REACT-ION: A Model-Based Runtime Environment for Situation-Aware Adaptations

MARTIN PFANNEMÜLLER and MARTIN BREITBACH, Universität Mannheim, Germany
MARKUS WECKESSER, Technische Universität Darmstadt, Germany
CHRISTIAN BECKER, Universität Mannheim, Germany
BRADLEY SCHMERL, Carnegie Mellon University, USA
ANDY SCHÜRR, Technische Universität Darmstadt, Germany
CHRISTIAN KRUPITZER, Universität Hohenheim, Germany

Trends such as the Internet of Things or edge computing lead to a growing number of networked devices. Hence, it is becoming increasingly important to manage communication systems at runtime. Adding self-adaptive capabilities is one approach to reduce administrative effort and cope with changing execution contexts. Existing frameworks for building self-adaptive software can help to reduce development effort in general. Yet, they are neither tailored towards the use in communication systems nor easily usable without profound knowledge in self-adaptive systems development. Accordingly, in previous work we proposed REACT, a reusable, model-based runtime environment to complement communication systems with adaptive behavior. It addresses the heterogeneity and distribution aspects of networks and reduces development effort. We showed the effectiveness and efficiency of our prototype in an experimental evaluation based on two distinct use cases from the communication systems domain: cloud resource management and software-defined networking. In this work, we propose REACT-ION— an extension of REACT for situation awareness. REACT-ION provides context management capabilities that enable self-improvement and prediction for proactive adaptations. These extensions can be used to optimize adaptation decisions at runtime based on the current situation. Therefore, REACT-ION is able to cope with uncertainty and situations that were not foreseeable at design time. We show and evaluate in two extensive case studies how REACT-ION's situation awareness enables proactive adaptation and self-improvement.

CCS Concepts: • **Computer systems organization → Self-organizing autonomic computing**; • **Software and its engineering → Middleware**; *System modeling languages*; *Unified Modeling Language (UML).*

Additional Key Words and Phrases: self-adaptive systems, model-based, runtime environment, framework

ACM Reference Format:
Martin Pfannemüller, Martin Breitbach, Markus Weckesser, Christian Becker, Bradley Schmerl, Andy Schürr, and Christian Krupitzer. 2021. REACT-ION: A Model-Based Runtime Environment for Situation-Aware Adaptations. *ACM Trans. Autonom. Adapt. Syst.* 37, 4, Article 111 (August 2021), 24 pages. https://doi.org/10.1145/1122445.1122456

Authors' addresses: Martin Pfannemüller, martin.pfannemueller@uni-mannheim.de; Martin Breitbach, martin.breitbach@uni-mannheim.de, Universität Mannheim, Schloss, Mannheim, Germany, 68131; Markus Weckesser, Technische Universität Darmstadt, Karolinenplatz 5, Darmstadt, Germany, 64289, markus.weckesser@es.tu-darmstadt.de; Christian Becker, Universität Mannheim, Schloss, Mannheim, Germany, 68131, christian.becker@uni-mannheim.de; Bradley Schmerl, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA, 15213, schmerl@cs.cmu.edu; Andy Schürr, Technische Universität Darmstadt, Karolinenplatz 5, Darmstadt, Germany, 64289, andy.schuerr@es.tu-darmstadt.de; Christian Krupitzer, Universität Hohenheim, Fruwirthstr. 21, Stuttgart, Germany, 70599, christian.krupitzer@uni-hohenheim.de.

1556-4665/2021/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

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 adapt a system at runtime according to changes in the execution context [48]. A self-adaptive system consists of the managed target system and an adaptation logic managing the target system. Adding self-adaptive capabilities to communication systems—computer networks as well as supporting structures such as overlays or middleware—is a major research focus. For instance, self-adaptive applications in the software-defined networking (SDN) domain can help to reduce management effort and improve the network's performance [18]. SDN provides possibilities to monitor and reconfigure a network by specifying selectors for packets and corresponding actions. An adaptation logic may use these capabilities for reconfiguring the packet flows at runtime.

Making such communication systems self-adaptive, however, is a challenging task for domain experts, i.e., communication systems developers. First, the distributed nature of those systems requires the collection of monitoring information from multiple hosts and the adaptation of distributed components. Second, communication systems consist of heterogeneous components, e.g., developed in different programming languages. Third, domain experts typically lack knowledge about the development of self-adaptive systems.

Instead of manually integrating self-adaptivity, the domain expert may rely on frameworks or tools. While approaches such as Rainbow [31], SASSY [44], or MUSIC [57] are suitable for the general purpose of engineering self-adaptive systems, they are neither tailored to communication systems, nor support the domain expert adequately in these use cases. To the best of our knowledge, no existing approach supports multiple programming languages, enables decentralized adaptation logics with distributed deployments, and is available as an easy-to-use open source project for domain experts.

Motivated by these observations, we proposed REACT, a **R**untime **E**nvironment for **A**dapting **C**ommunication Sys**T**ems[1] in [51]. REACT supports domain experts in specifying adaptation behavior in a model-based fashion with Clafer [7] and UML. By implementing language-independent interfaces and selecting deployment options, REACT connects to the target system and automatically deploys its integrated feedback loop. Thus, it is applicable to legacy systems as well. REACT is lightweight and easy-to-use while satisfying the specific requirements of adaptive communication systems. To bridge the prevailing gap between self-adaptive systems research and practice [22, 71], we implemented REACT, made it available as an open source project[2], and guided domain experts with a well-defined development process. We evaluated REACT by (i) comparing it with the state-of-the-art Rainbow framework in a cloud resource management scenario and (ii) applying it in a real-world use case from the SDN domain.

In this paper, we present REACT-ION— an extension of REACT that additionally integrates features for providing situation-awareness [25] as demanded for self-adaptive systems in [29]. REACT-ION contains a context management module, which provides the foundation for situation awareness. Additionally, situation awareness includes the concepts of being able to predict the future and derive higher-level context called situations [25]. Hence, situation awareness can especially be applied in combination with proactive adaptations and for adapting the adaptation logic (known as self-improvement [39]) for tackling uncertainty [27].

The remainder of this paper is structured as follows. Section 2 reviews related work. Section 3 briefly outlines REACT's architecture. This section also introduces how REACT's implementation

[1]This paper is an extended version of previous work appeared in the Proceedings of the 1st IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2020)
[2]Available here: https://github.com/martinpfannemueller/REACT.

supports self-adaptivity. Section 4 proposes REACT-ION. It first presents the concept of situation awareness. Based on this concept, the section describes REACT-ION's context management module. This includes improved interfaces enabling external systems outside of REACT-ION to use the context information for situation awareness. Following, the same section includes the application of proactive adaptation with REACT-ION, as well as the presentation of a possibility for compositional adaptation of the adaptation logic using multiple feedback loop instances as foundation for self-improving a REACT-based adaptation logic. Section 5 first briefly summarizes the previous evaluations presented in [51], which evaluated our approach in distinct use cases and compared REACT to the state of the art. Then, it presents evaluations considering proactivity and compositional self-improvement. Finally, Section 6 summarizes our findings and outlines future work.

2 RELATED WORK

Engineering of self-adaptive systems is a prominent research area with a large body of excellent related work that we can build upon. We review the research landscape in [37]. Several related approaches perform adaptations based on architectural models (e.g., [28, 48, 61]) or specify architecture definition languages for self-adaptive systems (e.g., [19, 24, 42]). Model-based engineering approaches such as [10, 30, 47, 52] often use DSPLs with feature models. The models@run.time research proposes to use runtime models that represent the system and environment for reasoning [8, 12]. All of the aforementioned approaches, however, do not offer an implementation explicitly designed to be used by others. Since we design an approach that aims at high applicability for practitioners and fellow researchers, we focus on implementation aspects of related work in the remainder of this section, as summarized in Table 1.

First, an approach that optimally assists domain experts should support all self-* properties [36]—self-configuration, self-optimization, self-healing, and self-protection—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 models, goals, or utilities makes the approach useful for domain experts 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 these requirements. However, FESAS [38] and HAFLoop [74] for instance, provide excellent support with reusable MAPE components, but do not integrate a decision engine.

We aim to support the domain expert during the development process. Especially in the heterogeneous communication systems landscape, an approach is easy to use if it supports multiple programming languages such as the approach by Malek *et al.* [56]. 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 [31] framework allow connecting the target system easily to the adaptation logic, which is especially important for legacy systems. Rainbow, however, belongs to the approaches [16, 17, 31, 44, 63] that do not specify an easy-to-follow development process.

We argue that an approach that is suitable for large and heterogeneous communication systems must support decentralized control with multiple feedback loops [21]. 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 [39], 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

	Capabilities				Dev. Sup.			Depl.		Eval.	
Author/*System*	**All Self-* Properties**	**Provides Decision Eng.**	**Supports ex. System**	**Use Case Independent**	**Multi Language Support**	**Predefined Interfaces**	**Specified Dev. Process**	**Decentralized Loop**	**Runtime Modifications**	**Code Available**	**Comparison Available**
ActivForms [72]	●	●	●	●			●		●	●	
Cetina [17]		●	●								
EUREMA [68, 69]	●		●	●					●		
FESAS [38]	●		●	●		●	●	●	●	●	
Genie [9]	●	●	●				●		●		
GRAF [1]	●	●	●	●		●	●		●		
HAFLoop [74]	●		●	●		●	●	●	●	●	
KX [49]	●	●	●	●				●	●		
Malek [56]	●	●		●	●	●	●		●		
MOSES [16]		●	●								
MUSIC [57]	●	●		●		●	●		●		
Preisler [53]	●	●	●	●			●				
Rainbow [20, 31]	●	●	●	●		●				●	●
REFRACT [63]		●	●						●		
SASSY [44]	●	●	●	●					●		
StarMX [4]	●		●	●		●	●			●	
Tomforde [64]	●	●	●	●		●	●		●		
Zanshin [62]	●	●	●	●		●	●			●	●
REACT	●	●	●	●	●	●	●	●	●	●	●

Table 1. Overview of related approaches (Depl. = Deployment, Dev. = Development, Eng. = Engine, Eval. = Evaluation, ex. = existing, Sup. = Support)

the system at any time. In several related approaches [4, 16, 17, 31, 53, 62, 68, 69, 72], the influence of the developer already ends with the design process.

Ideally, the source code of the implementation is publicly available and well documented. This helps to foster further research and enables adoption by domain experts in practice. Only a small subset of existing approaches [4, 31, 38, 62, 72, 74] is available 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 [31] and Zanshin [62] have been compared in [2].

Accordingly, in [51], we proposed REACT, a reusable runtime environment for model-based adaptations in communication systems. REACT contributes to the state of the art due to its focus on communication systems and domain expert support. None of the existing approaches offers multi-language support, enables decentralized control as well as distributed deployments, and is available as an open source project. We made the source code of REACT's implementation available and compared it with Rainbow. A summary of the comparison is provided in Section 5.

3 REACT- A REUSABLE RUNTIME ENVIRONMENT FOR ADAPTIVE COMMUNICATION SYSTEMS

This section briefly introduces REACT's architecture and internal feedback loop. We refer the interested reader to [51] for more details and justifications of design decisions.

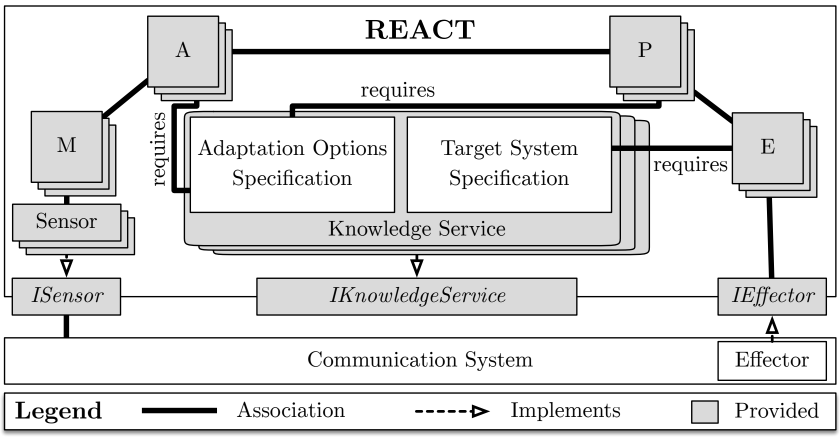

Fig. 1. REACT's architecture in a UML-like notation. It consists of one or multiple MAPE feedback loop(s) connected to instance(s) of the knowledge service with the *adaptation options specification* and *target system specification* provided by the domain expert. REACT's reusable feedback loop uses the *adaptation options specification* to solve the current adaptation problem and maps it to the target system with the *target system specification*. The target system connects to REACT via well-defined sensor and effector interfaces.

3.1 REACT's Architecture

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 a feedback loop as well as interfaces for connecting target systems. Potential target systems in the communication systems domain are overlay networks such as peer-to-peer systems and underlay networks, e.g., in SDN scenarios. However, REACT could possibly be used in other application domains as well. The feedback loop follows the MAPE-K architecture that consists of components for (i) **M**onitoring the system and the environment, (ii) **A**nalyzing the monitored data for necessary adaptations, (iii) **P**lanning the adaptations, and (iv) **E**xecuting the adaptations in the target system as well as (v) a shared **K**nowledge base [36]. The feedback loop uses information stored in an instance of the knowledge for reasoning. It receives sensor information from the communication system as an input and determines the required adaptations as an output via interfaces. Figure 1 shows REACT's architecture on top of a communication system using a UML-like notation. The MAPE components and the knowledge service are generic, internal parts of REACT and are independent from the use case. These gray parts in Figure 1 are encapsulated in a ready-to-use fashion and do not require any programming effort from the domain expert. The white boxes represent the specifications and the effector implementation that have to be provided by the domain expert.

REACT requires two models:

1) The ***adaptation options specification*** 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 target system, i.e., the solution space. After solving a problem in the problem space, REACT maps the result to the solution space according to the *target system specification*.

With these two models, REACT is able to perform architectural as well as parametric adaptation [43]. The separation of the two models decouples the specification of the reconfiguration behavior from the target system and its architecture. REACT 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. REACT's internal MAPE components themselves are reusable since they are working with arbitrary *adaptation options specifications* and *target system specifications.*

To connect to the underlying communication system, REACT provides programming language independent sensor and effector interfaces (`ISensor` and `IEffector`). The sensor receives live context information from different parts of the communication system and forwards it to the feedback loop. The effector transfers the result of the feedback loop to the respective part of the communication system. The exposed `IKnowledgeService` interface can be used by domain experts to update the specifications stored in a knowledge service instance at runtime. The `IKnowledgeService` interface thus allows, for instance, REACT to be connected to a self-improvement [39] module that continuously learns and improves the models. Multiple 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. Thus, this enables high scalability and allows distributed deployments and decentralized control. Fully decentralized or hybrid patterns, as described in [21], are realizable.

3.2 Enabling Self-Adaptivity with REACT

In this section, we summarize the implementation of REACT and how it achieves self-adaptivity. First, we describe how domain experts use a model-based specification approach for self-adaptation with REACT. Second, we explain REACT's integrated feedback loop that leverages the model-based specification without human intervention. Third, we show how REACT makes decentralized control, distributed deployment, and changes at runtime possible.

3.2.1 Modeling. An essential part of REACT are the models of the adaptation behavior (*adaptation options specification*) and of the target system (*target system specification*). The domain expert provides these models at design time and may update them at runtime. REACT uses the models at runtime to adapt the target system. REACT supports *adaptation options specifications* in the structural specification language Clafer (**cla**ss, **fe**ature, **r**eference) [7].

A Clafer-based model is created using a single type of element, named Clafer. 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 domain expert uses REACT 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) and additional servers are available. Listing 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 cardinalities, while the default 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" [3] like a class in object-oriented programming. Hence, a Clafer can inherit from an abstract Clafer and use abstract Clafers like a type. The lines 2-5 describe an abstract entity of type *Context* with integer attributes. A solution of this problem space requires to have exactly 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

```
 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  [ if Context.servers < Context.maxServers then one ExtraServers else no ExtraServers
 9    if Context.responseTime >= 75 then one HighRT else no HighRT
10    if HighRT && ExtraServers then one ServerLauncher else no ServerLauncher ]
```

Listing 1. Adaptation options specification in Clafer for self-adaptive cloud server management.

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.

REACT uses separate models for the adaptation behavior, which is modeled in Clafer, and the target system. Hence, REACT needs a mapping from the problem space to the solution space, which represents the target system. For this purpose, REACT uses the *target system specification*, which the domain expert provides in UML as class diagrams. REACT parses the UML class diagram as an XML file complying to the *UML 2 Abstract Syntax Metamodel* by the Object Management Group. Due to this standardized format, the domain expert can create the XML file manually or use a graphical editor that offers an export in this format such as Papyrus[3]. In the cloud server management example with its *adaptation options specification* in Listing 1, the simplest UML model only contains a single class named `ServerLauncher`. An instance of this UML model indicates if the corresponding class should be present in the target system or not.

3.2.2 Integrated Feedback Loop. The previous section describes the modeling of the *adaptation options specification* in Clafer and the *target system specification* in UML. Now, we show how REACT autonomously leverages these use case dependent models to achieve self-adaptivity. Figure 2 shows the behavior of REACT's integrated MAPE-K feedback loop in the aforementioned cloud server management example. The feedback loop starts as soon as 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 monitoring component.

REACT allows domain experts to choose from multiple integrated monitoring strategies. In the *default* strategy, the monitor parses the raw JSON data and hands it to the analyzer as a map ❷. REACT offers an *aggregation* strategy that additionally aggregates information from multiple sensors and a *windowing* strategy that applies a sliding window approach to the incoming sensor values. An `IMonitoringStrategy` interface further makes it possible for advanced users to create, share, and integrate custom monitoring strategies.

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. REACT 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.

REACT's planner merges the generated Clafers with the *adaptation options specification* to the problem specification. The problem specification thus contains the global constraints of the *adaptation options specification* and the current constraints imposed by the sensor data. Now, REACT

[3]https://www.eclipse.org/papyrus/

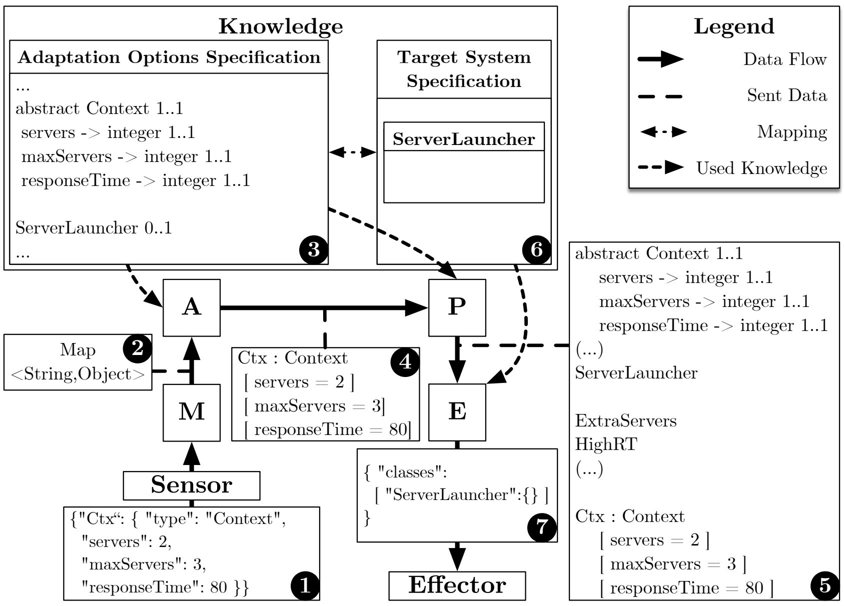

Fig. 2. An adaptation cycle of REACT for the cloud server management example. The analyzer maps the JSON-based sensor information to the *adaptation options specification* in Clafer. The planner evaluates the model and finds a valid instance. Here, it adds a `ServerLauncher` Clafer as starting a new server is desired. The effector maps the plan to the *target system specification* in UML and transfers the adaptation to the target system.

solves this problem specification as a constraint-satisfaction problem (CSP) with Chocosolver [54], a Java-based library for constraint programming. 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. REACT maps the Clafers by name to the classes or parameters of the UML model and creates an 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.* REACT transforms the UML instance to a language-independent representation. Finally, the executor passes this representation via the effector interface ❼ to the target system, where adaptations will take place. The integrated feedback loop of REACT works with arbitrary *adaptation options specifications* and *target system specifications* and is thus applicable to a wide range of scenarios.

3.2.3 Communication and Deployment. We showed how REACT makes it possible to build self-adaptive communication systems or integrate self-adaptive behavior into a legacy system while only demanding two models from the domain expert and low programming effort. Another main strength of REACT is its ability to run distributed. To achieve this, REACT's internal communication interfaces between MAPE components, knowledge service, and sensor/effector interfaces are specified in ZeroC Ice's Interface Definition Language [35]. Ice is a well-established framework for creating Remote Procedure Call (RPC) bindings to many programming languages. For supporting distribution, runtime change of the deployment, and bootstrapping, REACT's MAPE-K components and sensors are integrated into OSGi bundles with iPOJO [26]. The domain expert deploys the system with a key-value-based configuration file for each component. REACT's OSGi runtime then instantiates one component for each available key-value-based configuration file on a host. Thus, domain experts can deploy the feedback loop easily in a distributed way. For setting up the connections to the successor and knowledge component(s), REACT uses Multicast DNS in local networks or a Consul[4] registry for automatic setup, or manual IP address and port specifications.

Apart from distributed deployment, REACT further supports changes of the *adaptation options specification*, the *target system specification*, and the deployment at runtime. REACT allows to use an RPC at runtime to add models remotely to the knowledge service. Hence, a domain expert can change the self-adaptive behavior without interruptions. The domain expert can also change the deployment or re-locate REACT's components. After updating the configuration files, REACT's OSGi containers reconfigure automatically.

4 REACT-ION: SITUATION AWARENESS WITH REACT

In this paper, we present REACT-ION— an extension for REACT that achieves situation awareness. Situation awareness can be defined as"*the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future*" [25]. Accordingly, the three levels of situation awareness consist of (i) perception, (ii) comprehension, and (iii) prediction [25]. Situation awareness has been already applied in the ubiquitous and pervasive computing domain, mainly with a focus on the perception and comprehension levels (e. g., see [73]). Fredericks *et al.* have studied and discussed the usefulness of situation awareness in self-adaptive systems in [29]. REACT-ION supports all three levels of situation awareness. First, it extends REACT with perception by providing a context management module (cf. Section 4.1). As the comprehension of a situation is dependent on the use case and scenario, the context manager is also able to distribute context information for supporting the domain expert to reason about it and integrate arbitrary situation recognition techniques. For addressing the projection, we show how REACT-ION can be used to adapt a target system proactively (cf. Section 4.2). As soon as situations can be determined, this information can be used for adapting the adaptation logic (self-improvement [39]) for tackling uncertainty [27]. Thus, we exemplarily apply compositional adaptation of the adaptation logic by using multiple feedback loops (cf. Section 4.3). This enables the domain expert to choose a feedback loop based on the current situation of the system. As REACT-ION is an optional extension, domain experts are free to disable it for use cases that require a lightweight deployment of REACT.

4.1 Context Management Module

REACT-ION provides a context management module as a foundation for situation-awareness. The module includes a database that stores the current context, as well as past context, and the corresponding adaptations by the feedback loop. This has two implications. First, it paves the way towards self-improvement [39] with REACT, i.e. adapting the adaptation logic. The context manager is able to collect and distribute the context to an external software component such as a machine learning pipeline. This external component may reason on the data to infer the current situation. Based on the situation, the external component is then able to modify the REACT-based system with REACT's well-integrated options for runtime modification (cf. Section 3.1). Second, context management may accelerate adaptation. If the context has been similar in the past, REACT-ION may skip the planning process and perform the same adaptation again in case that the appropriateness of this adaptation was positively evaluated in the retrospective.

[4]https://www.consul.io/

Context management is a major research focus in pervasive and context-aware computing. The interested reader is referred to [33, 41] for an overview of context-aware systems. Prominent approaches for context management are Aura [32], CARISMA [15], Gaia [55], or PROACTIVE [66]. The approaches differ in terms of context *storage* and context *distribution*. As far as storage is concerned, designs range from model-based approaches (e.g., [15]) to ontology-based approaches (e.g., [55]). Perera *et al.* [50] as well as Lim and Dey [41] observe a shift from rules towards ontology-based approaches for reasoning. REACT-ION is applicable in a wide range of use cases, which makes a fixed ontology unsuitable for context management. Thus, we follow a model-based approach for REACT-ION. The distribution of context to an external component either works query- or subscription-based [50]. REACT-ION requires a flexible solution that is applicable in many use cases. Thus, the context management module is able to handle single queries or to notify subscribers about context changes.

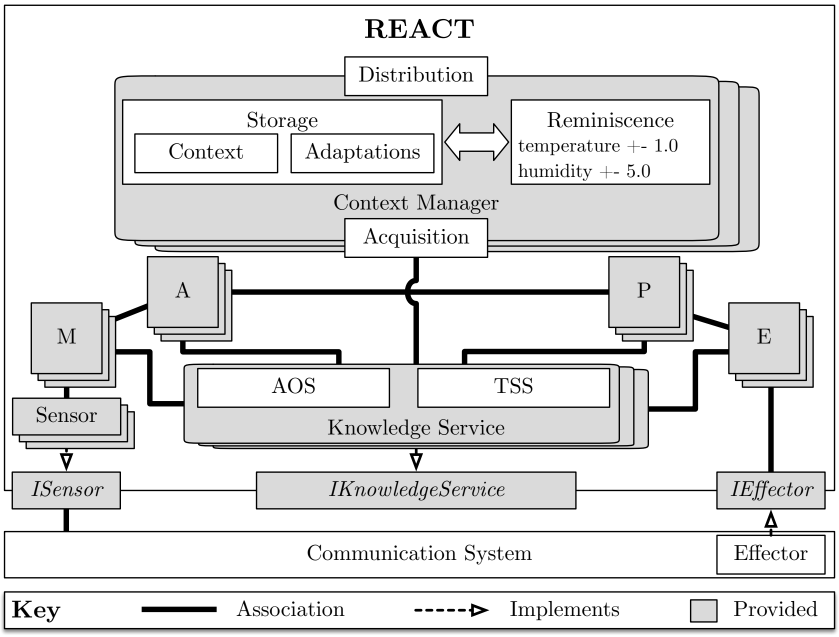

Fig. 3. Architecture of REACT including the optional context manager module. It consists of a *Acquisition*, *Storage*, *Reminiscence*, and *Distribution* functionality and is connected to the knowledge service. AOS: *adaptation options specification*, TSS: *target system specification*.

4.1.1 *Architecture.*

A holistic context management approach supports all four phases of the context life cycle, consisting of context *acquisition*, *modeling*, *reasoning*, and *distribution* [50, 60]. We design REACT-ION's context management module along these four phases. Figure 3 shows the architecture of REACT-ION including the *Context Manager.*

The context management module contains a *Storage* component with a database for modeling the context and interfaces for manipulating and querying the data. The database stores the context information and the corresponding adaptation decisions by REACT-ION. Following the context life cycle, context acquisition starts at REACT-ION's `ISensor` interface that receives sensor data from the target system. The analyzer sends the current context information to the knowledge service, which forwards it to the context manager for storage. In this step, the context information can be distinguished from system parts, which should be reconfigured, by excluding all parts that have a corresponding element in the *target system specification*.

REACT-ION offers two options for context reasoning. First, the internal *Reminiscence* component is able to decide whether the current context has been sensed in the past. In this case, REACT-ION skips the planning phase of its feedback loop and executes the previously planned adaptation. If a context is unknown by the context storage, the context is added to the context storage and the loop continues. Accordingly, the executor sends the corresponding *target system specification* to the knowledge component, which forwards it to the context manager. This enables to use this adaptation in future loop executions that skip the planning phase. The configuration file of the analyzer (cp. Section 3.2.3) allows domain experts to enable or disable this behavior. As slight deviations in a numerical context dimension should be interpreted as a similar context, the *Reminiscence* component uses absolute thresholds. Only if the new value differs from a previous value by more than this absolute value, it will be considered as a new state. The thresholds are configurable at design time and at runtime. Second, an external component may reason on the context data.

REACT-ION includes a *Distribution* component for the last of the four context life cycle phases. This component communicates with external software that, e.g., reasons on the context data to detect the current situation. It distributes (new) context information and planned adaptations via a publish/subscribe system. Apart from a subscription, the *Distribution* component also offers the option to query the *Storage* component via the publish/subscribe system.

4.1.2 Implementation. For the context acquisition, the context data in form of Clafers created in the analyzer is used as foundation. All Clafers in the *adaptation options specification* without a corresponding element in the UML-based *target system specification* represent the context. The *Storage* component includes a MySQL[5] database with two tables for i) context and ii) adaptations. REACT-ION automatically generates the table structures at the system start based on the *adaptation options specification* and the *target system specification*. The *adaptation options specification* already pre-defines a suitable schema for the SQL-based context table(s) to store the context information. A foreign key relation references the corresponding adaptation in the adaptations table. The *Reminiscence* component contains a map structure that stores the configurable percentage thresholds for numerical context dimensions. REACT-ION offers an interface to adjust these thresholds at runtime via a method in the `IKnowledgeService` interface (cp. Section 3.1). This is beneficial in use cases where the granularity of planning should be adjusted at runtime, e.g., to skip the planning phase more often when the computational load for the REACT-based feedback loop is high. The *Distribution* component uses the Message Queuing Telemetry Transport (MQTT) protocol to provide a lightweight publish/subscribe solution. The usage of the well-established protocol enables an intuitive communication of external components with REACT-ION's context management module. We use Eclipse Mosquitto[6] for the MQTT broker and Eclipse Paho[7] as the Java library for communicating with the broker. If the context distribution is enabled via a method in the `IKnowledgeService` interface (cp. Section 3.1), the module connects to an MQTT broker and publishes events. REACT-ION allows domain experts to start a local MQTT broker or to connect to an external one.

The context management module offers a foundation for several sophisticated use cases with REACT-ION. In the following, we investigate two options. First, we show in Section 4.2 how

[5]https://www.mysql.com
[6]https://mosquitto.org/
[7]https://www.eclipse.org/paho/

REACT-ION achieves proactive adaptation based on the context management module. Second, we show in Section 4.3 how context-awareness may lead to self-improvement with REACT-ION.

4.2 Proactive Adaptation with REACT-ION

Self-adaptive systems either perform *reactive* or *proactive* adaptation [40]. Traditionally, many self-adaptive systems only adapt after a change in the target system has been detected, which makes them reactive. This has several disadvantages such as slower adaptation to changes which may — in the worst case — lead to a failing target system. Proactive adaptation aims at avoiding such situations in the first place [33, 40]. In general, for performing proactive adaptation, the system context has to be known [66]. REACT-ION's context management module covers this requirement. In this section, we show how proactive adaptation can be achieved with REACT-ION.

Proactive adaptation with REACT-ION requires three steps: i) communicating the context to a prediction system, ii) predicting future context, and iii) using the prediction to plan adaptations. In REACT-ION's context management module, the context is represented as a Clafer-based specification which is transformed into a context database. Hence, the context management module provides a history of context information in a structured way in its *Storage* component. In addition, the *Distribution* component is able to communicate with external prediction and learning systems. Thus, REACT-ION's context management module is suitable to perform the first step of proactive adaptation.

The choice of the prediction system for step ii) is highly dependent on the use case. Domain experts are able to easily connect their prediction system of choice to REACT-ION's context management module via the platform-independent publish/subscribe system. Often, time series forecasting is used for prediction [76]. In Section 5.2, we therefore show how to connect a REACT-based self-adaptive system to the *Telescope* [75] time series forecasting framework to make context predictions.

REACT-ION offers two options to use the prediction for proactive adaptation in step iii): an *implicit* and an *explicit* approach. For the implicit approach, the prediction system sends the predictions to REACT-ION via the `ISensor` interface. Instead of the current context information, REACT-ION uses the prediction for the usual planning process. Consequently, REACT-ION adapts the system based on the predicted information instead of the current context, which results in proactive adaptation. This approach leads to minimal effort for domain experts since *adaptation options specification* and *target system specification* do not need to be changed. On the downside, this approach may lead to bad adaptation decisions as it only relies on — possibly inaccurate – predictions. Thus, REACT-ION also offers the explicit approach, where the predicted context is added to the *adaptation options specification*. In this case, the adaptation decisions are based on both the current context and the predicted context. Even though this approach requires additional modeling overhead, it enables domain experts to influence how the predictions are incorporated into the decision-making process.

4.3 Self-Improvement with REACT-ION

REACT-ION's context management module enables domain experts to introduce situation awareness to their system. Additionally, REACT-ION offers the option to modify the feedback loop at runtime (cp. Section 3.1). When combining both, domain experts are able to modify the feedback loop based on the current situation. This "adjustment of the adaptation logic to handle former unknown circumstances or changes" [39, p. 2] in the environment or the target system is called *self-improvement*. Self-improvement is important as complexity and uncertainty may lead to situations that were not foreseeable at design time [65]. Examples for such situations include a significant change of the characteristics of the system's environment or user group (hence, the users' objectives) or the requirement to add or update adaptation decision rules through learning.

REACT-ION offers three options for self-improvement. First, the *adaptation options specification* and the *target system specification* may be adjusted at runtime (cp. Section 3.1) based on the current situation. This leads to a change of the reconfiguration behavior. Second, the deployment of REACT-ION's MAPE-K components is changeable at runtime (cp. Section 3.2.3). For instance, in high load situations, analyzer and planner may be migrated to separate machines. Third, several MAPE-K loops might exist simultaneously and might be used for different situations. In this section, we show how REACT-ION is able to achieve self-improvement in this case.

In the literature, many approaches for analyzing and planning in self-adaptive systems exist. Some approaches provide fast adaptations, others are easy to use, applicable to a wider range of use cases, or require less memory. We propose to combine several reasoning approaches and to choose the suitable feedback loop based on the current situation. The reconfiguration behavior of a REACT-based system is modeled in Clafer and UML. Planning happens by solving a constraint-satisfaction problem (cp. Section 3.2). This approach is easy-to-use for domain experts but may lead to considerable overhead in terms of computational complexity and memory footprint. We now integrate an alternative reasoning approach into REACT-ION. This approach relies on context feature models (CFMs) [34, 58] for specifying the problem space. A CFM is a hierarchical tree-like model and specifies the reconfiguration space of a self-adaptive system including the adaptations based on the system context. While the left subtree represents the configuration features and attributes of the system, the right subtree represents context features and context attributes. Constraints between both subtrees resemble the reconfiguration behavior. Figure 4 shows a small example CFM of a smartphone reconfiguring its wireless connectivity. In the shown model, the system can turn on the *LTE* and/or *Wifi* features for providing the *Wireless Connectivity* feature. The context includes the current latency of the connection and the location of the phone. The phone can either be *Away* or at *Home*. Accordingly, the shown constraints turn Wifi on at home, and off when being away. Finally, in cases the use of the Wifi connection results in higher latency than 100 ms, the LTE connection is enforced for possibly lowering the latency.

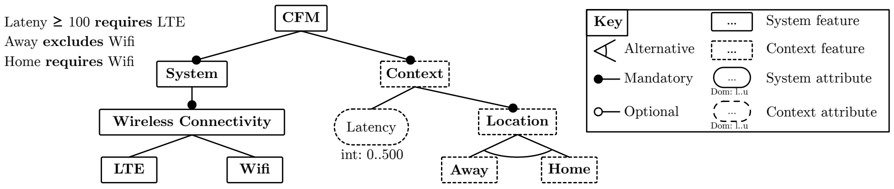

Fig. 4. Exemplary CFM that represents the reconfiguration space of the wireless connectivity of a smartphone. Depending on the constraints, it determines the selection of LTE or Wifi for providing wireless connectivity.

CFMs can be translated into boolean satisfiability (SAT) problems or mixed-integer linear programming (MILP) problems [70]. While SAT problems can be solved relatively fast with lower expressiveness, MILP problems can be applied to specifically state integer or real parameters and optimize the results using multi-objective optimization. Figure 5 shows REACT-ION's architecture with CFM-based reasoning. The knowledge consists of the CFM specified with CardyGAn [59], which models the problem space, and a (UML) class diagram, which models the solution space. In the monitoring step, the sensor data is preprocessed. In the analyzing phase, the right part of the CFM — the context — is instantiated. Based on this context information, the planning component transforms the CFM and the context instance to a SAT or MILP problem resulting in a complete system configuration including the system features. Finally, the executing phase creates a class diagram instance of the solution space from the completed CFM. For more information on this feedback loop, the interested reader is referred to [70].

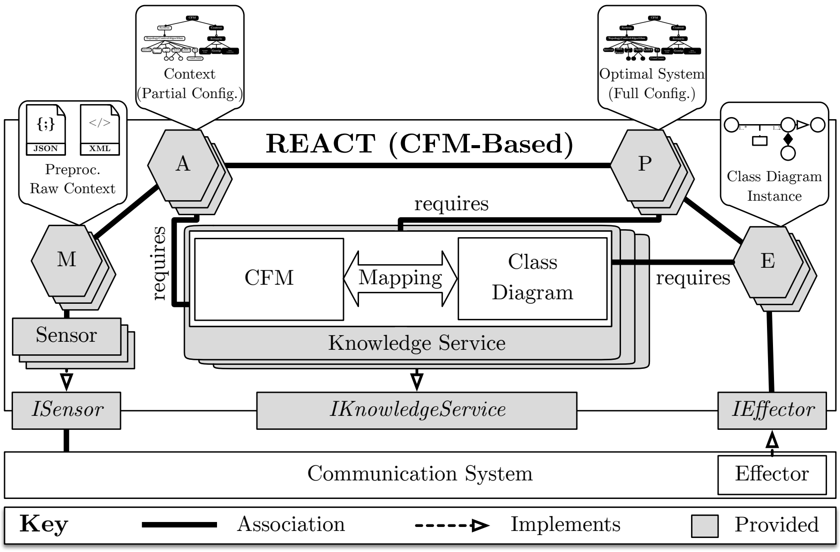

Fig. 5. Architecture and functionality of REACT-ION using the CFM-based feedback loop. The knowledge consists of a CFM representing the problem space, a class diagram representing the solution space, and an explicit mapping between both. The boxes attached to the MAPE functionalities show the results of each component [70].

Deploying both the Clafer-based feedback loop and the CFM-based loop simultaneously has several advantages. For instance, it might be beneficial to execute the CFM-based reasoning approach with a SAT solver if the target system is in a critical state. While the result might not be optimal, it could be good enough for bringing the system back into a non-critical state as fast as possible. At the same time, a more complex Clafer-based planner could be executed as well, which provides an optimized solution later. Hence, situation awareness enabled by the context management module may improve adaptation decisions at runtime by selecting the suitable feedback loop or by executing multiple loops in parallel. In Section 5.3, we evaluate self-improvement with REACT-ION in a case study in which we use the Clafer-based and the CFM-based feedback loops simultaneously.

5 EVALUATION

This section, first, presents (i) a brief summary of the results comparing REACT with Rainbow [31], a well-known and frequently applied framework for model-based adaptation, and (ii) a condensed overview of the application of REACT in an emulated communication system in the field of Software-Defined Networking (SDN) based on [51] (cp. Section 5.1). Second, Section 5.2 builds upon the extensions of this paper and evaluates proactive adaptations using REACT in a smart grid use case. Third, Section 5.3 outlines the evaluation of applying two different feedback loops as part of REACT, which enables to select feedback loops at runtime for self-improvement. Finally, Section 5.4 discusses possible threats to validity. With the evaluation, we target the prediction dimension for situation-awareness [25] as the other two dimensions — perception and comprehension — mainly would be a functional evaluation of the context management module. Additionally, the interplay of the feedback loops shows a practical use case for the benefits of situation awareness.

5.1 Cloud Server Management and SDN-Based Wifi Handover

In our first experiment, we compared REACT with the well-known Rainbow framework [31] in terms of development effort, performance, and features. 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. The adaptation manager, on receiving the adaptation trigger, chooses the “best” adaption plan — on the basis of stakeholder utility preferences and the current state of the system, as reflected in the models — to execute, and passes it on to the strategy executor, which executes the strategy on the target system via effectors. The underlying decision making model is based on decision theory and utility [20]. 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 [13], predictive proactive adaptation [45], and human-machine cooperation [14]. For this first evaluation, we used the SEAMS exemplar SWIM (Simulator for Web Infrastructure and Management) [46], which represents a cloud system.

In the second experiment, we showed REACT’s focus on communication systems in a real-world SDN-based use case adding adaptive behavior to an underlay network. In this second case, a car receives a live stream from a streaming server via a wireless network connection. 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 using SDN as foundation. In the following, we summarize the results of the evaluations of REACT’s development effort, performance, and capabilities in comparison to the Rainbow framework and the applicability of REACT in the real-world SDN-based communication system. For the full details, the interested reader is referred to [51].

RQ1.1: How much development effort is required for REACT compared to the state of the art?

As far as development effort is concerned, two metrics influence the domain expert’s experience: the lines of code (SLOC) 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.

We observed that specifying the adaptive behavior with REACT requires considerably fewer SLOC. The domain expert has to write 152 SLOC in 2 files with clear responsibilities. To achieve the same behavior with Rainbow, the domain expert has to write 593 SLOC in 6 different files using various languages. Next, we assessed the development effort for the interface implementation. We could show 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, domain experts can use their preferred language. We acknowledge that SLOC as a metric might have different shortcoming, however, it is frequently applied as a metric to provide an estimation of the development effort (e.g., in [20, 38, 67]).

RQ1.2: How well performs REACT compared to the state of the art?

REACT considerably outperforms Rainbow in the monitoring and analyzing phase. Since Rainbow holds an exact architecture model of the target system, 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. This design choice thus allows a more complex analysis of the target system architecture at the cost of slower adaptation. The total execution time of an adaptation cycle in REACT is determined to a very 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 [6, 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. In total, REACT's average adaptation cycle execution requires 84 ms in comparison to 215 ms in Rainbow. Thus, we argue that REACT is well-applicable in scenarios where fast adaptation is required.

RQ1.3: How differs REACT and Rainbow in terms of capabilities?

Rainbow has its strengths in more in-depth analysis using its architecture model and a less complex planning phase as a result. In addition, it works utility-based with the possibility to weight optimization goals, which may considerably reduce a domain expert's effort in scenarios with multiple goals. 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.

RQ2: Can REACT be implemented and used effectively in a real-world communication system?

In [51] we could show that REACT can be applied effectively in a real-world communication system within an SDN-based Wifi handover scenario. In addition, REACT makes it possible to efficiently change the behavior of the SDN controller by changing the *adaptation options specification*. It further allows to port 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 requires specific SDN applications with different interfaces to the controller for applying a certain behavior in the network.

5.2 Proactive Adaptation

In the third experiment, we apply proactive adaptation with REACT in a smart grid scenarioas prediction and, resulting, proactive adaptation is an important aspect for situation awareness [25]. The smart grid consists of multiple households that consume power and multiple power sources that produce the same. The goal of the smart grid is to be self-sufficient. If the energy consumers more power than available, the required power is taken from the general power lines outside of the smart grid. Analogously, excess power is given away to the surrounding power lines. A domain expert uses REACT to implement adaptive behavior in the smart grid with two goals. First, the smart grid should activate at least as many power sources as needed to fulfill the current power demand. This is the primary goal. Second, the production of excess power — by activating too many power sources — should be kept at a minimum if possible.

We simulate the scenario with the Python-based smart grid simulator Mosaik[8]. The simulation includes 10 households, which consume power based on realistic usage profiles. Additionally, 40 power sources produce power. Immediately after activating the power sources, they produce power at a constant rate. Each simulation run simulates a time period of two weeks in steps of 15 minutes. We execute 30 runs with different household power profiles, each once with reactive adaptation and once with proactive adaptation.

[8]https://mosaik.offis.de/

Additionally to a reactive adaptation mechanism, we implement proactive adaptation based on REACT's context management module as described in Section 4.2. We connect REACT to Telescope [75] — an R-based software for univariate time-series forecasting — for predicting the future power consumption. Telescope uses the data of the first week for predicting the second week. The *adaptation options specification* in Clafer activates power sources based on the currently required power, the number of already running power sources, and a predicted power requirement for the next simulation step. When applying reactive adaptation, this prediction value is not used. The horizon of Telescope is set to 1, i.e., only the power consumption in the following simulation step — the next 15 minutes — is predicted.

RQ3.1: How improve system performance when applying proactive adaptations using REACT?

Figure 6 (a) compares the average number of overloads for reactive and proactive adaptation. An overload occurs if the power production in the smart grid is lower than the power consumption. We observe that proactive adaptation with REACT is able to decrease the number of overloads from 230 to 203 on average in comparison to reactive adaptation. Figure 6 (b) shows the excess power produced in the smart grid. Since activating power sources increases the production step-wise and not continuously, proactive adaptation leads to more excess power. This, however, is the desired behavior, as overloads are dangerous for grid stability and require buying additional energy from outside of the smart grid. We therefore conclude that proactive adaptation with REACT and Telescope is able to anticipate increases in power consumption and to activate power sources accordingly. This not only helps to improve the stability of the grid but also contributes to its resilience.

RQ3.2: How is proactive adaptation related to situation awareness?

According to [25], prediction is an important requirement for situation awareness, i.e., the requirement to foresee changes in the situation and react accordingly, e.g., through proactive adaptation. However, our evaluation shows a second facet in the relation of situation awareness and proactive adaptation. Figure 7 depicts an excerpt of an exemplary simulation run. The first five hours of the excerpt show the strengths of situation awareness: Through prediction of the new demand (i.e., the new situation) and proactive adaptation, the production is increased in advance to avoid overloads. However, sudden peaks in the power consumption — as shown between 7:45 and 8:15 — are difficult to predict and provide an interesting example that the relation between situation awareness and proactive adaptation is bi-directional. When integrating proactive adaptation, the reliability of the prediction/forecasts are an important aspect. If those are not reliable for a specific situation, reactive adaptation might be beneficial as the best known adaptation for the situation is performed rather than applying an adaptation for a situation that might not happen and potentially decreases system performance even stronger as a reactive, delayed adaptation would impact performance. Further, reactive adaptation is required as backup for unknown situation. For the smart grid scenario, the current situation of Corona lockdowns would be such an example, because its creates completely different situations — people stay at home at times when they would be usually at work / school — that the prediction/forecasting framework did not encounter and hence was not able to learn those patterns. Consequently, it is important to integrate a situation-aware choice whether to apply proactive or reactive adaptation, depending on the reliability of the prediction of the future situation.

5.3 Self-Improvement with Multiple REACT-Based Feedback Loops

This section applies self-improvement with REACT by using multiple MAPE-K feedback loops in one use case. In this experiment, REACT adapts a cloud server deployment providing a web application. This experiment again uses SWIM [46], which offers a reproducible way for evaluating

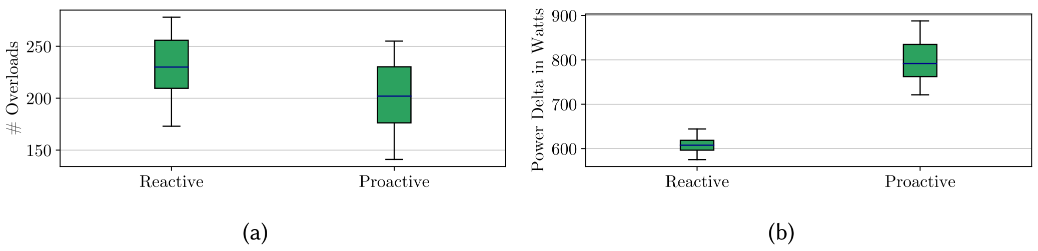

Fig. 6. Average number of overloaded simulation steps (a) and the average power delta (b) of the reactive and the proactive approach.

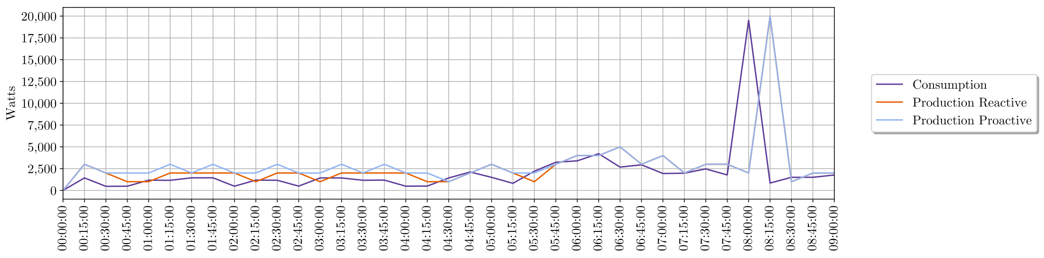

Fig. 7. Excerpt of the timeline of an exemplary run of the proactivity evaluation.

adaptation logics in a web server environment. It is a simulation environment based on OMNeT++. 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 can contain optional content (e.g., advertisements) which increases the response time but also leads to additional revenue for the web site operator. The overall goal of the system is thus continuously reaching a fixed response time goal, while 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 number of responses with optional content.

Experimental Setup: We deploy three MAPE-K feedback loops with REACT. The first feedback loop uses the original reasoning approach that solves the CSP (cp. Section 3). This approach applies Chocosolver [54]. The other two feedback loops use the CFM-based reasoning approach introduced in Section 4.3. The second loop plans adaptations with the SAT4J [11] solver for boolean satisfiability problems whereas the third loop uses the CPLEX[9] solver for MILP problems.

In the scenario, the 30 minute ClarkNet [23] trace provided with SWIM is used. Every run is repeated 20 times, and the context of the system is fetched every 10 seconds. We measure the average planning time, the standard deviation of the planning time, as well as the average utility. The utility is provided by SWIM and describes the quality of the adaptation decisions.

We perform two experiments. First, we use a basic problem space specification that can be solved by all three reasoning approaches. This ensures that the different capabilities in terms of problem specification do not have an influence on the results. Second, we use three different problem space specifications that exhaust the respective capabilities of the solvers. In this experiment, the MILP- and CSP-based reasoning approaches are able to directly plan the absolute dimmer value

[9]http://www.ibm.com/analytics/cplex-optimizer

and number of servers to start or stop. The specification for the SAT-based reasoning approaches remains the same.

This section answers the following research question:

RQ4: Is self-improvement with REACT able to combine the advantages of several reasoning approaches with regards to adaptation speed, effectiveness, and expressiveness of the specification?

Figure 8a shows the results of the first experiment with the same (simplified) specification for all three feedback loops as well as the full specifications in comparison. Figure 8b shows the runtimes of the planners. We observe that the SAT-based reasoning approach leads to the fastest adaptations. It is 10 % faster than the MILP-based planner and 73 % faster than the CSP-based planner. As far as the quality of the adaptations is concerned, we observe only minor differences. The SAT-based approach leads to the highest utility comparing the simplified cases with it. We argue that the faster adaptation is the reason for this. With the SAT-based planner, the system reacts more promptly to changes in the load. Next, we look at the results of the second experiment with different specifications for all three feedback loops. These specifications exploit the different capabilities of the solvers in terms of expressiveness. The MILP-based approach needs considerably more time compared to the results from the first experiment. The CSP-based approach, however, only requires additional 5.5 ms in this experiment.

Figure 8b combines the results of the two experiments. The advanced modeling capabilities in the MILP- and CSP-based feedback loops considerably improve the quality of the adaptations.

Answering *RQ4*, we therefore conclude that there is a tradeoff between planning time and adaptation quality. If fast adaptations are required, employing a SAT solver or a MILP solver with a restricted SAT-based specification is possibly the better choice. This, however, leads to worse adaptation decisions. Thus, it is beneficial to choose the reasoning approach based on the current situationand the system objectives. REACT-ION's context management module supports situation awareness and self-improvement. It is even possible to use multiple loops simultaneously, e. g., to get a result as fast as possible, and, simultaneously, run in the background another solver that tries to identify a better solution. However, the integration of many feedback loops could lead to conflicts. Hence, selecting and handling multiple feedback loops is a field for future research.

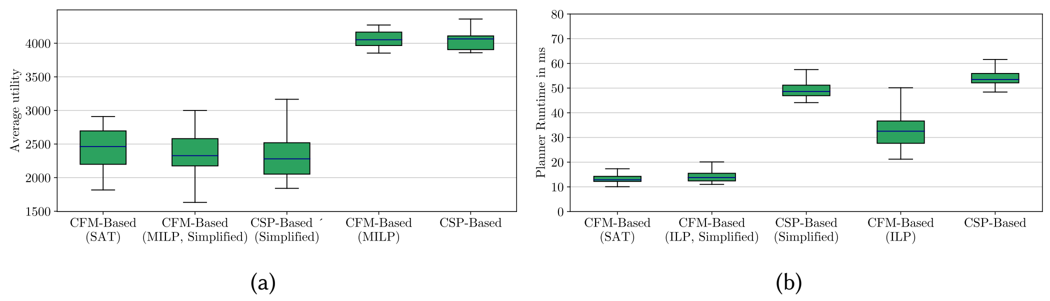

Fig. 8. Average utility per feedback loop type (a) and according average planning times (b).

5.4 Threats to Validity

We identified the following threats to validity for your evalaution results. In the evaluation, we measure SLOC and the number of different languages to show REACT's low development effort for domain experts. Even though SLOC are frequently used as a metric (e.g., in [20, 38, 67]), a future user study with domain experts who apply REACT in different scenarios would strengthen validity.

In the second experiment, we adapt an underlay network with REACT, showing its capabilities for decentralized control, distributed deployment, and multi-sensor support. However, we omit an analysis of the sclability of our approach w.r.t. (i) large Clafer and UML models and (ii) larger system sizes. This work is further limited to a comparison with Rainbow and to two use cases only. Future research may include a comparison to other frameworks such as SASSY [44] or StarMX [4] in additional use cases from the communication systems domain. Considering situation awareness, only Telescope [75] as external prediction approach has been applied and we omit a comparison of several approaches for predition. Next, we plan to investigate possibilities to provide a standardized proactivity solution as part of REACT itself. Also, we focus in this work on compositional self-improvement. As future work, we aim at evaluating also other situation-aware self-improvement possibilities besides compositional adaptations. This includes parametric adaptation, i. e., changing the models autonomously as well as deployment changes of the adaptation logic. These deployment changes could, e. g., result in moving the computationally intensive planner to faster machines at runtime in high-load situations.

6 CONCLUSION

In this paper, we present REACT-ION, a reusable runtime environment for model-based adaptations in communication systems that supports situation awareness. REACT-ION is an extension of REACT, which integrates a MAPE-K feedback loop that leverages a Clafer and a UML model provided by the domain expert to autonomously achieve self-adaptivity. Due to its support for multiple programming languages, decentralized control, distributed deployments, and runtime modifications, REACT is well-applicable for adapting overlay and underlay networks. We compared REACT to the well-known Rainbow framework, showing that it is easy-to-use for domain experts and suitable for use cases that require fast adaptations. As extensions, this paper presents a context management module, which can be used for providing situation awareness capabilities, executing proactive adaptations, and self-improvement. We applied proactive adaptations using REACT-ION and showed the possibility to run multiple REACT-ION-based feedback loops. This capability can be used for self-improvement by selecting a feedback loop based on the current system situation.

As future work, we plan to integrate additional interfaces that allow developers to directly use own analyzing and planning techniques such as machine learning or a different specification language such as Stitch [5] instead of Clafer. As verification and validation (V&V) is an important research challenge [5, 22], we plan to add verification of dynamic properties such as runtime V&V techniques and guarantees according to costs into REACT, e.g., using model-checking methods. This will ensure the correctness of the models and REACT will give certain runtime guarantees. Future work additionally includes a user study with domain experts which further investigates the development effort. Focussing on such empirical evidence with practitioners has been identified as general challenge for further self-adaptive systems research [71]. For prediction, we integrate the Telescope [75] framework as external prediction approach. We plan to investigate possibilities to provide a standardized solution for prediction/forecasts as part of REACT itself. For example, it might be possible to integrate a recommendation system for time series forecasting (e.g., [76]) which autonomously decides the best suitable algorithm depending on the data characteristics.

ACKNOWLEDGMENT

This work has been co-funded by the German Research Foundation (DFG) as part of project A4 within CRC 1053 – MAKI. The authors would like to thank Veronika Lesch from University of Würzburg for the initial implementation of the Java-R-Bridge.

REFERENCES

[1] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. 2012. Achieving dynamic adaptation via management and interpretation of runtime models. *J. Syst. Softw.* 85, 12 (2012), 2720–2737. https://doi.org/10.1016/

j.jss.2012.05.033

[2] Konstantinos Angelopoulos, Vitor E. Silva Souza, and João Pimentel. 2013. Requirements and architectural approaches to adaptive software systems: A comparative study. In *Proc. SEAMS*. 23–32. https://doi.org/10.1109/SEAMS.2013.6595489

[3] Michal Antkiewicz, Kacper Bak, Krzysztof Czarnecki, Zinovy Diskin, Dina Zayan, and Andrzej Wasowski. 2013. Example-Driven Modeling using Clafer. In *Proc. of MoDELS*, Vol. 1104. 32–41.

[4] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. 2009. StarMX: A framework for developing self-managing Java-based systems. In *Proc. of SEAMS*. 58–67. https://doi.org/10.1109/SEAMS.2009.5069074

[5] B. Cheng et al. 2011. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time - Foundations, Applications, and Roadmaps*. Springer, 101–136. https://doi.org/10.1007/978-3-319-08915-7_4

[6] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *Proc. of SLE*. 102–122. https://doi.org/10.1007/978-3-642-19440-5_7

[7] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2016. Clafer: unifying class and feature modeling. *Software and System Modeling* 15, 3 (2016), 811–845. https://doi.org/10.1007/s10270-014-0441-1

[8] Nelly Bencomo, Robert B. France, Betty Cheng, and Uwe Aßmann (Eds.). 2014. *Models@run.time - Foundations, Applications, and Roadmaps*. Springer. https://doi.org/10.1007/978-3-319-08915-7

[9] Nelly Bencomo, Paul Grace, Carlos A. Flores-Cortés, Danny Hughes, and Gordon S. Blair. 2008. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *Proc. of ICSE*. 811–814. https://doi.org/10.1145/1368088.1368207

[10] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. 2008. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *Proc. of SPLC)*. 23–32.

[11] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 2-3 (2010), 59–6. https://doi.org/10.3233/SAT190075

[12] Gordon S. Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ run.time. *IEEE Computer* 42, 10 (2009), 22–27. https://doi.org/10.1109/MC.2009.326

[13] Javier Cámara, David Garlan, Bradley R. Schmerl, and Ashutosh Pandey. 2015. Optimal planning for architecture-based self-adaptation via model checking of stochastic games. In *Proc. of SAC*. 428–435. https://doi.org/10.1145/2695664.2695680

[14] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2015. Reasoning about Human Participation in Self-Adaptive Systems. In *Proc. of SEAMS*. 146–156. https://doi.org/10.1109/SEAMS.2015.14

[15] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. 2003. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering* 29, 10 (2003), 929–945. https://doi.org/10.1109/TSE.2003.1237173

[16] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. 2012. MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. *IEEE Trans. Software Eng.* 38, 5 (2012), 1138–1159. https://doi.org/10.1109/TSE.2011.68

[17] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. 2008. A model-driven approach for developing self-adaptive pervasive systems. *Models@ runtime* 8 (2008), 97–106.

[18] Marinos Charalambides, George Pavlou, Paris Flegkas, Ning Wang, and Daphné Tuncer. 2011. Managing the future internet through intelligent in-network substrates. *IEEE Network* 25, 6 (2011), 34–40. https://doi.org/10.1109/MNET.2011.6085640

[19] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012), 2860–2875. https://doi.org/10.1016/j.jss.2012.02.060

[20] Shang-Wen Cheng. 2004. *Rainbow: cost-effective software architecture-based self-adaptation*. Ph.D. Dissertation. Carnegie Mellon University.

[21] D. Weyns et al. 2010. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, 76–107. https://doi.org/10.1007/978-3-642-35813-5_4

[22] Rogério de Lemos et al. 2010. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

[23] John Dilley. 1996. Web server workload characterization. *HP Laboratories Technical Report* 24, 96-160 (dec 1996), 1–16. https://doi.org/10.1145/233008.233034

[24] Jim Dowling and Vinny Cahill. 2001. The K-Component Architecture Meta-model for Self-Adaptive Software. In *Proc. of REFLECTION*. 81–88. https://doi.org/10.1007/3-540-45429-2_6

[25] Mica R. Endsley. 1995. Toward a Theory of Situation Awareness in Dynamic Systems. *Hum. Factors* 37, 1 (1995), 32–64. https://doi.org/10.1518/001872095779049543

[26] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. 2007. iPOJO: an Extensible Service-Oriented Component Framework. In *Proc. of SCC*. 474–481. https://doi.org/10.1109/SCC.2007.74

[27] Naeem Esfahani and Sam Malek. 2010. Uncertainty in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems II (Lecture Notes in Computer Science, Vol. 7475)*. Springer, 214–238. https://doi.org/10.1007/978-3-642-35813-5_9

[28] Jacqueline Floch, Svein O. Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. 2006. Using Architecture Models for Runtime Adaptability. *IEEE Software* 23, 2 (2006), 62–70. https://doi.org/10.1109/MS.2006.61

[29] Erik M. Fredericks, Ilias Gerostathopoulos, Christian Krupitzer, and Thomas Vogel. 2019. Planning as Optimization: Dynamically Discovering Optimal Configurations for Runtime Situations. In *Proc. SASO*.

[30] Nadia Gámez, Lidia Fuentes, and José M. Troya. 2015. Creating Self-Adapting Mobile Systems with Dynamic Software Product Lines. *IEEE Software* 32, 2 (2015), 105–112. https://doi.org/10.1109/MS.2014.24

[31] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer* 37, 10 (2004), 46–54. https://doi.org/10.1109/MC.2004.175

[32] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste. 2002. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing* 1, 2 (2002), 22–31. https://doi.org/10.1109/MPRV.2002.1012334

[33] Marcus Handte, Gregor Schiele, Verena Majuntke, Christian Becker, and Pedro José Marrón. 2012. 3PC: System support for adaptive peer-to-peer pervasive computing. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7, 1 (2012), 10:1–10:19. https://doi.org/10.1145/2168260.2168270

[34] Herman Hartmann and Tim Trew. 2008. 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, 12–21. https://doi.org/10.1109/SPLC.2008.15

[35] Michi Henning. 2004. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing* 8, 1 (2004), 66–75. https://doi.org/10.1109/MIC.2004.1260706

[36] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[37] Christian Krupitzer, Martin Breitbach, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2018. A survey on engineering approaches for self-adaptive systems (extended version). https://doi.org/10.1016/j.pmcj.2014.09.009

[38] Christian Krupitzer, Felix Maximilian Roth, Christian Becker, Markus Weckesser, Malte Lochau, and Andy Schürr. 2016. FESAS IDE: An Integrated Development Environment for Autonomic Computing. In *Proc. ICAC*. 15–24. https://doi.org/10.1109/ICAC.2016.49

[39] Christian Krupitzer, Felix Maximilian Roth, Martin Pfannemüller, and Christian Becker. 2016. Comparison of Approaches for Self-Improvement in Self-Adaptive Systems. In *Proc. of ICAC*. 308–314. https://doi.org/10.1109/ICAC.2016.18

[40] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.* 17 (2015), 184–206. https://doi.org/10.1016/j.pmcj.2014.09.009

[41] Brian Y. Lim and Anind K. Dey. 2010. Toolkit to support intelligibility in context-aware applications. In *UbiComp (ACM International Conference Proceeding Series)*. ACM, 13–22. https://doi.org/10.1145/1864349.1864353

[42] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. 1995. Specifying Distributed Software Architectures. In *Proc. of ESEC*. 137–153. https://doi.org/10.1007/3-540-60406-5_12

[43] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty Cheng. 2004. Composing Adaptive Software. *IEEE Computer* 37, 7 (2004), 56–64. https://doi.org/10.1109/MC.2004.48

[44] Daniel A. Menascé, Hassan Gomaa, Sam Malek, and João Pedro Sousa. 2011. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Software* 28, 6 (2011), 78–85. https://doi.org/10.1109/MS.2011.22

[45] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. 2018. Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation. *ACM TAAS* 13, 1 (2018), 3:1–3:36. https://doi.org/10.1145/3149180

[46] Gabriel A. Moreno, Bradley R. Schmerl, and David Garlan. 2018. SWIM: an exemplar for evaluation and comparison of self-adaptation approaches for web applications. In *Proc. of SEAMS*. 137–143. https://doi.org/10.1145/3194133.3194163

[47] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. 2009. Models@ Run.time to Support Dynamic Adaptation. *IEEE Computer* 42, 10 (2009), 44–51. https://doi.org/10.1109/MC.2009.327

[48] P. Oreizy et al. 1999. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and their Applications* 14, 3 (1999), 54–62. https://doi.org/10.1109/5254.769885

[49] Janak J. Parekh, Gail E. Kaiser, Philip Gross, and Giuseppe Valetto. 2006. Retrofitting Autonomic Capabilities onto Legacy Systems. *Cluster Computing* 9, 2 (2006), 141–159. https://doi.org/10.1007/s10586-006-7560-6

[50] Charith Perera, Arkady B. Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials* 16, 1 (2014), 414–454. https://doi.org/10.1109/SURV.2013.042313.00197

[51] Martin Pfannemüller, Martin Breitbach, Christian Krupitzer, Markus Weckesser, Christian Becker, Bradley Schmerl, and Andy Schürr. 2020. REACT: A Model-Based Runtime Environment for Adapting Communication Systems. In *Proc. of ACSOS*. IEEE, 65–74. https://doi.org/10.1109/ACSOS49614.2020.00027

[52] Martin Pfannemüller, Christian Krupitzer, Markus Weckesser, and Christian Becker. 2017. A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems. In *Proc. of ICAC*. 247–254. https://doi.org/10.1109/ICAC.2017.18

[53] Thomas Preisler, Tim Dethlefs, and Wolfgang Renz. 2015. Middleware for Constructing Decentralized Control in Self-Organizing Systems. In *Proc. of ICAC*. 325–330. https://doi.org/10.1109/ICAC.2015.56

[54] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2017. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. http://www.choco-solver.org

[55] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. 2002. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing* 1, 4 (2002), 74–83. https://doi.org/10.1109/MPRV.2002.1158281

[56] S. Malek et al. 2010. An architecture-driven software mobility framework. *J. Syst. Softw.* 83, 6 (2010), 972–989. https://doi.org/10.1016/j.jss.2009.11.003

[57] S. O. Hallsteinsen et al. 2012. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software* 85, 12 (2012), 2840–2859. https://doi.org/10.1016/j.jss.2012.07.052

[58] Karsten Saller, Malte Lochau, and Ingo Reimund. 2013. Context-aware DSPLs: model-based runtime adaptation for resource-constrained systems. In *Proceedings of the Software Product Line Conference (SPLC)*. ACM, 106–113. https://doi.org/10.1145/2499777.2500716

[59] Thomas Schnabel, Markus Weckesser, Roland Kluge, Malte Lochau, and Andy Schürr. 2016. CardyGAn: Tool Support for Cardinality-based Feature Models. In *Proc. of VaMoS, 2016*. ACM, 33–40. https://doi.org/10.1145/2866614.2866619

[60] Omer Berat Sezer, Erdogan Dogdu, and Ahmet Murat Özbayoglu. 2018. Context-Aware Computing, Learning, and Big Data in Internet of Things: A Survey. *IEEE Internet Things J.* 5, 1 (2018), 1–27. https://doi.org/10.1109/JIOT.2017.2773600

[61] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. 2008. Using components for architecture-based management: the self-repair case. In *Proc. of ICSE*. 101–110. https://doi.org/10.1145/1368088.1368103

[62] Vítor Estêvão Silva Souza. 2012. *Requirements-based Software System Adaptation*. Ph.D. Dissertation. University of Trento, Italy.

[63] Jacob Swanson, Myra B. Cohen, Matthew B. Dwyer, Brady J. Garvin, and Justin Firestone. 2014. Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In *Proc. of SIGSOFT*. 377–388. https://doi.org/10.1145/2635868.2635915

[64] Sven Tomforde. 2011. *An architectural framework for self-configuration and self-improvement at runtime*. Ph.D. Dissertation. University of Hannover. https://doi.org/10.15488/7766

[65] Sven Tomforde and Christian Müller-Schloer. 2013. Incremental Design of Adaptive Systems. *J. Ambient Intell. Smart Environ.* 6, 2 (2013), 179–198. https://doi.org/10.3233/AIS-140252

[66] Sebastian VanSyckel, Dominik Schäfer, Gregor Schiele, and Christian Becker. 2013. Configuration Management for Proactive Adaptation in Pervasive Environments. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 131–140. https://doi.org/10.1109/SASO.2013.28

[67] Thomas Vogel. 2018. *Model-Driven Engineering of Self-Adaptive Software*. Ph.D. Dissertation. University of Potsdam.

[68] Thomas Vogel and Holger Giese. 2013. *Model-driven engineering of adaptation engines for self-adaptive software : executable runtime megamodels*. Technical Report.

[69] Thomas Vogel and Holger Giese. 2014. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8, 4 (2014), 18:1–18:33. https://doi.org/10.1145/2555612

[70] Markus Weckesser, Malte Lochau, Michael Ries, and Andy Schürr. 2018. Mathematical Programming for Anomaly Analysis of Clafer Models. In *Proc. of MODELS*. 34–44. https://doi.org/10.1145/3239372.3239398

[71] Danny Weyns. 2019. Software Engineering of Self-adaptive Systems. In *Handbook of Software Engineering*. Springer, 399–443. https://doi.org/10.1007/978-3-030-00262-6_11

[72] Danny Weyns and M. Usman Iftikhar. 2019. ActivFORMS: A Model-Based Approach to Engineer Self-Adaptive Systems. *CoRR* abs/1908.11179 (2019). arXiv:1908.11179

[73] Juan Ye, Simon Dobson, and Susan McKeever. 2012. Situation identification techniques in pervasive computing: A review. *Pervasive Mob. Comput.* 8, 1 (2012), 36–66. https://doi.org/10.1016/j.pmcj.2011.01.004

[74] Edith Zavala, Xavier Franch, Jordi Marco, and Christian Berger. 2020. HAFLoop: An architecture for supporting Highly Adaptive Feedback Loops in self-adaptive systems. *Future Gener. Comput. Syst.* 105 (2020), 607–630. https://doi.org/10.1016/j.future.2019.12.026

[75] Marwin Züfle, André Bauer, Nikolas Herbst, Valentin Curtef, and Samuel Kounev. 2017. Telescope: A Hybrid Forecast Method for Univariate Time Series. In *Proc. of ITISE*.

[76] Marwin Züfle, André Bauer, Veronika Lesch, Christian Krupitzer, Nikolas Herbst, Samuel Kounev, and Valentin Curtef. 2019. Autonomic Forecasting Method Selection: Examination and Ways Ahead. In *Proc. of ICAC*. IEEE, 167–176. https://doi.org/10.1109/ICAC.2019.00028