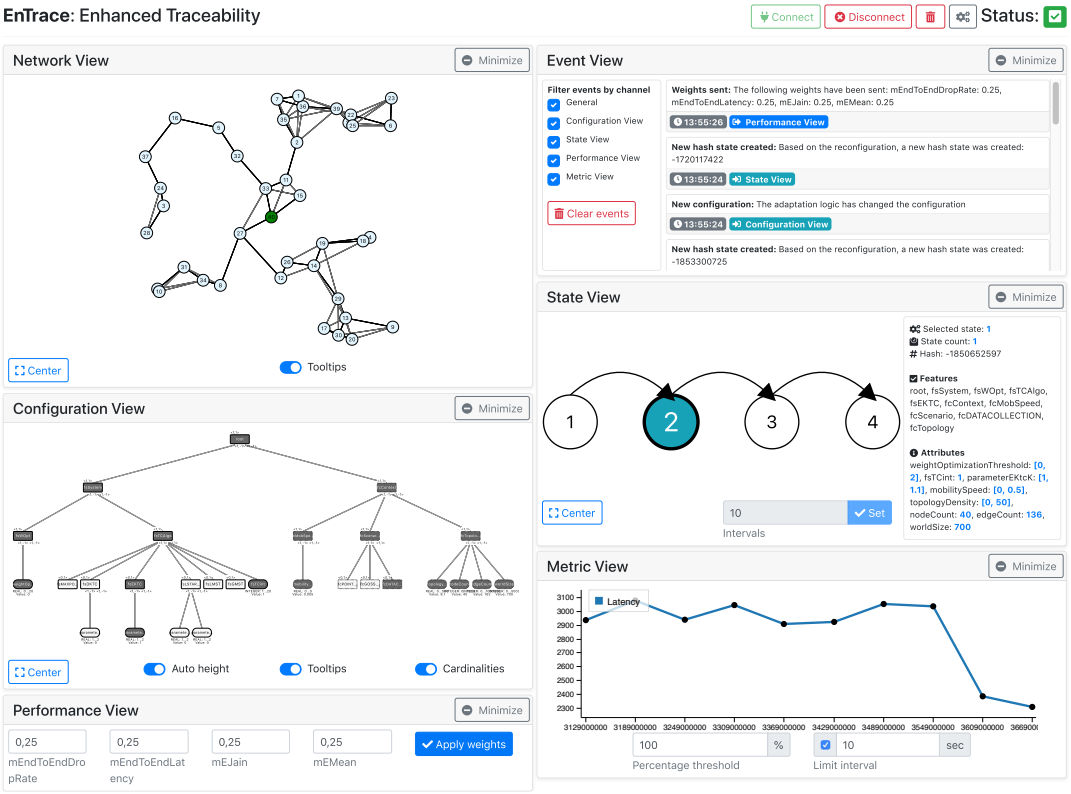


Figure 8.5.: Screenshot of the EnTrace dashboard in a WSN use case. The dashboard contains views that illustrate the network topology of the managed resource, the configuration and context of the managed resource, the progression of non-functional goals over time, and the transitions between different system states. Additionally, EnTrace enables to set non-functional goals via the dashboard and shows an event history [262].

the human in the loop and observing the influence of goal changes. EnTrace contains a *state view* which automatically illustrates system states and transitions between these states based on the data from the configuration view. With this view, the user is able to detect loop behavior in the adaptation decisions as the state view counts how often states are visited and transitions are traversed. It highlights transitions that occur more frequently. To make this view usable with continuous attributes in the CFM, EnTrace applies configurable discretization. In the WSN use case in Figure 8.5, the state view currently highlights state 2, which was selected by the user to get detailed information about the system and context configuration in this state. The *event view* summarizes events of all other views,

errors, and also important changes in the managed resource. The user is able to define a threshold that determines which change, i.e., in a metric or attribute value, triggers such an event that is displayed in the event view.

EnTrace connects to a SAS via MQTT, as depicted in Figure 8.6. This decouples EnTrace from the SAS and enables to easily collect status information from different, distributed (sub-) parts of the system. Additionally, several instances of EnTrace may connect to the same broker to use multiple displays with customized view setups at the same time. The SAS sends status updates while EnTrace sends human-in-the-loop actions—such as goal changes—via the MQTT connection.

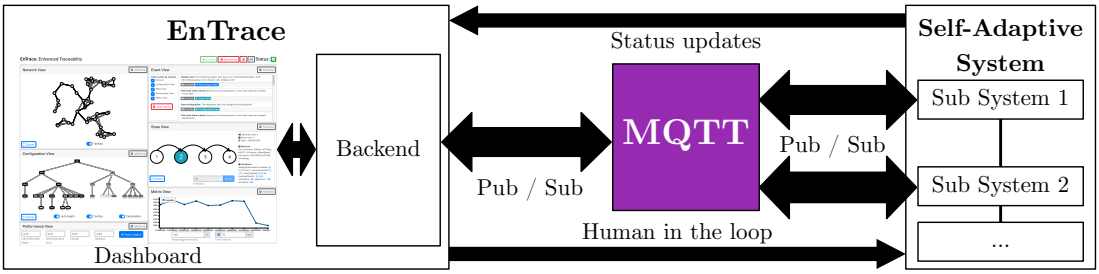


Figure 8.6.: Architecture of EnTrace. It consists of a dashboard and a backend connected to the SAS via MQTT. EnTrace receives status updates and publishes human-in-the-loop actions [262].

8.2.4. Implementation

EnTrace is implemented as a stand-alone JavaScript web application. It features a dashboard-style user interface with six views, which can be freely arranged. Additionally, each view can be closed if it is not needed by the user. The JavaScript web application is implemented with the Vue.js framework using Bootstrap for the UI[6]. The public repository contains documentation on how to start EnTrace as well as about all available event types. The provided quick start guide enables to easily start EnTrace, a JavaScript-based broker, and a replay of the WSN events used as part of the evaluation. This facilitates to evaluate EnTrace without setting up a self-adaptive communication system.

[6]see `https://vuejs.org/` and `https://getbootstrap.com/`, accessed 2020-12-08

8.2.5. Evaluation

First, we qualitatively evaluate whether EnTrace provides enhanced traceability in SAS. Second, we qualitatively evaluate whether EnTrace supports distributed and decentralized adaptive systems. Third, we quantitatively evaluate the responsiveness in comparison to CoalaViz.

Enhanced Traceability

We qualitatively evaluate whether EnTrace achieves enhanced traceability by assessing *explainability*, *visualization*, and *interactivity*.

Explainability: EnTrace helps the user to monitor and trace the system state in different aspects. The different nodes and edges of a distributed system are shown using the network view, while the configuration and environment state is shown in the configuration view. EnTrace also shows changes in the non-functional goals and if, e.g., the system is oscillating between two states. The event view helps to provide a storyline showing critical events and changes in the system. In general, EnTrace provides *transparency* [281], helps in understanding *causality* [260], and fosters *trust* [264] in the underlying SAS. We thus conclude that EnTrace achieves explainability.

Visualization: All views are vector-based and support different screen resolutions. Animations are used if helpful (e.g., for showing changed values in the metric view). The presentation is as clean as possible, only showing the relevant data and options. Additional data is shown on demand by using tooltips, e.g., for showing weights of edges in the network view. In the current approach, no tables are used for showing the behavior of the system. This supports the dynamics of the adaptive systems and is easier to follow by the user. Hence, we conclude that EnTrace's visualizes the information appropriately.

Interactivity: EnTrace makes considerable use of all the interactivity techniques proposed by Shneiderman [271]. Mostly, tooltips and hovers are used for showing "details-on-demand" [271]. Filtering possibilities and options for, e.g., disabling tooltips can be used for reducing clutter [267]. Automatic zoom when drilling down or collapsing the CFM fits to the *coupled zooming and drilling* [282] methodology. Therefore, we conclude that EnTrace provides a suitable level of interactivity.

Support for Distributed and Decentralized Systems

EnTrace decouples the connection to the SAS with MQTT for supporting distributed and decentralized systems as proposed in [15]. MQTT is a widely-used publish-subscribe standard, which is especially attractive for the IoT due to its low overhead [283]. This property is used in EnTrace to publish events from different parts of a SAS, even if the resources of these parts are limited. It further allows executing multiple instances of EnTrace on multiple systems and displays simultaneously without additional overhead. By using MQTT, we, therefore, declare that EnTrace supports distributed and decentralized systems as desired.

Responsiveness

We perform two experiments to investigate EnTrace's responsiveness. First, we use artificial system events to assess the scalability of the different views on the dashboard. Hence, we measure the responsiveness of each view separately. Second, we use a WSN replay for comparing the overall responsiveness of EnTrace to the responsiveness of CoalaViz. In both experiments, a response time of below 100 ms is considered as responsive to the human as defined in [259]. The responsiveness is measured from the time an event is received via MQTT until it is displayed in the browser. Hence, the network transmission is omitted here. In general, however, MQTT is used in many IoT scenarios with low latencies [283]. Each individual run was executed 30 times on an i7-3615QM CPU.

In the first experiment with artificial events, we study the responsiveness of the network, metric, configuration, and state views separately. We do not assess performance view and event view since these views do not respond to incoming status information from the SAS but provide an input mask for human-in-the-loop actions and process internal events of EnTrace, respectively. A Poisson distribution with an average arrival time of 1 per second is used for node and edge events in the network view. The metric view was tested with two metrics and random values between 0 and 100 each second. For updating the configuration view and the state view, random system and context configuration events are sent every second. One single run consists of 300 seconds.

As shown in Figure 8.7, the mean response time of each view is low. We observe that the response times of the network view and state view increase over time as it gets gradually more complex to update the rendering with an increasing

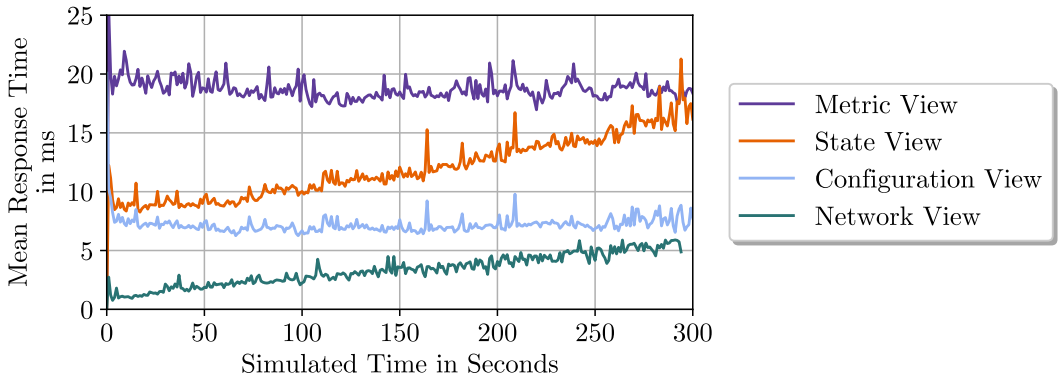


Figure 8.7.: Mean response times of the dashboard views running separately with artificial events. All values are well below 100 ms, which demonstrates the scalability of EnTrace's dashboard [262].

number of nodes and edges. Metric view and configuration view stay stable, as they are only updated and not extended with new elements over time. Taking Table 8.1 into account, even the maximum value of the state view with around 78 ms is well below 100 ms. This demonstrates the scalability of the state view since—due to the random configuration view values—the state space gets large in this experiment.

View	Mean (ms)	Min. (ms)	Max. (ms)	SD (ms)
Network View	3.34	0.46	15.74	1.43
Metric View	18.72	13.16	61.99	2.37
Configuration View	7.21	4.56	49.94	2.00
State View	11.98	5.78	77.79	3.64
	12.02	**0.46**	**77.79**	**6.60**

Table 8.1.: Summary of response time measurements of EnTrace's dashboard views under artificial load, SD: standard deviation [262].

Second, we compare EnTrace to CoalaViz by replaying the WSN trace via MQTT for measuring the responsiveness. Figure 8.8 shows the comparison of the response time of the dashboard with all views. We observe that EnTrace is much more stable with response times well below 100 ms and mostly below 50 ms. In comparison, the average responsiveness of CoalaViz is more unstable. Occasionally, CoalaViz' response times exceed 100 ms, which leads to a worse user experience. Hence, we conclude that EnTrace achieves high responsiveness and constitutes a considerable improvement to CoalaViz in this metric.

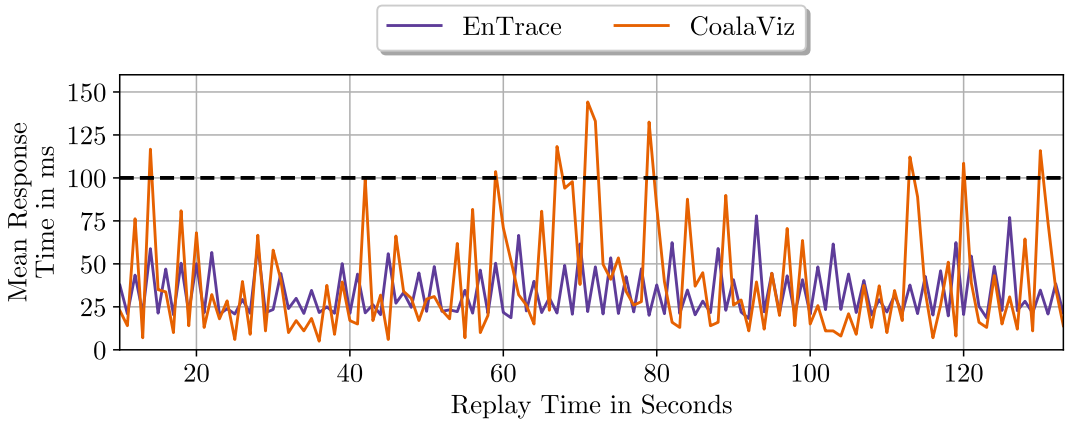


Figure 8.8.: Response times of EnTrace's dashboard in comparison to CoalaViz in a WSN use case. EnTrace is more responsive on average and avoids peaks of response times that are higher than 100 ms, which ensures smooth user experience. Please note: The x-axis starts at $x = 10$ to omit setup time [262].

This section presented EnTrace—a publicly available tool for providing enhanced traceability capabilities for SASs in a responsive way. EnTrace offers a customizable dashboard that provides an overview of a SAS to developers and administrators. Also, it supports distributed and decentralized managed resources in a scalable way with MQTT. The system helps developers and administrators to monitor the adaptation behavior at design and runtime.

9. Discussion

The evaluation of the feedback loop instantiations called REACT Loops presented in Chapter 7 showed the broad applicability of REACT in different use cases ranging from simulations to emulations of real systems. We were able to show that the use case-specific requirements were fulfilled resulting in systems adapting successfully to changes in the execution environment. Also, compared to the state-of-the-art approach Rainbow [17], applying REACT in combination with the CP-based REACT Loop considerably decreases the development effort for enhancing a system with adaptive behavior. In the following, the fulfillment of the requirements presented in Chapter 4 is discussed. This section first discusses the functional requirements and then the non-functional requirements. As each requirement is based on the research questions specified in the introduction as well as the insights of the requirements chapter, we answer the three research questions by discussing all of them. Finally, Section 9.2 discusses possible threats to validity and future work.

9.1. Fulfillment of the Requirements

This section discusses the fulfillment of functional and non-functional requirements. Table 9.1 gives an overview while the following sections outline more details.

Functional Requirement	E	Non-Functional Requirement	E
R_F1: Support for all Self-* Properties	•	$R_{NF}1$: Generalizability	•
R_F2: Ready-to-Use Decision Engine	•	$R_{NF}2$: Simple Specification	○
R_F3: Multi-Language Support	•	$R_{NF}3$: Performance	○
R_F4: Language-Independent Predef. Interfaces	•	$R_{NF}4$: Reusability	•
R_F5: Support for Existing Systems	•	$R_{NF}5$: Flexibility	•
R_F6: Development Process	○	$R_{NF}6$: Extensibility.	○
R_F7: Distributed and Dec. Feedback Loops	•		
R_F8: Runtime Monitoring and Modifications	•		

Table 9.1.: Overview of the fulfillment of the functional and non-functional requirements for REACT. The column E (Evaluation) shows if a requirement is fulfilled. Dec: Decentralized, •: fulfilled, ○ partially fulfilled.

9.1.1. Functional Requirements

This section discusses if the proposed functional requirements are fulfilled by the prototypical implementation of REACT.

***R_F*1: Support for all Self-* Properties.** REACT consisting of REACT Core and the different REACT Loops does not restrict its use considering the four self-* properties. Self-configuration is the foundation for the other self-* properties (cf. Section 2.1.1). Accordingly, REACT handles self-configuration for, e.g., starting and stopping brokers or setting cache list parameters in the Tasklet use case. Considering the other self-* properties, the use cases reach from self-healing in the SWIM use case, when the response times exceed certain thresholds (cf. Section 7.3) to self-optimization in the WSN use case (cf. Section 7.2). Self-protection has not been covered explicitly. However, it is easily possible to deploy, e.g., packet drop flow rules instead of packet duplication rules in the case of the SDN-based Wifi handover case of Section 7.3. Thus, this would result in an adaptive firewall functionality for self-protection. Accordingly, we consider R_F1 as fulfilled.

***R_F*2: Ready-to-Use Decision Engine.** In order to provide a ready-to-use decision engine, REACT offers different model-based feedback loop instantiations. The developer has to choose a loop instance and can then use the corresponding modeling capability. Depending on the requirements, in our case, a system developer can choose from SAT, MILP, or CSP-based ready-to-use REACT Loops. This makes it possible to directly apply REACT instead of writing MAPE-K components from scratch. Hence, this allows building a SAS without explicit knowledge about building MAPE-K components. Accordingly, REACT provides ready-to-use engines for making decisions fulfilling requirement R_F2.

***R_F*3: Multi-Language Support.** One concern of REACT is to support a wide range of different managed resources. As the heterogeneity of systems includes different programming languages, REACT can be used with a multitude of languages. In the evaluations, we used Java in the WSN and SDN evaluations (cf. Sections 7.2 and 7.3) and Python in the SWIM case (cf. Section 7.3). In Section 7.4, presenting a comparison of the feedback loop instances, C++ has been applied. Therefore, REACT fulfills requirement R_F3.

***R_F*4: Language-Independent Predefined Interfaces.** Connected with the previous requirement of multi-language support, language-independent predefined

interfaces are needed. In REACT, the interfaces facilitate the data exchange between managed resource and the used REACT Loop, as well as the capability of influencing and changing the deployment at runtime. The system developer is enabled to use these fixed interfaces for connecting a system to REACT and changing it. This provides fixed conventions for programmers. REACT uses the IDL-based RPC framework ZeroC Ice as part of its implementation, which provides support for many programming languages and assists developers with IDE plugins. Accordingly, requirement R_F4 is fulfilled.

***R_F*5: Support for Existing Systems.** Instead of always building SAS by forcing developers to rebuild their managed resource using a specific approach, REACT's goal is to provide the possibility to support legacy systems. This is achieved by providing an external adaptation logic approach. REACT Core provides interfaces, which can be used by any existing system, as long as this system supports monitoring its own and the environment's state, and is capable of applying adaptations. As shown in the different evaluations, none of the use cases had to be rewritten for applying REACT. This results in support for existing legacy systems fulfilling requirement R_F5.

***R_F*6: Development Process.** In order to provide system developers a guideline for applying REACT, the three-step development process specified in Section 6.4 provides a procedure for applying the different feedback loop instantiations. The three steps differentiate only in the used modeling approach and programming language. To further support developers, the two visualization approaches CoalaViz (cf. Section 8.1) and EnTrace (cf. Section 8.2) support developers at development time by providing monitoring capabilities. However, for a full development life cycle, the overall process is too broad and not detailed enough. Additionally, compared to, e.g., the FESAS [21] approach, only the system developer, who is comparable to the designer role in the FESAS approach, is considered here. So, in case a new instance of a REACT-based loop should be created, there is no process available yet. Also, currently, there is no repository for the model-based specifications, which could help in applying REACT even further, closing gaps in the development process. So, we consider requirement R_F6 as partly fulfilled.

***R_F*7: Distributed and Decentralized Feedback Loops.** REACT Core provides the necessary foundations for distributed and decentralized feedback loops as its OSGi-based runtime environment is able to run complete loops or parts

of it. When specifying the distributed loop structure, REACT Core handles the communication between the distributed loop elements. In the SDN evaluation (cf. Section 7.3) we showed the instantiation of the regional planning pattern [15] as decentralization pattern. In this case, the planner ran on a dedicated and faster machine compared to the rest of the feedback loop. Additionally, we showed the use of multiple distributed sensors in this use case. Thus, R_F7 is fulfilled.

***R_F*8: Runtime Monitoring and Modifications.** REACT has been designed to directly support runtime monitoring and modifications. The OSGi runtime used as part of REACT Core enables the reconfiguration of MAPE-K components on the fly by changing the configuration files. Additionally, OSGi provides REACT with the possibility to start and stop instances of the different MAPE-K components at runtime. Besides changing the configuration and distribution of the MAPE-K components themselves, the knowledge component's interface enables the update of specifications at runtime as well. This can be used to directly influence the adaptation behavior. In combination with the context module (cf. Section 6.3) these runtime modification possibilities allow developers to reason on the past contexts and adaptations for applying self-improvement by changing the MAPE-K deployment or the specifications at runtime. Additionally, CoalaViz and EnTrace enable to monitor a REACT-based system as well as to change the weights of non-functional goals at runtime. Still, there are open issues such as handling state, e.g., when updating the specifications. Also, detecting quiescence in the adaptation logic before reconfiguring it would help to reconfigure consistently and without concurrency problems. Accordingly, currently it is the user's responsibility to modify an instance of REACT at the right point in time. Summing up, REACT functionally provides runtime monitoring and modifications, while there are interesting possibilities for future work. Requirement R_F8 is considered as fulfilled.

9.1.2. Non-Functional Requirements

While the functional requirements provide the core functionalities for a model-based runtime environment for adapting communication systems, non-functional requirements measure how well the system provides its functionality. In the following, we discuss REACT's fulfillment of the proposed non-functional requirements.

***R_{NF}1*: Generalizability.** REACT is as general as possible to enable the application in different use cases. Design and implementation of REACT, including REACT Core, as well as the three REACT Loops, are not customized or built with a specific use case in mind. Rather, all parts can be used in a generalized fashion. As the different evaluations show, REACT could be applied in various use cases, using the different REACT Loops. Additionally, even though REACT has been developed with communication systems in mind, in the feasibility study as part of Section 6.3, it could also be applied in a smart crossroad scenario adapting the traffic lights, which is not considered a communication system. Accordingly, the generalizability is considered as fulfilled.

***R_{NF}2*: Simple Specification.** As the goal of REACT is to offer system developers the possibility of specifying adaptation behavior without programming a feedback loop, the simplicity of the used specification approach is crucial. Based on the four possibilities for decision-making presented in Section 2.3, we decided to select a model-based approach for REACT. We state that models offer a good level of abstraction while still providing many possibilities. Models often can also be tested at design time by providing sample input and by applying model-checking to verify the model in general. Specifically, the ready-to-use feedback loops use a custom meta-model, CardyGAn [229], or Clafer [237] for modeling the reconfiguration or problem space. All approaches follow the feature modeling approach of DSPLs. In general, the gap between the concept of UML class diagrams and feature modeling is not high, as presented in [284] and [285], where UML is used to specify feature models. Instead of a UML-based approach, in the MILP-based REACT Loop, CardyGAN employs a domain-specific language for modeling DSPLs and provides an Eclipse-based toolset. This includes a model-checker and an instance generator. Clafer is a structural modeling language, which also provides tool support for testing the model with example input and creating instances of the model. Considering the simplicity of modeling in the case of Clafer, in [238], a scenario for modeling a room booking system is described. The example shows that Clafer itself needs a few concepts, which can possibly be used easily by system developers. The model of the solution space of REACT already uses UML class diagrams, which does not impose the need to learn new concepts for software engineers. When taking the comparison between the CP-based REACT Loop and Rainbow of Section 7.3 into account, we saw that the CP-based REACT Loop

needed fewer different files for the specification of the same behavior. Additionally, the SLOC needed for the same behavior was also lower in the case of the CP-based REACT Loop in this comparison. By that, the focus of REACT on a simple specification of the problem and solution space helps in applying it for adapting a communication system. Based on these observations, we conclude that the design-wise specification of the models is as simple as possible. However, the complete specification process when applying REACT has not been evaluated with practitioners and leaves opportunities for future work. As it is not clear how easy the specification capabilities are in practice, this non-functional requirement is considered partially fulfilled.

***R_{NF}*3: Performance.** The performance of an adaptation logic as part of a SAS determines how fast a decision can be made, e.g., reacting to changes in the execution environment. In Chapter 7, different REACT Loops have been presented. When we compared the CP-based REACT Loop with Rainbow [17] in Section 7.3, the measurements reveal that even though REACT performs better, it still requires approximately 84 ms for executing the entire CP-based feedback loop, with the planner needing 80 ms on average and thereby the vast majority of the time. As shown in Section 7.4, comparing the feedback loops reveals different runtime properties for each of them. Depending on the applied feedback loop instantiation and corresponding modeling approach, the planner runtimes differ considerably. Even when using SAT4J, which is the least expressive planner missing support for integer or real values, the planner component alone needed approximately 13 ms on average. These results indicate that REACT is more suitable for managed resources requiring fast changes compared to Rainbow. Still, especially in the networking domain, this is not sufficient for all use cases as, e.g., for the extreme use case of deciding on the packet level, how each packet should individually be treated. Hence, the current prototype of REACT can be used on higher, more strategic levels, such as in the adaptive SDN use case (cf. Section 7.3). This results in opportunities for future work and in the fact that the performance non-functional requirement is considered as partially fulfilled.

***R_{NF}*4: Reusability.** In order to provide an added value for system developers, a framework or runtime environment should provide a high degree of reusability. Looking at REACT from the perspective of the system developers, they can completely reuse the implementation of REACT consisting of REACT Core and

the implemented REACT Loops. Additionally, in case a system developer created an intermediate layer connecting a managed resource to REACT, this intermediate layer can be reused to a large degree, as it contains the RPC-based connection to REACT. In the best case, such as in the SDN evaluation (cf. Section 7.3), the sensor and effector implementations are generalized as well. Specifically, in the SDN use case, the sensor and effector implementations can be reused entirely for different use cases. In the SDN example, the sensor, as part of the SDN controller, provides general information about the topology, while the effector can deploy arbitrary flow rules. Thus, both implementations can possibly be reused for other adaptation specifications and scenarios. Additionally, REACT Core as a foundation provides a lot of reusable services and structures for engineering custom feedback loops consisting, e.g., of the provided interfaces and communication facilities to the deployment capabilities. All in all, the reusability of REACT is considered as high leading to the fulfillment of requirement $R_{NF}4$.

***R_{NF}5*: Flexibility.** When we consider a REACT-based feedback loop as managed resource, as it is done in the hierarchical control pattern [15] as well as in self-improvement [21], changes are also possible using parameter and compositional adaptation. A high degree of flexibility allows us changing the specification as well as the actual deployment of the feedback loop itself. Both types of change are possible with REACT. The provided knowledge interface supports to change the specifications at runtime while the OSGi-based runtime as part of REACT Core permits updating the deployment itself. These changes can either be executed using a separate feedback loop or manually by some administrator or developer. Based on these observations, REACT is considered as flexible, fulfilling requirement $R_{NF}5$.

***R_{NF}6*: Extensibility.** Extensibility requires that the provided solution can be extended with additional functionality, which is not in place yet. REACT enables extensibility by providing different interfaces developers can use to extend the provided functionality. This includes the possibility for developing own monitoring strategies in the case of the provided feedback loops. In general, REACT Core provides the capability of writing a custom feedback loop instance using the provided supporting structures. Accordingly, this leaves possibilities for further development, such as providing interfaces for custom analyzing strategies without writing a new analyzer from scratch. The same need for additional interfaces is

imposed by the planner, which could enable a way of changing the solver to a custom one or use a solver, which is not used as part of a provided REACT Loop. Based on these observations, we consider requirement $R_{NF}6$ as partially fulfilled.

9.2. Threats to Validity and Limitations

This section outlines threats to validity and limitations of this thesis' approach. First, this section discusses REACT Core. Second, the REACT Loop instantiations are examined. Finally, this section discusses CoalaViz and EnTrace.

9.2.1. REACT Core

Generally, for identifying the gaps of existing approaches, Chapter 5 examines related work. However, this related work in the field of self-adaptive systems focuses on implementation approaches only. Thus, it is possible that software architectures or (formal) methodologies with similar objectives as REACT and REACT Core are available. Still, as REACT aims at providing a useable software artifact, the related work in the implementation approach is considered as justified for the comparison.

REACT Core itself uses the well-known and broadly used MAPE-K loop as the architectural abstraction of the feedback loop. There are also other approaches such as the learning, reasoning, and acting loop using model-based learning (LRA-M) [23] from the self-aware computing domain as well as the organic computing-based loop [25]. Although MAPE-K ist the de-facto standard architecture, (dis-) advantages of the different architectures could lead to changing results. Thus, in future work, other architectures for abstracting feedback loops could be investigated as part of REACT.

Considering the current design of REACT Core, each component can only have a single successor component. Multiple successors could help to deploy more sophisticated MAPE-K patterns. However, this is mostly a question of engineering. As REACT's components do not contain a reference to potential predecessors, multiple inputs are already possible, as shown in the SDN case with multiple sensors (cf. Section 7.2).

REACT Core is implemented using ZeroC Ice [193]. The RPC framework is a fundamental component of REACT Core determining the communication behavior between REACT's components as well as the communication between the adaptation logic and the managed resource. For determining and for quantifying the influence of Ice, other RPC frameworks could be used for comparison. For the communication, other communication facilities could be interesting as well. As an example, an MQTT-based solution, which has already been used in REACT Core's context module, could further decouple the components from each other. This would make the behavior even more transparent, as all inputs and outputs are visible on the broker side. Hence, this could also be used for engineering an even more advanced solution for traceability in comparison to EnTrace (cf. Section 8.2).

Also, more (external) interfaces could help to increase the customization possibilities. This includes ways to create custom analyzing and planning techniques. Currently, the provided analyzers and planners have to be used. This limits the applicability of custom machine learning techniques of the current implementation. As of the current implementation, a REACT Loop has to explicitly support machine learning results or techniques. Such support has been integrated into the MILP-based REACT Loop, which supports the results of SPL Conqueror for taking non-functional performance influences into account.

9.2.2. Feedback Loop Instantiations

Looking at the feedback loop instantiations, besides the model- and problem domain-specific limitations, there are other limitations considering the implementations themselves.

As part of the REACT Loops, CFM model-based specifications have been applied for the problem space since they provide a sufficient level of abstraction for system developers. Accordingly, other modeling approaches for specifying the adaptation behavior as part of REACT should be investigated. From the perspective of the developers, the specification of higher-level goals is considered simpler compared to specifying models. Therefore, using goal-based specifications instead of models could lead to even better usability but this also imposes new challenges.

The context module, as part of REACT Core, is mostly not used in the evaluations. The reason for this is that for measuring the effectiveness and efficiency of a REACT Loop skipping parts of it hinders to measure the performance of the loop properly. Hence, for evaluating a REACT Loop integrating the context module would require to execute more runs to get enough measurements of the loop itself.

The MILP-based REACT Loop needs many measurements in case the non-functional properties should be taken into account (cf. Section 7.2). These measurements are used to learn performance-influence models and can also be taken from a real system. However, as a real system should not run in problematic situations, it is not easily possible to explore the entire reconfiguration space. Therefore, for applying the MILP-based REACT Loop in combination with performance-influence models possibly requires a simulator of the real system.

The CP-based feedback loop currently does not support manually weighted multi-objective optimization. However, weighting objectives is possible in Rainbow as well as in the MILP-based feedback loop of REACT. One possible improvement is to integrate an easy-to-use API for system developers to forward multiple, weighted optimization goals to the Chocosolver in future work. Also, it could be possible to model an objective function as Clafer attribute consisting of multiple objective variables manually. Another drawback of the CP-based REACT Loop is that although Clafer itself enables to specify real-valued attributes, currently there is no backend supporting them.

In the evaluation of the CP-based feedback loop, we measure the SLOC and the number of different languages to show its low development effort for system developers in comparison with Rainbow. First, there might be simpler ways to model the adaptation behavior either in REACT or in Rainbow. Hence, the comparison depends on the experience of the person specifying the models. Second, even though SLOC are frequently used as a metric (e.g., in [16, 18, 247]), other metrics such as the modeling or cognitive effort [286, 287] could be taken into account as well. In the future, it would be beneficial to conduct a study with system developers who apply REACT in different scenarios for strengthening validity. This could include more sophisticated measurements instead of only taking the SLOC into account. Additionally, e.g., by conducting structured interviews or using the think-aloud methodology [288], problems with our approach could be identified more precisely.

As part of this thesis, REACT was only compared to the state-of-the-art approach Rainbow using the CP-based REACT Loop. Future research may include a comparison to other frameworks such as SASSY [14] or StarMX [144] in combination with additional use cases from the communication systems domain.

Also, the number of use cases of this thesis is limited. Specifically, the evaluations do not contain measurements of running real-world communication systems. However, since the SDN evaluation in Section 7.3 is an emulated network, this is de-facto a real-world communication system sending actual packets. In future work, REACT could also be evaluated by deploying it in a working system.

Taking scalability into account, it would be interesting to observe the scalability of our approach as far as (i) large Clafer and UML models and (ii) larger system sizes are concerned. According to [241], Clafer itself is considered as scalable, as it is able to find an instance of a model in realistic feature models considerably fast. However, scalability testing using Clafer with REACT is interesting as well.

As stated in the discussion of R_F1, the evaluations did not take self-protection specifically into account. In fact, this is also the least researched self-* property so far [39]. Still, as mentioned, the SDN evaluation could easily be changed into a firewall scenario, where the flow rules discard packets instead of duplicating them.

Finally, there was no end-to-end evaluation of REACT using all capabilities in a single use case, including the context module, a REACT Loop, and EnTrace.

9.2.3. Visualization of REACT

Examining the visualization approaches CoalaViz and EnTrace, there are also limitations in place. Besides the (technical) limitations already described as part of Chapter 8, a limitation is the way of the evaluation. Currently, the functionality of both approaches has been evaluated qualitatively, while the performance was evaluated quantitatively. However, this is rather limited when considering the goals of the systems. To ensure the goal of providing a traceability solution for system developers and administrators is achieved in both systems, a user study should be conducted in future work. This user study would need participants from the described groups when developing or deploying a SAS. Therefore, mainly professionals have to be incorporated as part of such a study.

10. Conclusion and Outlook

The previous chapters presented REACT, consisting of REACT Core, REACT-based feedback loop instantiations, as well as CoalaViz and EnTrace. In the discussion, we examined the fulfillment of the requirements as well as limitations and possible solutions for them. This chapter closes this thesis with a conclusion, an outlook, and additional options for future work, which are not based on specific limitations.

10.1. Conclusion

With the growing number of networked devices, the management effort and complexity of networks are constantly increasing. These facts induce the need for adaptive capabilities in communication systems. This thesis presented a runtime environment for adapting communication systems called REACT. The overall goal of the thesis is providing a *model-based* and reusable *runtime environment* for adapting *communication systems* with the possibility to *change the deployment at runtime*. For tackling this objective, the thesis follows the design science research methodology of Peffers *et al.* [22], beginning with the problem of adding adaptive behavior to communication systems. Based on a requirements analysis, categories for comparing existing works are presented. This categorization is used to compare related works revealing that the specific requirements for adapting (existing) communication systems are not fulfilled by an existing approach. The main missing properties in the existing work considering SAS engineering approaches consist of the support for decentralized feedback loop deployments, predefined interfaces including multi-language support, as well as a specified development process. Looking at recent Autonomic Networking approaches, all of them need to completely rebuild systems from scratch to be applied.

This thesis fills this research gap with REACT. REACT consists of an OSGi-based core, which is able to execute (predefined) REACT Loops distributedly, following

the concept of decentralized control. REACT Core handles the communication within the adaptation logic and with the managed resource. Additionally, it supports system developers with a development process they can follow for adding adaptive behavior to an existing or new system. Based on REACT Core, three different REACT Loops with different modeling and problem-solving capabilities have been presented. They have been evaluated in different use cases as well compared against each other. We showed that the modeling effort using the CP-based REACT Loop is comparably low compared to Rainbow and that the proposed requirements are in general fulfilled. The visualization approaches for tracing adaptation decisions additionally support system developers and administrators. Hence, overall, REACT enables to tame the complexity of networked devices by providing adaptive behavior to them.

10.2. Outlook

Besides the mentioned limitations, REACT offers various opportunities for further research. First, currently, REACT's feedback loops have to be deployed without taking interdependencies between the adaptive system and other (adaptive) systems into account. This offers options for coordination between different (sub) systems, including the idea to add consensus protocols for adapting cooperatively in a coordinated way. This would allow to, e.g., globally switch from one serialization method to another without breaking the data flow, in case only parts of the system adapt while others keep the current method.

Considering the functional aspect of runtime modifications, there are open questions for handling the current situation of the adaptation logic when the deployment gets modified. In future work, it would be interesting to explicitly handle modifications of the specifications and the deployment. This could, e.g., make sure that no adaptation is running while modifying the adaptation logic. Also, in future work the context manager needs to be updated in terms of changing database schemas according to changing specifications. Finally, knowledge as part of the context manager could get obsolete in the case of changes, which is also an issue for future work.

Other than that, CFMs possibly could be decomposed into system-specific subtrees assigned to different parts of an actual system. This enables to specifically model

constraints between the (sub) systems. Also, each subtree could be used with a different REACT Loop, depending on the requirements of the system developer. Additionally, based on the assigned parts, REACT could be enhanced to directly support automatic deployment. This would lower the administrative effort for the deployment even more.

As shown in this thesis, feedback loops have different modeling and runtime properties based on the used modeling technology and solver. In future work, the combination of feedback loops could lead to new ways of solving different problems. This combination could be done as part of a self-improvement module, which uses one loop for the parameters only, while another one is selected for the compositional adaptation. Additionally, the selection of a feedback loop based on heuristics or machine learning-based models could improve the performance of the loop itself and optimize the adaptation decisions considering the different problem domains. Also, Clafer can be used in combination with other solver backends besides a CP-based solver [237]. Hence, an adaptive Clafer-based REACT Loop could automatically detect the employed capabilities and Clafer types to adaptively select the fastest or most suitable solver backend at runtime.

Looking at the specification options of the REACT Loops, Clafer shows an adequate level of abstraction. As it represents a solid foundation for future work, it would be possible to extend Clafer with notions to express (real-)time aspects when reconfiguring from one feature to another, e.g., when information has to be collected before a feature can be activated. Also, currently, it is always possible to adapt to any other (system) configuration without constraints. Extending the Clafer-based specifications with particular capabilities to model disallowed switches between features could further increase their expressiveness.

In general, the integration of machine learning techniques into the REACT Loops is a relevant field for further research. Also, verification and validation methodologies can directly be integrated into REACT. Finally, REACT could be evaluated in further use cases. As shown in [289][1], REACT could be applied and evaluated in car-to-cloud communication scenarios in the future. This not only includes simulations but also experiments with actual communication devices in a testbed.

[1] [289] is joint work with S. Herrnleben, C. Krupitzer, S. Kounev, M. Segata, F. Fastnacht, and M. Nigmann.

Bibliography

[1] International Data Corporation, "The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast," https://www.idc.com/getdoc.jsp?containerId=prUS45213219, 2019, accessed: 2020-12-18.

[2] ——, "Worldwide Connected Vehicle Shipments Forecast to Reach 76 Million Units by 2023, According to IDC," https://apnews.com/press-release/pr-businesswire/d72b6ba2039540aabcce8f1e96721edc, 2019, accessed: 2020-12-18.

[3] R. Laddaga, P. Robertson, and H. E. Shrobe, "Introduction to self-adaptive software: Applications," in *Proceedings of the International Workshop on Self-Adaptive Software (IWSAS)*, ser. Lecture Notes in Computer Science, vol. 2614. Springer, 2001, pp. 1–5.

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[5] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May 1999.

[6] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, Jul. 2004.

[7] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, ser. Lecture Notes in Computer Science, vol. 5525. Springer, 2009, pp. 48–70.

[8] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, vol. 5525. Springer, 2009, pp. 1–26.

[9] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[10] C. Krupitzer, M. Pfannemüller, V. Voss, and C. Becker, "Comparison of approaches for developing self-adaptive systems," Chair of Information Systems II, University of Mannheim, Germany, Tech. Rep., 2018, https://ub-madoc.bib.uni-mannheim.de/43909.

[11] M. Pfannemüller, M. Breitbach, C. Krupitzer, M. Weckesser, C. Becker, B. Schmerl, and A. Schürr, "REACT: A Model-Based Runtime Environment for Adapting Communication Systems," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020.

[12] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. O. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, "MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, vol. 5525. Springer, 2009, pp. 164–182.

[13] S. O. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. A. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2840–2859, 2012.

[14] D. A. Menascé, H. Gomaa, S. Malek, and J. P. Sousa, "SASSY: A framework for self-architecting service-oriented systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, 2011.

[15] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, vol. 7475. Springer, 2010, pp. 76–107.

[16] C. Krupitzer, F. M. Roth, M. Pfannemüller, and C. Becker, "Comparison of approaches for self-improvement in self-adaptive systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE, 2016, pp. 308–314.

[17] D. Garlan, S. Cheng, A. Huang, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[18] S.-W. Cheng, "Rainbow: cost-effective software architecture-based self-adaptation," Ph.D. dissertation, Carnegie Mellon University, 2004.

[19] C. Krupitzer, S. VanSyckel, and C. Becker, "FESAS: towards a framework for engineering self-adaptive systems," in *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2013, pp. 263–264.

[20] C. Krupitzer, F. M. Roth, S. VanSyckel, and C. Becker, "Towards reusability in autonomic computing," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE, 2015, pp. 115–120.

[21] C. Krupitzer, F. M. Roth, C. Becker, M. Weckesser, M. Lochau, and A. Schürr, "FESAS IDE: An Integrated Development Environment for Autonomic Computing," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE, 2016, pp. 15–24.

[22] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2008.

[23] S. Kounev, P. R. Lewis, K. L. Bellman, N. Bencomo, J. Cámara, A. Diaconescu, L. Esterle, K. Geihs, H. Giese, S. Götz, P. Inverardi, J. O. Kephart, and A. Zisman, "The notion of self-aware computing," in *Self-Aware Computing Systems*. Springer, 2017, pp. 3–16.

[24] D. Weyns, "Software engineering of self-adaptive systems," in *Handbook of Software Engineering*. Springer, 2019, pp. 399–443.

[25] S. Tomforde, "An architectural framework for self-configuration and self-improvement at runtime," Ph.D. dissertation, University of Hannover, 2011.

[26] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, pp. 14:1–14:42, 2009.

[27] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing - degrees, models, and applications," *ACM Computing Surveys*, vol. 40, no. 3, pp. 7:1–7:28, 2008.

[28] Z. Movahedi, M. Ayari, R. Langar, and G. Pujolle, "A survey of autonomic network architectures and evaluation criteria," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 2, pp. 464–490, 2012.

[29] A. S. Tanenbaum and A. S. Woodhull, *Operating systems - design and implementation, 3rd Edition*. Pearson Education, 2006.

[30] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing - Principles, Design and Implementation*, ser. Undergraduate Topics in Computer Science. Springer, 2013.

[31] M. G. Hinchey and R. Sterritt, "Self-managing software," *IEEE Computer*, vol. 39, no. 2, pp. 107–109, 2006.

[32] B. N. Schilit, N. Adams, and R. Want, "Context-aware computing applications," in *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)*. IEEE, 1994, pp. 85–90.

[33] A. K. Dey, "Understanding and using context," *Personal and Ubiquitous Computing*, vol. 5, no. 1, pp. 4–7, 2001.

[34] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Proceedings of the International Workshop on Unconventional Programming Paradigms (UPP)*, ser. Lecture Notes in Computer Science, vol. 3566. Springer, 2004, pp. 257–269.

[35] P. Vromant, D. Weyns, S. Malek, and J. Andersson, "On interacting control loops in self-adaptive systems," in *Proceedings of the International Sym-*

posium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). ACM, 2011, pp. 202–207.

[36] A. M. Elkhodary, N. Esfahani, and S. Malek, "FUSION: a framework for engineering self-tuning self-adaptive software systems," in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 7–16.

[37] B. R. Schmerl, J. Cámara, J. Gennari, D. Garlan, P. Casanova, G. A. Moreno, T. J. Glazier, and J. M. Barnes, "Architecture-based self-protection: composing and reasoning about denial-of-service mitigations," in *Proceedings of the Symposium and Bootcamp on the Science of Security (HotSoS)*. ACM, 2014, p. 2.

[38] T. Vogel, "mRUBiS: an exemplar for model-based architectural self-healing and self-optimization," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018, pp. 101–107.

[39] D. Weyns, M. U. Iftikhar, S. Malek, and J. Andersson, "Claims and supporting evidence for self-adaptive systems: A literature study," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2012, pp. 89–98.

[40] J. Kephart, J. Kephart, D. Chess, C. Boutilier, R. Das, J. O. Kephart, and W. E. Walsh, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2003.

[41] Y. Brun, R. J. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, "A design space for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, vol. 7475. Springer, 2010, pp. 33–50.

[42] N. J. Nilsson, *Principles of Artificial Intelligence*, ser. Symbolic computation. Springer, 1982.

[43] E. Gat, *Three-Layer Architectures*. MIT Press, 1998, p. 195–210.

[44] C. Perera, A. B. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[45] A. Frömmgen, R. Rehner, M. Lehn, and A. P. Buchmann, "Fossa: Learning ECA rules for adaptive distributed systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE, 2015, pp. 207–210.

[46] T. Kühne, "Matters of (meta-)modeling," *Software and Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[47] J. P. S. da Silva, M. Ecar, M. S. Pimenta, G. T. A. Guedes, L. P. Franz, and L. Marchezan, "A systematic literature review of uml-based domain-specific modeling languages for self-adaptive systems," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018, pp. 87–93.

[48] D. Garlan, R. T. Monroe, and D. Wile, "Acme: an architecture description interchange language," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, 1997, p. 7.

[49] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990.

[50] C. Ghezzi, A. Mocci, and M. Sangiorgio, "Runtime monitoring of functional component changes with behavior models," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. Lecture Notes in Computer Science, vol. 7167. Springer, 2011, pp. 152–166.

[51] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Proceedings of the Conference on the Future of Software Engineering (FOSE)*. IEEE, 2007, pp. 259–268.

[52] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni, "A survey of context modelling and reasoning techniques," *Pervasive and Mobile Computing*, vol. 6, no. 2, pp. 161–180, 2010.

[53] H. Hartmann and T. Trew, "Using feature diagrams with context variability to model multiple product lines for software supply chains," in *Proceedings of the Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 12–21.

[54] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes, "Using constraint programming to manage configurations in self-adaptive systems," *IEEE Computer*, vol. 45, no. 10, pp. 56–63, 2012.

[55] I. Gerostathopoulos, T. Bures, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, and N. Plouzeau, "Self-adaptation in software-intensive cyber-physical systems: From system goals to architecture configurations," *Journal of Systems and Software*, vol. 122, pp. 378–397, 2016.

[56] S. Götz, T. Kühn, C. Piechnick, G. Püschel, and U. Aßmann, "A models@run.time approach for multi-objective self-optimizing software," in *Proceedings of the International Conference on Adaptive and Intelligent Systems (ICAIS)*, ser. Lecture Notes in Computer Science, vol. 8779. Springer, 2014, pp. 100–109.

[57] A. Paz and H. Arboleda, "A model to guide dynamic adaptation planning in self-adaptive systems," in *Selected Papers of the Latin American Computer Conference*, ser. Electronic Notes in Theoretical Computer Science, vol. 321. Elsevier, 2016, pp. 67–88.

[58] E. M. Fredericks, B. DeVries, and B. H. C. Cheng, "Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 17–26.

[59] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[60] M. Salehie and L. Tahvildari, "Towards a goal-driven approach to action selection in self-adaptive software," *Software: Practice and Experience*, vol. 42, no. 2, pp. 211–233, 2012.

[61] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From goals to components: a combined approach to self-management," in *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2008, pp. 1–8.

[62] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "A case study in goal-driven architectural adaptation," in *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, ser. Lecture Notes in Computer Science, vol. 5525. Springer, 2009, pp. 109–127.

[63] W. N. Robinson, "A requirements monitoring framework for enterprise systems," *Requirements Engineering*, vol. 11, no. 1, pp. 17–41, 2006.

[64] Y. Wang, S. A. McIlraith, Y. Yu, and J. Mylopoulos, "An automated approach to monitoring and diagnosing requirements," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 293–302.

[65] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IEEE, 2004, pp. 464–471.

[66] S. O. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *IEEE Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[67] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Proceedings of the International Conference Companion on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2003, pp. 16–27.

[68] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[69] J. D. Goldhar and M. Jelinek, "Plan for economies of scope," *Harvard Business Review*, vol. 61, no. 6, pp. 141–148, 1983.

[70] S. M. Davis, *Future Perfect*. Addison-Wesley, 1987.

[71] C. M. U. Software Engineering Institute, "Software product line essentials," https://resources.sei.cmu.edu/asset_files/Presentation/2008_017_001_24246.pdf, accessed: 2020-12-18.

[72] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration using feature models," in *Proceedings of the Software Product Line Conference (SPLC)*, ser. Lecture Notes in Computer Science, vol. 3154. Springer, 2004, pp. 266–283.

[73] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[74] L. Chen, M. A. Babar, and N. Ali, "Variability management in software product lines: a systematic review," in *Proceedings of the Software Product Line Conference (SPLC)*, vol. 446. ACM, 2009, pp. 81–90.

[75] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, pp. 143–168, 1998.

[76] P. Schobbens, P. Heymans, and J. Trigaux, "Feature diagrams: A survey and a formal semantics," in *Proceedings of the International Conference on Requirements Engineering (RE)*. IEEE, 2006, pp. 136–145.

[77] M. L. Griss, J. M. Favaro, and M. D'Alessandro, "Integrating feature modeling with the RSEB," in *Proceedings of the International Conference on Software Reuse (ICSR)*. IEEE, 1998, pp. 76–85.

[78] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, "Extending feature diagrams with UML multiplicities," in *Proceedings of the Conference on Integrated Design & Process Technology (IDPT)*, vol. 23, 2002, pp. 1–7.

[79] D. Benavides, P. Trinidad, and A. R. Cortés, "Automated reasoning on feature models," in *Proceeding of the International Conference on Advanced Information Systems Engineering (CAiSE)*, ser. Lecture Notes in Computer Science, vol. 3520. Springer, 2005, pp. 491–503.

[80] R. Capilla, J. Bosch, P. Trinidad, A. R. Cortés, and M. Hinchey, "An overview of dynamic software product line architectures and techniques: Observations from research and industry," *Journal of Systems and Software*, vol. 91, pp. 3–23, 2014.

[81] K. Saller, M. Lochau, and I. Reimund, "Context-aware DSPLs: model-based runtime adaptation for resource-constrained systems," in *Proceedings of the Software Product Line Conference (SPLC)*. ACM, 2013, pp. 106–113.

[82] N. Abbas, J. Andersson, and W. Löwe, "Autonomic software product lines (ASPL)," in *Proceedings of the European Conference on Software Architecture (ECSA)*. ACM, 2010, pp. 324–331.

[83] N. Bencomo, S. O. Hallsteinsen, and E. S. de Almeida, "A view of the dynamic software product line landscape," *IEEE Computer*, vol. 45, no. 10, pp. 36–41, 2012.

[84] H. A. Simon, "The science of design: Creating the artificial," *Design Issues*, vol. 4, no. 1/2, pp. 67–82, 1988.

[85] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *Management Information Systems Quarterly (MIS Q.)*, vol. 28, no. 1, pp. 75–105, 2004.

[86] L. A. Maciaszek, *Requirements analysis and system design (3rd edition)*. Addison-Wesley, 2007.

[87] M. Pfannemüller, "Self-adaptive middleware for model-based network adaptations," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2018, pp. 462–463.

[88] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[89] P. E. Rosove, *Developing Computer-Based Information Systems*. Wiley, 1967.

[90] R. Kneuper, "Sixty years of software development life cycle models," *IEEE Annals of the History of Computing*, vol. 39, no. 3, pp. 41–54, 2017.

[91] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM Press, 1987, pp. 328–339.

[92] P. Abrahamsson, N. V. Oza, and M. T. Siponen, "Agile software development methods: A comparative review[1]," in *Agile Software Development - Current Research and Future Directions*. Springer, 2010, pp. 31–59.

[93] R. M. Soley and C. M. Stone, *Object management architecture guide*. John Wiley & Sons, Inc., 1995.

[94] "Common Object Request BrokerArchitecture (CORBA)," 2012, version 3.3, https://www.omg.org/spec/CORBA/3.3.

[95] C. Krupitzer, M. Breitbach, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems (extended version)," Chair of Information Systems II, University of Mannheim, Germany, Tech. Rep., 2018, https://madoc.bib.uni-mannheim.de/44034.

[96] S. Sicard, F. Boyer, and N. D. Palma, "Using Components for Architecture-Based Management: The Self-Repair case," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 101–110.

[97] J. Floch, S. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, 2006.

[98] J. Dowling and V. Cahill, "The K-Component Architecture Meta-model for Self-Adaptive Software," in *Proceedings of the International Conference on Metalevel Architectures and Reflection (REFLECTION)*. Springer, 2001, pp. 81–88.

[99] S. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[100] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of the European Software Engineering Conference (ESEC)*. Springer, 1995, pp. 137–153.

[101] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace, "Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems," in *Proceedings of the Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 23–32.

[102] B. Morin, O. Barais, J. Jézéquel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *IEEE Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[103] N. Gámez, L. Fuentes, and J. M. Troya, "Creating self-adapting mobile systems with dynamic software product lines," *IEEE Software*, vol. 32, no. 2, pp. 105–112, 2015.

[104] M. Pfannemüller, C. Krupitzer, M. Weckesser, and C. Becker, "A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2017, pp. 247–254.

[105] G. S. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[106] N. Bencomo, R. B. France, B. Cheng, and U. Aßmann, Eds., *Models@run.time - Foundations, Applications, and Roadmaps*. Springer, 2014.

[107] M. U. Iftikhar and D. Weyns, "ActivFORMS: Active formal models for self-adaptation," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 125–134.

[108] ——, "ActivFORMS: A runtime environment for architecture-based adaptation with guarantees," in *Proceedings of the International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 278–281.

[109] D. Weyns, S. Shevtsov, and S. Pllana, "Providing assurances for self-adaptation in a mobile digital storytelling application using activforms," in *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2014, pp. 110–119.

[110] D. Weyns and M. U. Iftikhar, "ActivFORMS: A model-based approach to engineer self-adaptive systems," *CoRR*, vol. abs/1908.11179, 2019.

[111] D. Weyns, S. Malek, and J. Andersson, "FORMS: a formal reference model for self-adaptation," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. ACM, 2010, pp. 205–214.

[112] ——, "FORMS: unifying reference model for formal specification of distributed self-adaptive systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 1, pp. 8:1–8:61, 2012.

[113] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[114] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "A model-driven approach for developing self-adaptive pervasive systems," *Models@ run.time*, vol. 8, pp. 97–106, 2008.

[115] N. Bencomo, G. S. Blair, and R. B. France, "Model-driven software adaptation," in *European Conference on Object-Oriented Programming (ECOOP)*

Workshop Reader, ser. Lecture Notes in Computer Science, vol. 4906. Springer, 2007, pp. 132–141.

[116] T. Vogel and H. Giese, "Model-driven engineering of adaptation engines for self-adaptive software : executable runtime megamodels," Tech. Rep., 2013.

[117] ——, "Model-driven engineering of self-adaptive software with EUREMA," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 4, pp. 18:1–18:33, 2014.

[118] T. Vogel, A. Seibel, and H. Giese, "Toward megamodels at runtime," in *Proceedings of the Workshop on Models@run.time*, ser. CEUR Workshop Proceedings, vol. 641. CEUR-WS.org, 2010, pp. 13–24.

[119] ——, "The role of models and megamodels at runtime," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ser. Lecture Notes in Computer Science, vol. 6627. Springer, 2010, pp. 224–238.

[120] N. Bencomo, P. Grace, C. A. Flores-Cortés, D. Hughes, and G. S. Blair, "Genie: supporting the model driven development of reflective, component-based adaptive systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 811–814.

[121] P. Grace, G. Coulson, G. S. Blair, and B. Porter, "Deep middleware for the divergent grid," in *Middleware*, ser. Lecture Notes in Computer Science, vol. 3790. Springer, 2005, pp. 334–353.

[122] N. Bencomo and G. Blair, "Genie: a domain-specific modeling tool for the generation of adaptive and reflective middleware families," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Domain-Specific Modeling*, 2006.

[123] M. Amoui, M. Derakhshanmanesh, J. Ebert, and L. Tahvildari, "Achieving dynamic adaptation via management and interpretation of runtime models," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2720–2737, 2012.

[124] J. Ebert, V. Riediger, and A. Winter, "Graph technology in reverse engineering: The tgraph approach," in *Workshop Software Reengineering (WSR)*, ser. LNI, vol. P-126. GI, 2008, pp. 67–81.

[125] E. Zavala, X. Franch, J. Marco, and C. Berger, "HAFLoop: An architecture for supporting Highly Adaptive Feedback Loops in self-adaptive systems," *Future Generation Computer Systems*, vol. 105, pp. 607–630, 2020.

[126] E. B. Zavala Rodríguez, "Towards adaptative monitoring for self-adaptative systems," Ph.D. dissertation, UPC, Departament de Llenguatges i Sistemes Informàtics, Jul 2019.

[127] G. E. Kaiser, A. Stone, and S. E. Dossick, "A mobile agent approach to lightweight process workflow," Tech. Rep., 1999.

[128] G. Valetto, G. E. Kaiser, and G. S. Kc, "A mobile agent approach to process-based dynamic adaptation of complex software systems," in *Proceedings of the European Workshop on Software Process Technology (EWSPT)*, ser. Lecture Notes in Computer Science, vol. 2077. Springer, 2001, pp. 102–116.

[129] G. Valetto and G. E. Kaiser, "A case study in software adaptation," in *Proceedings of the Workshop on Self-Healing Systems (WOSS)*. ACM, 2002, pp. 73–78.

[130] G. E. Kaiser, J. J. Parekh, P. Gross, and G. Valetto, "Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems," in *Active Middleware Services*. IEEE, 2003, pp. 22–31.

[131] J. J. Parekh, G. E. Kaiser, P. Gross, and G. Valetto, "Retrofitting autonomic capabilities onto legacy systems," *Cluster Computing*, vol. 9, no. 2, pp. 141–159, 2006.

[132] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto, "An approach to autonomizing legacy systems," Columbia University New York, Tech. Rep., 2005.

[133] G. Valetto and G. E. Kaiser, "Using process technology to control and coordinate software adaptation," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2003, pp. 262–273.

[134] G. Kaiser and G. Valetto, "Ravages of time: Synchronized multimedia for internet-wide process-centered software engineering environments," in *3rd ICSE Workshop on Software Engineering over the Internet*, 2000.

[135] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. S. Sukhatme, "An architecture-driven software

mobility framework," *Journal of Systems and Software*, vol. 83, no. 6, pp. 972–989, 2010.

[136] S. Malek, "A user-centric approach for improving a distributed software system's deployment architecture," Ph.D. dissertation, University of Southern California, 2007.

[137] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola, "MOSES: A framework for qos driven runtime adaptation of service-oriented systems," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1138–1159, 2012.

[138] T. Preisler, T. Dethlefs, and W. Renz, "Middleware for constructing decentralized control in self-organizing systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE, 2015, pp. 325–330.

[139] B. R. Schmerl and D. Garlan, "Acmestudio: Supporting style-centered architecture development," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2004, pp. 704–705.

[140] V. E. S. Souza, "Requirements-based software system adaptation," Ph.D. dissertation, University of Trento, Italy, 2012.

[141] K. Angelopoulos, V. E. S. Souza, and J. Pimentel, "Requirements and architectural approaches to adaptive software systems: A comparative study," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2013, pp. 23–32.

[142] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the Rainbow: Self-Adaptive Failure Avoidance in Configurable Systems," in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 377–388.

[143] S. A. White, "Introduction to BPMN," Tech. Rep., 2004, IBM Cooperation.

[144] R. Asadollahi, M. Salehie, and L. Tahvildari, "StarMX: A framework for developing self-managing Java-based systems," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2009, pp. 58–67.

[145] S. Tomforde, H. Prothmann, F. Rochner, J. Branke, J. Hähner, C. Müller-Schloer, and H. Schmeck, "Decentralised progressive signal systems for organic traffic control," in *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2008, pp. 413–422.

[146] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer, and H. Schmeck, "Organic control of traffic lights," in *Proceedings of the International Conference on Autonomic and Trusted Computing (ATC)*, ser. Lecture Notes in Computer Science, vol. 5060. Springer, 2008, pp. 219–233.

[147] H. Prothmann, S. Tomforde, J. Branke, J. Hähner, C. Müller-Schloer, and H. Schmeck, "Organic traffic control," in *Organic Computing*. Springer, 2011, pp. 431–446.

[148] B. Hurling, S. Tomforde, and J. Hähner, "Organic network control," in *Organic Computing*. Springer, 2011, pp. 611–613.

[149] B. Hurling, S. Tomforde, S. Rudolph, J. Hähner, and C. Müller-Schloer, "Evolving network protocols," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. ACM, 2011, pp. 173–174.

[150] S. Tomforde, M. Steffen, J. Hähner, and C. Müller-Schloer, "Towards an organic network control system," in *Proceedings of the International Conference on Autonomic and Trusted Computing (ATC)*, ser. Lecture Notes in Computer Science, vol. 5586. Springer, 2009, pp. 2–16.

[151] S. Dobson, S. G. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223–259, 2006.

[152] A. Bassi, S. Denazis, A. Galis, C. Fahy, M. Serrano, and J. Serrat, "Autonomic internet: A perspective for future internet services based on autonomic principles," *International Week on Management of Networks and Services (MANWEEK)*, 2007.

[153] A. Galis, S. G. Denazis, A. Bassi, P. Giacomin, A. Berl, A. Fischer, H. de Meer, J. Srassner, S. Davy, D. F. Macedo, G. Pujolle, J. Rubio-Loyola, J. Serrat, L. Lefèvre, and A. Cheniour, "Management architecture

and systems for future internet networks," in *Future Internet Assembly*. IOS Press, 2009, pp. 112–122.

[154] D. F. Macedo, Z. Movahedi, J. Rubio-Loyola, A. Astorga, G. V. Koumoutsos, and G. Pujolle, "The autoi approach for the orchestration of autonomic networks," *Annals of Telecommunications - Annales des Télécommunications*, vol. 66, no. 3-4, pp. 243–255, 2011.

[155] A. M. Hadjiantonis, A. Malatras, and G. Pavlou, "A context-aware, policy-based framework for the management of manets," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*. IEEE, 2006, pp. 23–34.

[156] A. Malatras, A. M. Hadjiantonis, and G. Pavlou, "Exploiting context-awareness for the autonomic management of mobile ad hoc networks," *Journal of Network and Systems Management*, vol. 15, no. 1, pp. 29–55, 2007.

[157] A. Malatras, G. Pavlou, and S. Sivavakeesar, "A programmable framework for the deployment of services and protocols in mobile ad hoc networks," *IEEE Transactions on Network and Service Management*, vol. 4, no. 3, pp. 12–24, 2007.

[158] R. Chadha, Yuu-Heng Cheng, J. Chiang, G. Levin, Shih-Wei Li, and A. Poylisher, "Policy-based mobile ad hoc network management for DRAMA," in *Proceedings of the Military Communications Conference (MILCOM)*, vol. 3. IEEE, 2004, pp. 1317–1323 Vol. 3.

[159] R. Chadha, H. Cheng, Y. Cheng, C. J. Chiang, A. Ghetie, G. Levin, and H. Tanna, "Policy-based mobile ad hoc network management," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*. IEEE, 2004, pp. 35–44.

[160] C. J. Chiang, R. Chadha, S. Newman, and R. Lo, "Towards automation of management and planning for future military tactical networks," in *Proceedings of the Military Communications Conference (MILCOM)*. IEEE, 2006.

[161] R. Chadha and C. J. Chiang, "DRAMA: distributed policy management for manets," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*. IEEE, 2008, pp. 235–237.

[162] C. J. Chiang, R. Chadha, G. Levin, Shihwei Li, Yuu-Heng Cheng, and A. Poylisher, "AMS: An adaptive middleware system for wireless ad hoc networks," in *Proceedings of the Military Communications Conference (MILCOM)*. IEEE, 2005.

[163] D. M. Chess, A. Segal, I. Whalley, and S. R. White, "Unity: Experiences with a prototype autonomic computing system," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE, 2004, pp. 140–147.

[164] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[165] M. Ayari, Z. Movahedi, G. Pujolle, and F. Kamoun, "ADMA: autonomous decentralized management architecture for manets: a simple self-configuring case study," in *Proceedings of the International Conference on Wireless Communications and Mobile Computing (IWCMC)*. ACM, 2009, pp. 132–137.

[166] M. Ayari, F. Kamoun, and G. Pujolle, "Towards autonomous mobile ad hoc networks: A distributed policy-based management approach," in *Proceedings of the International Conference on Wireless and Mobile Communications (ICWMC)*. IEEE, 2008, pp. 201–206.

[167] M. Ayari, F. Kamoun, and G. Pujolle, "Distributed policy management protocol for self-configuring mobile ad hoc networks," in *Advances in Ad Hoc Networking - Proceedings of the Annual Mediterranean Ad Hoc Networking Workshop*, ser. IFIP, vol. 265. Springer, 2008, pp. 73–84.

[168] M. Sifalakis, A. Louca, A. Mauthe, L. Peluso, and T. Zseby, "A functional composition framework for autonomic network architectures," in *Proceedings of the Network Operations and Management Symposium (NOMS)*. IEEE, 2008, pp. 328–334.

[169] G. Bouabene, C. Jelger, C. F. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ANA)," *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 1, pp. 4–14, 2010.

[170] C. Foley, S. Balasubramaniam, E. Power, M. P. de Leon, D. Botvich, D. Dudkowski, G. Nunzi, and C. Mingardi, "A framework for in-network management in heterogeneous future communication networks," in *Proceedings*

of the International Workshop on Modelling Autonomic Communications Environments (MACE), ser. Lecture Notes in Computer Science, vol. 5276. Springer, 2008, pp. 14–25.

[171] A. Gonzales et al., "In-network Management Design," 4WARD Project, Tech. Rep. Deliverable D4.3, May 2010.

[172] R. W. Thomas, D. H. Friend, L. A. DaSilva, and A. B. MacKenzie, "Cognitive networks: adaptation and learning to achieve end-to-end performance objectives," *IEEE Communications Magazine*, vol. 44, no. 12, pp. 51–57, 2006.

[173] ——, *Cognitive Networks*. Springer, 2007, pp. 17–41.

[174] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals," RFC 7575, Jun. 2015.

[175] ETSI, "ETSI GS AFI 002 V1.1.1 (2013-04) Autonomic network engineering for the self-managing Future Internet (AFI); Generic Autonomic Network Architecture (An Architectural Reference Model for Autonomic Networking, Cognitive Networking and Self-Management)," 2013.

[176] S. Jiang, B. E. Carpenter, and M. H. Behringer, "General Gap Analysis for Autonomic Networking," RFC 7576, Jun. 2015.

[177] N. Samaan and A. Karmouch, "Towards autonomic network management: an analysis of current and future research directions," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 3, pp. 22–36, 2009.

[178] M. H. Behringer, B. E. Carpenter, T. Eckert, L. Ciavaglia, and J. C. Nobre, "A Reference Model for Autonomic Networking," Internet Engineering Task Force, Internet-Draft draft-ietf-anima-reference-model-10, Nov. 2018, work in Progress.

[179] T. Eckert, M. H. Behringer, and S. Bjarnason, "An Autonomic Control Plane (ACP)," Internet Engineering Task Force, Internet-Draft draft-ietf-anima-autonomic-control-plane-28, Jul. 2020, work in Progress.

[180] Ranganai Chaparadza et al., "Requirements for a generic autonomic network architecture (GANA), suitable for standardizable autonomic behavior

specifications for diverse networking environments," *Annual review of communications*, vol. 61, pp. 165–194, 2008.

[181] R. Chaparadza, S. Papavassiliou, T. Kastrinogiannis, M. Vigoureux, E. Dotaro, A. Davy, K. Quinn, M. Wódczak, A. Toth, A. Liakopoulos, and M. Wilson, "Creating a viable evolution path towards self-managing future internet via a standardizable reference model for autonomic network engineering," in *Towards the Future Internet - A European Research Perspective*. IOS Press, 2009, pp. 136–147.

[182] R. Chaparadza, L. Ciavaglia, M. Wódczak, C. Chen, B. A. Lee, A. Liakopoulos, A. Zafeiropoulos, E. Mancini, U. Mulligan, A. Davy, K. Quinn, B. Radier, N. Alonistioti, A. Kousaridas, P. Demestichas, K. Tsagkaris, M. Vigoureux, L. Vreck, M. Wilson, and L. Ladid, "ETSI industry specification group on autonomic network engineering for the self-managing future internet (ETSI ISG AFI)," in *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, ser. Lecture Notes in Computer Science, vol. 5802. Springer, 2009, pp. 61–62.

[183] T. B. Meriem, R. Chaparadza, B. Radier, S. Soulhi, and A. P. López, "GANA -Generic Autonomic Networking Architecture," *ETSI Whitepaper*, 2016.

[184] J. Moy, "OSPF Version 2," RFC 2328, Apr. 1998.

[185] ETSI, "PoC Framework," https://intwiki.etsi.org/index.php?title=PoC_Framework, accessed: 2020-12-18.

[186] M. Pfannemüller, M. Breitbach, C. Krupitzer, C. Becker, and A. Schürr, "Enhancing a Communication System with Adaptive Behavior using REACT," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020.

[187] S. Tomforde and C. Müller-Schloer, "Incremental Design of Adaptive Systems," *Journal of Ambient Intelligence and Smart Environments*, vol. 6, no. 2, pp. 179–198, 2013.

[188] S. Mahdavi-Hezavehi, P. Avgeriou, D. Weyns, I. Mistrik, N. Ali, R. Kazman, G. John, and B. Schmerl, "A classification of current architecture-based approaches tackling uncertainty in self-adaptive systems with multiple

requirements," *Managing Trade-offs in Adaptable Software Architectures*, 2016.

[189] R. S. Hall, K. Pauls, S. McCulloch, and D. Savage, *OSGi in Action: Creating Modular Applications in Java*. Manning Publications Co., 2011.

[190] OSGi Alliance, "OSGi Specifications," https://www.osgi.org/developer/specifications/.

[191] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *Proceedings of the International Conference on Services Computing (SCC)*. IEEE, 2007, pp. 474–481.

[192] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed Applications Through Software Modularization," in *Middleware*, ser. Lecture Notes in Computer Science, vol. 4834. Springer, 2007, pp. 1–20.

[193] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, 2004.

[194] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[195] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Communications of the ACM*, vol. 42, no. 7, pp. 76–82, 1999.

[196] C. Surianarayanan, G. Ganapathy, and R. Pethuru, *Essentials of Microservices Architecture: Paradigms, Applications, and Techniques*, 2020.

[197] M. Henning, "The rise and fall of CORBA," *ACM Queue*, vol. 4, no. 5, pp. 28–34, 2006.

[198] S. Cheshire and M. Krochmal, "Multicast DNS," RFC 6762, Feb. 2013.

[199] ——, "DNS-Based Service Discovery," RFC 6763, Feb. 2013.

[200] O. B. Sezer, E. Dogdu, and A. M. Özbayoglu, "Context-aware computing, learning, and big data in internet of things: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 1–27, 2018.

[201] R. Cattell, "Scalable SQL and nosql data stores," *Special Interest Group on Management of Data (SIGMOD) Record*, vol. 39, no. 4, pp. 12–27, 2010.

[202] J. S. van der Veen, B. van der Waaij, and R. J. Meijer, "Sensor data storage performance: SQL or nosql, physical or virtual," in *Proceedings of the International Conference on Cloud Computing*. IEEE, 2012, pp. 431–438.

[203] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *Proceedings of the Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 2013, pp. 15–19.

[204] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing nosql mongodb to an SQL DB," in *ACM Southeast Regional Conference*. ACM, 2013, pp. 5:1–5:6.

[205] B. Schünemann, "V2X simulation runtime infrastructure VSimRTI: An assessment tool to design smart traffic management systems," *Computer Networks*, vol. 55, no. 14, pp. 3189–3198, 2011.

[206] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz, "SUMO – Simulation of Urban MObility: An Overview," in *Proceedings of the International Conference on Advances in System Simulation (SIMUL)*, 2011.

[207] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: Toward distraction-free pervasive computing," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22–31, 2002.

[208] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: context-aware reflective middleware system for mobile applications," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 929–945, 2003.

[209] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, 2002.

[210] S. VanSyckel, D. Schäfer, G. Schiele, and C. Becker, "Configuration management for proactive adaptation in pervasive environments," in *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2013, pp. 131–140.

[211] B. Y. Lim and A. K. Dey, "Toolkit to support intelligibility in context-aware applications," in *UbiComp*, ser. ACM International Conference Proceeding Series. ACM, 2010, pp. 13–22.

[212] M. Handte, G. Schiele, V. Majuntke, C. Becker, and P. J. Marrón, "3pc: System support for adaptive peer-to-peer pervasive computing," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 1, pp. 10:1–10:19, 2012.

[213] D. Dhungana, P. Grünbacher, and R. Rabiser, "Domain-Specific Adaptations of Product Line Variability Modeling," in *Proceedings of the International Federation for Information Processing (IFIP)*, vol. 244. Springer, 2007, pp. 238–251.

[214] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: "better than best-effort" computing," in *Proceedings of the International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2016, pp. 1–11.

[215] R. Kluge, M. Stein, G. Varró, A. Schürr, M. Hollick, and M. Mühlhäuser, "A systematic approach to constructing families of incremental topology control algorithms using graph transformation," *Software and Systems Modeling*, vol. 18, no. 1, pp. 279–319, 2019.

[216] G. A. Moreno, B. R. Schmerl, and D. Garlan, "SWIM: an exemplar for evaluation and comparison of self-adaptation approaches for web applications," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018, pp. 137–143.

[217] R. dos Reis Fontes and C. E. Rothenberg, "Mininet-WiFi: A Platform for Hybrid Physical-Virtual Software-Defined Wireless Networking Research," in *Proceedings of the SIGCOMM Conference*. ACM, 2016, pp. 607–608.

[218] M. Pfannemüller, "A dynamic software product line approach for planning and execution of reconfigurations in self-adaptive systems," University of Mannheim, Tech. Rep., 2017, https://ub-madoc.bib.uni-mannheim.de/41782.

[219] J. Petke, *Bridging Constraint Satisfaction and Boolean Satisfiability*, ser. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015.

[220] H. Bennaceur, "A comparison between SAT and CSP techniques," *Constraints*, vol. 9, no. 2, pp. 123–138, 2004.

[221] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach, Third International Edition.* Pearson Education, 2010.

[222] Satcompetition.org, "CNF File Format," http://www.satcompetition.org/2009/format-benchmarks2009.html, 2009, accessed: 2020-12-18.

[223] D. L. Berre and A. Parrain, "The sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–6, 2010.

[224] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker, "Fault-avoidance strategies for context-aware schedulers in pervasive computing systems," in *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom).* IEEE, 2017, pp. 79–88.

[225] M. Weckesser, R. Kluge, M. Pfannemüller, M. Matthé, A. Schürr, and C. Becker, "Optimal Reconfiguration of Dynamic Software Product Lines Based on Performance-Influence Models," in *Proceedings of the Software Product Line Conference (SPLC)*, 2018.

[226] H. P. Williams, *Model Building in Mathematical Programming.* John Wiley & Sons, 2013.

[227] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/SIGSOFT, FSE).* ACM, 2015, pp. 284–294.

[228] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518.

[229] T. Schnabel, M. Weckesser, R. Kluge, M. Lochau, and A. Schürr, "Cardygan: Tool support for cardinality-based feature models," in *VaMoS.* ACM, 2016, pp. 33–40.

[230] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the International Conference Companion on Object-Oriented Programming Systems Languages and Applications (OOPSLA).* ACM, 2010, pp. 307–309.

[231] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL conqueror: Toward optimization of non-functional properties

in software product lines," *Software Quality Journal*, vol. 20, no. 3-4, pp. 487–517, 2012.

[232] P. Santi, "Topology control in wireless ad hoc and sensor networks," *ACM Computing Surveys*, vol. 37, no. 2, pp. 164–194, 2005.

[233] B. Richerzhagen, D. Stingl, J. Rückert, and R. Steinmetz, "Simonstrator: simulation and prototyping platform for distributed mobile applications," in *Proceedings of the International Conference on Simulation Tools and Techniques (SimuTools)*. ICST/ACM, 2015, pp. 99–108.

[234] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557–581, 1999.

[235] I. J. Lustig and J. Puget, "Program does not equal program: Constraint programming and its relationship to mathematical programming," *Interfaces*, vol. 31, no. 6, pp. 29–53, 2001.

[236] N. Chen. (2006) Convention over configuration. http://softwareengineering.vazexqi.com/files/pattern.html.

[237] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: unifying class and feature modeling," *Software and System Modeling*, vol. 15, no. 3, pp. 811–845, 2016.

[238] M. Antkiewicz, K. Bak, K. Czarnecki, Z. Diskin, D. Zayan, and A. Wasowski, "Example-Driven Modeling using Clafer," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, vol. 1104. CEUR-WS.org, 2013, pp. 32–41.

[239] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled," in *Proceedings of the International Conference on Software Language Engineering (SLE)*, ser. Lecture Notes in Computer Science. Springer, 2010, pp. 102–122.

[240] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Springer, 2018, pp. 305–343.

[241] K. Bak, "Modeling and analysis of software product line variability in Clafer," Ph.D. dissertation, University of Waterloo, Ontario, Canada, 2013.

[242] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*, http://www.choco-solver.org, TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[243] J. Cámara, D. Garlan, B. R. Schmerl, and A. Pandey, "Optimal planning for architecture-based self-adaptation via model checking of stochastic games," in *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, 2015, pp. 428–435.

[244] G. A. Moreno, J. Cámara, D. Garlan, and B. R. Schmerl, "Flexible and efficient decision-making for proactive latency-aware self-adaptation," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 13, no. 1, pp. 3:1–3:36, 2018.

[245] J. Cámara, G. A. Moreno, and D. Garlan, "Reasoning about human participation in self-adaptive systems," in *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2015, pp. 146–156.

[246] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the International Conference on Simulation Tools and Techniques (SimuTools)*. ICST/ACM, 2008, p. 60.

[247] T. Vogel, "Model-Driven Engineering of Self-Adaptive Software," Ph.D. dissertation, University of Potsdam, 2018.

[248] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[249] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. M. Parulkar, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the Workshop on Hot topics in Software-Defined Networking (HotSDN)*. ACM, 2014, pp. 1–6.

[250] J. Dilley, "Web server workload characterization," *HP Laboratories Technical Report*, vol. 24, no. 96-160, pp. 1–16, dec 1996.

[251] M. Pfannemüller, M. Weckesser, R. Kluge, J. Edinger, M. Luthra, R. Klose, C. Becker, and A. Schürr, "CoalaViz: Supporting Traceability of Adaptation Decisions in Pervasive Communication Systems," in *Proceedings of the*

International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). IEEE, 2019, pp. 590–595.

[252] M. Pfannemüller, J. Edinger, M. Weckesser, R. Kluge, M. Luthra, R. Klose, C. Becker, and A. Schürr, "Demo: Visualizing adaptation decisions in pervasive communication systems," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2019, pp. 335–337.

[253] C. Buschmann, D. Pfisterer, S. Fischer, S. P. Fekete, and A. Kröller, "Spyglass: a wireless sensor network visualizer," *Special Interest Group on Embedded Systems (SIGBED) Record*, vol. 2, no. 1, pp. 1–6, 2005.

[254] Y. Yang, P. Xia, L. Huang, Q. Zhou, Y. Xu, and X. Li, "Snamp: A multi-sniffer and multi-view visualization platform for wireless sensor networks," in *Proceedings of the Conference on Industrial Electronics and Applications*. IEEE, 2006.

[255] Y. Hu, D. Li, X. He, T. Sun, and Y. Han, "The implementation of wireless sensor network visualization platform based on wetland monitoring," in *Proceedings of the International Conference on Intelligent Networks and Intelligent Systems*. IEEE, 2009, pp. 224–227.

[256] D. Estrin, M. Handley, J. S. Heidemann, S. McCanne, Y. Xu, and H. Yu, "Network visualization with nam, the VINT network animator," *IEEE Computer*, vol. 33, no. 11, pp. 63–68, 2000.

[257] Y. Wang, "Topology control for wireless sensor networks," in *Wireless sensor networks and applications*. Springer, 2008, pp. 113–147.

[258] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "TCEP: adapting to dynamic user environments by enabling transitions between operator placement mechanisms," in *Proceedings of the International Conference on Distributed and Event-based Systems (DEBS)*. ACM, 2018, pp. 136–147.

[259] J. Nielsen, *Usability engineering*. AP Professional, 1993.

[260] H. Hagras, "Toward human-understandable, explainable AI," *IEEE Computer*, vol. 51, no. 9, pp. 28–36, 2018.

[261] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. P. Chau, "Activis: Visual exploration of industry-scale deep neural network models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 88–97, 2018.

[262] M. Pfannemüller, M. Breitbach, and C. Becker, "EnTrace: Achieving Enhanced Traceability in Self-Aware Computing Systems," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020.

[263] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *Proceedings of the Conference on the Future of Software Engineering (FOSE)*. ACM, 2014, pp. 55–69.

[264] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," Tech. Rep., 2017.

[265] E. R. Tufte, *The visual display of quantitative information*. Graphics Press, 2001, vol. 2.

[266] I. Vessey, "Cognitive fit: A theory-based analysis of the graphs versus tables literature," *Decision Sciences*, vol. 22, no. 2, pp. 219–240, 1991.

[267] G. P. Ellis and A. J. Dix, "A taxonomy of clutter reduction for information visualisation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1216–1223, 2007.

[268] R. Kosara and J. D. Mackinlay, "Storytelling: The next step for visualization," *IEEE Computer*, vol. 46, no. 5, pp. 44–50, 2013.

[269] M. Borkin, A. A. Vo, Z. Bylinskii, P. Isola, S. Sunkavalli, A. Oliva, and H. Pfister, "What makes a visualization memorable?" *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2306–2315, 2013.

[270] M. Tory and T. Möller, "Human factors in visualization research," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 1, pp. 72–84, 2004.

[271] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *VL*. IEEE, 1996, pp. 336–343.

[272] J. Heer and G. G. Robertson, "Animated transitions in statistical data graphics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1240–1247, 2007.

[273] A. Sarikaya, M. Correll, L. Bartram, M. Tory, and D. Fisher, "What do we talk about when we talk about dashboards?" *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 682–692, 2019.

[274] J. Wang, L. Gou, W. Zhang, H. Yang, and H. Shen, "Deepvid: Deep visual interpretation and diagnosis for image classifiers via knowledge distillation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 6, pp. 2168–2180, 2019.

[275] M. Kahng, N. Thorat, D. H. P. Chau, F. B. Viégas, and M. Wattenberg, "GAN lab: Understanding complex deep generative models using interactive visual experimentation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 310–320, 2019.

[276] B. C. Kwon, M. Choi, J. T. Kim, E. Choi, Y. B. Kim, S. Kwon, J. Sun, and J. Choo, "Retainvis: Visual analytics with interpretable and interactive recurrent neural networks on electronic medical records," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 299–309, 2019.

[277] S. Chung, S. Suh, C. Park, K. Kang, J. Choo, and B. C. Kwon, "ReVACNN: Real-Time visual analytics for convolutional neural network," in *ACM SIGKDD Workshop on Interactive Data Exploration and Analytics (IDEA)*, 2016.

[278] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 1–12, 2018.

[279] H. Strobelt, S. Gehrmann, M. Behrisch, A. Perer, H. Pfister, and A. M. Rush, "Seq2seq-vis: A visual debugging tool for sequence-to-sequence models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 353–363, 2019.

[280] D. A. Keim, "Information visualization and visual data mining," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 1–8, 2002.

[281] A. Weller, "Challenges for transparency," *CoRR*, vol. abs/1708.01870, 2017.

[282] N. Elmqvist and J. Fekete, "Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 3, pp. 439–454, 2010.

[283] A. I. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[284] P. Dolog and W. Nejdl, "Using uml-based feature models and UML collaboration diagrams to information modelling for web-based applications," in *Proceedings of the International Conference on the Unified Modeling Language (UML)*, ser. Lecture Notes in Computer Science, vol. 3273. Springer, 2004, pp. 425–439.

[285] V. Vranic and J. Snirc, "Integrating feature modeling into UML," in *Proceedings of NODe/GSEM*, ser. Lecture Notes in Informatics (LNI), vol. P-88. GI, 2006, pp. 3–15.

[286] Y. Wu, F. Hernandez, F. Ortega, P. J. Clarke, and R. France, "Measuring the effort for creating and using domain-specific models," in *Proceedings of the Workshop on Domain-Specific Modeling (DSM)*. ACM, 2010.

[287] Jingqiu Shao and Yingxu Wang, "A new measure of software complexity based on cognitive weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69–74, 2003.

[288] M. W. v. Someren, Y. F. Barnard, and J. A. Sandberg, *The think aloud method : a practical guide to modelling cognitive processes*. Academic Press, 1994.

[289] S. Herrnleben, M. Pfannemüller, C. Krupitzer, S. Kounev, M. Segata, F. Fastnacht, and M. Nigmann, "Towards adaptive car-to-cloud communication," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2019, pp. 119–124.

Appendix

A. Appendix to SAT-Based Feedback Loop

This chapter contains additional material of the SAT-based feedback loop instance presented in Section 7.1.

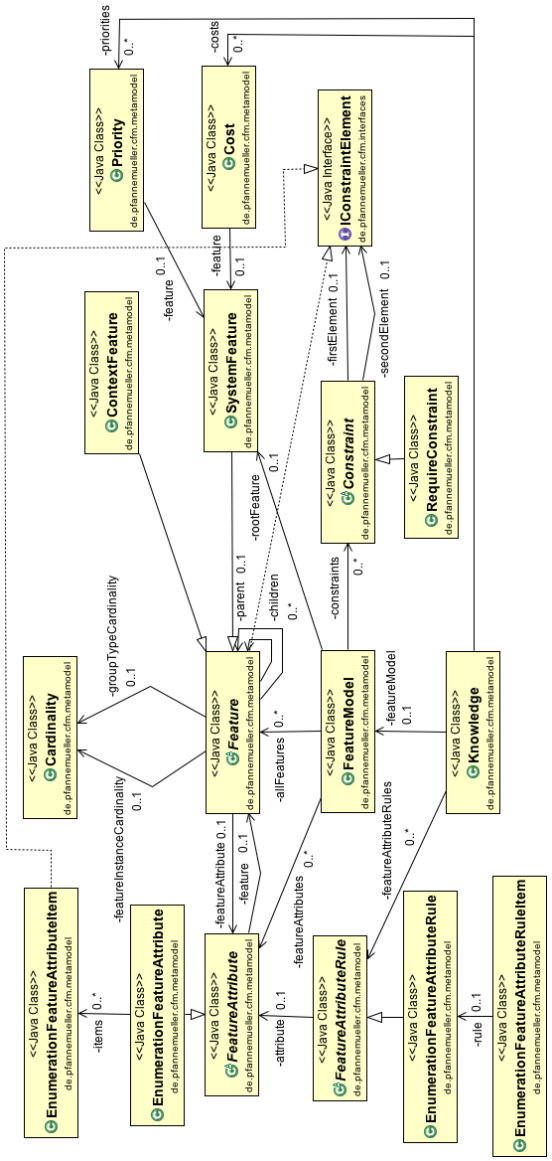


Figure A.1.: UML class diagram of the knowledge meta-model used in the SAT-based REACT Loop [104, 218].

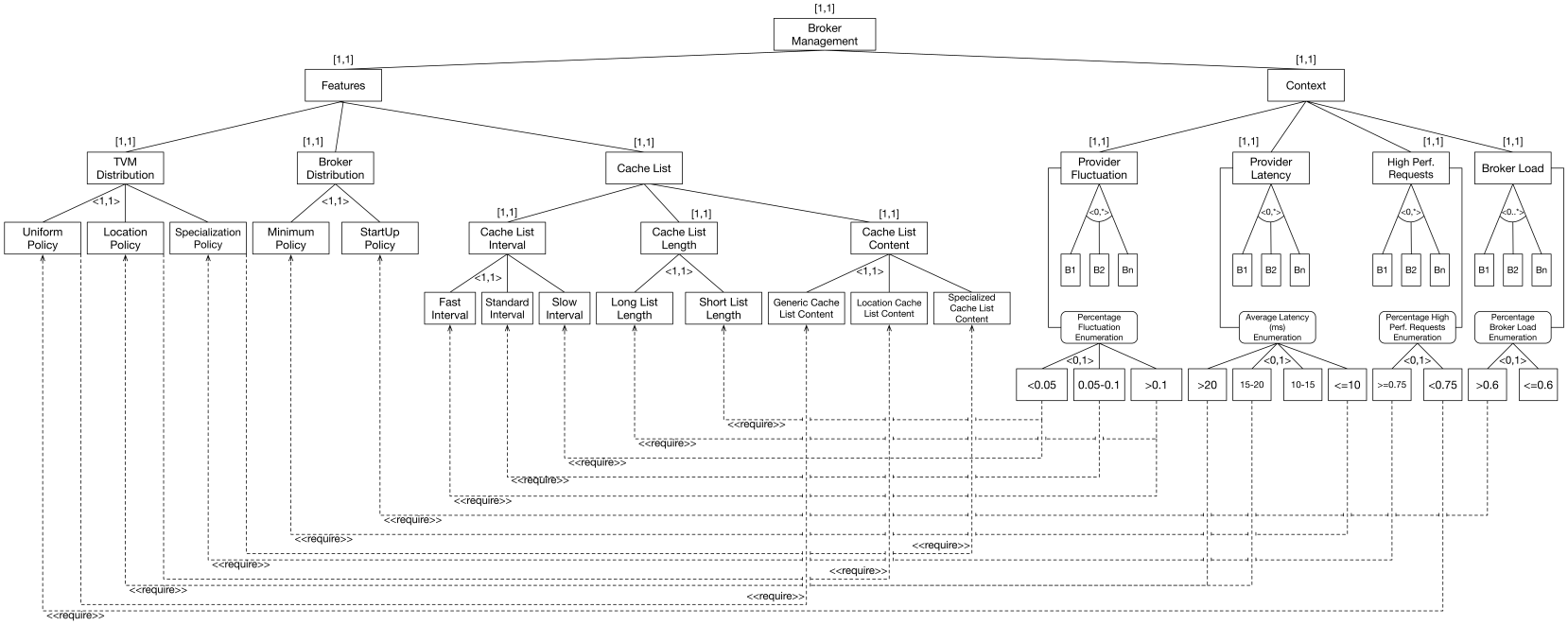


Figure A.2.: CFM of the broker management system managing the Tasklet system [104, 218].

B. Appendix to Comparison of Feedback Loops

This chapter contains the full CardyGAn and Clafer specifications, which have been used in the comparison of the MILP- and CP-based REACT Loops presented in Section 7.4.

```
1..1 root gt=1..* gi=1..* {
        1..1 fsSystem gt=xor gi=xor {
                        0..1 DimmerSetter_Plus
                        0..1 DimmerSetter_Minus
                        0..1 ServerStarter_1
                        0..1 ServerRemover_1
                        0..1 ServerStarter_2
                        0..1 ServerRemover_2 }
        1..1 fcContext gt=1..* gi=1..* {
                1..1 responseTime gt=xor gi=xor {
                        0..1 rt0_25
                        0..1 rt26_30
                        0..1 rt31_75
                        0..1 rt76_100
                        0..1 rt101_150
                        0..1 rt151 }
        }
}
[ 1..1 rt0_25 require 1..1 ServerRemover_2
        1..1 rt26_30 require 1..1 ServerRemover_1
        1..1 rt31_75 require 1..1 DimmerSetter_Plus
        1..1 rt76_100 require 1..1 DimmerSetter_Minus
        1..1 rt101_150 require 1..1 ServerStarter_1
        1..1 rt151 require 1..1 ServerStarter_2 ]
```

Listing B.1: CardygGAn representation of the SAT specification.

```
1..1 root gt=or gi=or {
    1..1 fsSystem gt=xor gi=xor {
        0..1 Dimmer gt=or gi=or {
```

```
4             dimmerValue:Integer 0..100
5         }
6         0..1 ServerStarter gt=or gi=or {
7                 numberToAdd:Integer 1..3
8         }
9         0..1 ServerRemover gt=or gi=or {
10                numberToRemove:Integer 1..3
11        }
12    }
13    1..1 fcContext gt=or gi=or {
14        currentDimmerValue:Integer 0..100
15        responseTime:Integer 1..10000
16    }
17 }
18 [ ((responseTime <= 15) => numberToRemove = 2) &&
19        (((responseTime > 15) && (responseTime <= 30)) => numberToRemove =
              1) &&
20        (((responseTime > 30) && (responseTime <= 75) && (
              currentDimmerValue + 25 <= 100)) => dimmerValue =
              currentDimmerValue + 25) &&
21        (((responseTime > 30) && (responseTime <= 75) && (
              currentDimmerValue + 25 > 100)) => dimmerValue = 100) &&
22        (((responseTime > 75) && (responseTime <= 100) && (
              currentDimmerValue + -25 >= 0)) => dimmerValue =
              currentDimmerValue + -25) &&
23        (((responseTime > 75) && (responseTime <= 100) && (
              currentDimmerValue + -25 < 0)) => dimmerValue = 0) &&
24        (((responseTime > 100) && (responseTime <= 150)) => numberToAdd =
              1) &&
25        ((responseTime > 150) => numberToAdd = 2) ]
```

Listing B.2: CardygGAn representation of the simplified MILP specification.

```
1  [if responseTime < 15
2    then RemoveServer.number = 2 && one RemoveServer
3  else
4    if responseTime > 15 && responseTime <= 30
5      then RemoveServer.number = 1 && one RemoveServer
6    else
7      if responseTime > 30 && responseTime <= 75
8        then
9          if dimmer + 25 <= 100
10           then SetDimmer.dimmer = dimmer + 25 && one SetDimmer
11         else
12           SetDimmer.dimmer = 100 && one SetDimmer
13     else
14       if responseTime > 75 && responseTime <= 100
15         then
16           if dimmer - 25 >= 0
17             then SetDimmer.dimmer = dimmer - 25 && one SetDimmer
18           else
19             SetDimmer.dimmer = 0 && one SetDimmer
20       else
21         if responseTime > 100 && responseTime <= 150
22           then AddServer.number = 1 && one AddServer
23         else
24           if responseTime > 150
25             then AddServer.number = 2 && one AddServer
26           else one NoAdaptation
27 ]
28
29 abstract Context 1..1
30   dimmer -> integer 1..1
31   servers -> integer 1..1
32   activeServers -> integer 1..1
33   responseTime -> integer 1..1
34   maxServers -> integer 1..1
35   totalUtilization -> integer 1..1
36
37 AddServer 0..1
38   number -> integer 1..1
39 RemoveServer 0..1
40   number -> integer 1..1
41 SetDimmer 0..1
```

```
42   dimmer -> integer 1..1
43 NoAdaptation 0..1
```

Listing B.3: Clafer representation of the simplified specification.

```
1..1 root gt=or gi=or {
    1..1 fsSystem gt=or gi=or {
        0..1 SetDimmer gt=or gi=or {
            dimmerValue:Integer 1..100
        }
        0..1 Server gt=xor gi=xor{
                0..1 AddServer gt=or gi=or {
                        numberToAdd:Integer 1..3
                }
                0..1 RemoveServer gt=or gi=or {
                        numberToRemove:Integer 1..3
                }
        }
        0..1 NoAdaptation
    }
    1..1 fcContext gt=or gi=or {
        currentDimmerValue:Integer 0..100
        servers:Integer 1..3
        activeServers:Integer 1..3
        responseTime:Integer 1..100000
        maxServers:Integer 1..3
        totalUtilization:Integer 1..10000
    }

    // Helping variables
    revenue:Real 0..100
    revenue2:Real 0..1000
    nDimmer:Integer 0..100
    nServers:Integer 1..3
    capacity:Integer 0..1000
    rFactor:Real 0..10000
}
[ fcContext => capacity = (100 * activeServers + (-totalUtilization))
        (responseTime <= 75) => rFactor = (0.333 * responseTime)
        (responseTime > 75 && responseTime <= 200) => rFactor = 0.4 *
            responseTime
        (responseTime > 200) => rFactor = 0.11 * responseTime
        100 + -rFactor > 0 => revenue = 100.0 + -rFactor
        100 + -rFactor <= 0 => revenue = 0
        fcContext => revenue2 = 3 * revenue
        (responseTime <= 30) => nServers = 1
```

```
41          (responseTime > 30 && responseTime <= 75) => nServers = 2
42          (responseTime > 75) => nServers = 3
43
44          revenue2 >= 300 => revenue2 = 299
45          3 * nDimmer > revenue2
46
47      (((responseTime > 75) && !(servers > activeServers) && (servers <
            maxServers)) => numberToAdd = 1 && dimmerValue = nDimmer)
48          (((responseTime > 75) && !(!(servers > activeServers) && servers <
                maxServers) && (dimmer > 0)) => dimmerValue = nDimmer)
49          (((responseTime > 75) && !(!(servers > activeServers) && servers <
                maxServers) && (dimmer <= 0)) => NoAdaptation)
50
51          (((responseTime < 75) && (capacity > 100) && (currentDimmerValue
                <= 90)) => dimmerValue = nDimmer)
52          (((responseTime < 75) && (capacity > 130) && (currentDimmerValue >
                90) && !(servers > activeServers) && (servers > 1)) =>
                numberToRemove = 1 && dimmerValue = nDimmer)
53          (((responseTime < 75) && (capacity > 100) && (currentDimmerValue >
                90) && !(!(servers > activeServers) && servers > 1)) =>
                NoAdaptation)
54          (((responseTime < 75) && (capacity <= 100)) => dimmerValue =
                nDimmer) ]
```

Listing B.4: CardygGAn representation of the full MILP specification.

```
1  [ if (100 - (rFactor * responseTime) / 100) > 0
2    then nDimmer = (100 - (rFactor * responseTime) / 100)
3  else nDimmer = 0 ]
4
5  [ if Context.responseTime.dref > 75
6    then
7      if !(Context.servers.dref > Context.activeServers.dref) && (Context.
           servers.dref < Context.maxServers.dref)
8        then AddServer.number = 1 && one AddServer && SetDimmer.dimmer =
             Variables.nDimmer.dref && one SetDimmer
9      else
10       if Context.dimmer.dref > 0
11         then SetDimmer.dimmer = nDimmer.dref && one SetDimmer
12       else one NoAdaptation
13   else
14     if responseTime < 75
15       then
16         if capacity > 100
17           then
18             if Context.dimmer.dref > 90
19               then
20                 if !(!(Context.servers.dref > Context.activeServers.dref)
                       && (Context.servers.dref > 1))
21                   then one NoAdaptation
22                 else
23                   if Variables.capacity.dref > 130
24                     then RemoveServer.number.dref = 1 && one RemoveServer &&
                            SetDimmer.dimmer = Variables.nDimmer.dref && one
                           SetDimmer
25                   else one NoAdaptation
26             else SetDimmer.dimmer = Variables.nDimmer.dref && one SetDimmer
27         else SetDimmer.dimmer = Variables.nDimmer.dref && one SetDimmer
28     else one NoAdaptation ]
29
30 Variables 1..1
31   rFactor -> integer 1..1
32   [ if responseTime <= 75
33       then rFactor = 33
34     else
35       if (responseTime <= 200)
36         then rFactor = 40
```

```
37       else rFactor = 11]
38
39   nDimmer -> integer 1..1
40   [ nDimmer >= 0 && nDimmer <= 100 ]
41
42   capacity -> integer 1..1
43   [ if (activeServers * 100 - totalUtilization) <= 256
44       then capacity = activeServers * 100 - totalUtilization
45     else capacity = 256 ]
46
47   nServer -> integer 1..1
48   [ if responseTime <= 30
49       then nServer = 1
50     else
51       if responseTime <= 75
52         then nServer = 2
53       else nServer = 3 ]
54
55 abstract Context 1..1
56   dimmer -> integer 1..1
57   servers -> integer 1..1
58   activeServers -> integer 1..1
59   responseTime -> integer 1..1
60   maxServers -> integer 1..1
61   totalUtilization -> integer 1..1
62
63 AddServer 0..1
64   number -> integer 1..1
65 RemoveServer 0..1
66   number -> integer 1..1
67 SetDimmer 0..1
68   dimmer -> integer 1..1
69 NoAdaptation 0..1
```

Listing B.5: Clafer representation of the full specification.

Publications Contained in This Thesis

[**289**] S. Herrnleben, M. Pfannemüller, C. Krupitzer, S. Kounev, M. Segata, F. Fastnacht, and M. Nigmann, "Towards adaptive car-to-cloud communication," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2019, pp. 119–124.
My Contribution: Mapping of the use case to the design for a SAS (Section VI).

[**11**] M. Pfannemüller, M. Breitbach, C. Krupitzer, M. Weckesser, C. Becker, B. Schmerl, and A. Schürr, "REACT: A Model-Based Runtime Environment for Adapting Communication Systems," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020.
My Contribution: Responsible author of the paper.

[**186**] M. Pfannemüller, M. Breitbach, C. Krupitzer, C. Becker, and A. Schürr, "Enhancing a Communication System with Adaptive Behavior using REACT," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020.
My Contribution: Responsible author of the paper.

[**262**] M. Pfannemüller, M. Breitbach, and C. Becker, "EnTrace: Achieving Enhanced Traceability in Self-Aware Computing Systems," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020.
My Contribution: Responsible author of the paper.

[**251**] M. Pfannemüller, M. Weckesser, R. Kluge, J. Edinger, M. Luthra, R. Klose, C. Becker, and A. Schürr, "CoalaViz: Supporting Traceability of Adaptation Decisions in Pervasive Communication Systems," in *Proceedings of the International Conference on Pervasive Computing and Communications*

Workshops (PerCom Workshops). IEEE, 2019, pp. 590–595.
My Contribution: Responsible author of the paper.

[252] M. Pfannemüller, J. Edinger, M. Weckesser, R. Kluge, M. Luthra, R. Klose, C. Becker, and A. Schürr, "Demo: Visualizing adaptation decisions in pervasive communication systems," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2019, pp. 335–337.
My Contribution: Responsible author of the paper.

[87] M. Pfannemüller, "Self-adaptive middleware for model-based network adaptations," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2018, pp. 462–463.
My Contribution: Responsible author of the paper.

[104] M. Pfannemüller, C. Krupitzer, M. Weckesser, and C. Becker, "A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2017, pp. 247–254.
My Contribution: Responsible author of the paper.

[218] M. Pfannemüller, "A dynamic software product line approach for planning and execution of reconfigurations in self-adaptive systems," University of Mannheim, Tech. Rep., 2017, `https://ub-madoc.bib.uni-mannheim.de/41782`.
My Contribution: Master thesis.

[225] M. Weckesser, R. Kluge, M. Pfannemüller, M. Matthé, A. Schürr, and C. Becker, "Optimal Reconfiguration of Dynamic Software Product Lines Based on Performance-Influence Models," in *Proceedings of the Software Product Line Conference (SPLC)*, 2018.
My Contribution: Design and implementation of the adaptation logic (Section 3).

Lebenslauf

Seit 02/2017	Akademischer Mitarbeiter Lehrstuhl für Wirtschaftsinformatik II Universität Mannheim
08/2013 – 01/2014	Informatik Linköpings Universitet
08/2011 – 01/2017	Bachelor & Master of Science Wirtschaftsinformatik Universität Mannheim