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

12

Trends such as the Internet of Things lead to a growing number of networked devices and to a variety of communication systems. Adding self-adaptive capabilities to these communication systems is one approach to reducing administrative effort and coping with changing execution contexts. Existing frameworks can help reducing development effort but are neither tailored toward the use in communication systems nor easily usable without 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. REACT addresses heterogeneity and distribution aspects of such systems and reduces development effort. In this article, we propose REACT-ION—an extension of REACT for situation awareness. REACT-ION offers a context management module that is able to acquire, store, disseminate, and reason on context data. The context management module is the basis for (i) proactive adaptation with REACT-ION and (ii) self-improvement of the underlying feedback loop. REACT-ION can be used to optimize adaptation decisions at runtime based on the current situation. Therefore, it can cope with uncertainty and situations that were not foreseeable at design time. We show and evaluate in two 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, situation awareness

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. Auton. Adapt. Syst.* 15, 4, Article 12 (December 2021), 29 pages.
https://doi.org/10.1145/3487919

This work has been co-funded by the German Research Foundation (DFG) as part of project A4 within CRC 1053—MAKI.
Authors' addresses: M. Pfannemüller, M. Breitbach, and C. Becker, Universität Mannheim, Schloss, Mannheim, Germany, 68131; emails: {martin.pfannemueller, martin.breitbach, christian.becker}@uni-mannheim.de; M. Weckesser and A. Schürr, Technische Universität Darmstadt, Karolinenplatz 5, Darmstadt, Germany, 64289; emails: {markus.weckesser, andy.schuerr}@es.tu-darmstadt.de; B. Schmerl, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA, 15213; email: schmerl@cs.cmu.edu; C. Krupitzer, Universität Hohenheim, Fruwirthstr. 21, Stuttgart, Germany, 70599; email: christian.krupitzer@uni-hohenheim.de.

1556-4665/2021/12-ART12 $15.00
https://doi.org/10.1145/3487919

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 [61]. 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 [21]. 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 [36], SASSY [54], or MUSIC [38] 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 **Runtime Environment for Adapting Communication SysTems**[1] in Reference [67] and applied it in a demo case in Reference [66]. REACT supports domain experts in specifying adaptation behavior in a model-based fashion with Clafer [8] 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 [25, 90], we implemented REACT and made it available as an open source project.[2]

In this article, we present REACT-ION—an extension of REACT that additionally integrates features for providing situation awareness [28] as demanded for self-adaptive systems in Reference [33]. Situation awareness is "*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*" [28]. The core of REACT-ION is a context management module that is able to acquire context data, store it in a context model, and perform reasoning on the data. These features cover the *perception* aspect of situation awareness. In addition to its internal context reasoning capabilities, REACT-ION is able to disseminate context information to (external) components, which may reason on the data as well. Therefore, REACT-ION provides two options for the *comprehension* part of situation awareness. As far as *projection* is concerned, we show in this article how REACT-ION can be used for proactive adaptation. Proactive adaptation typically requires forecasts, i.e., projections

[1] This article is an extended version of previous work published 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.

of the status in the future. In addition, we use REACT-ION to adapt the underlying adaptation logic (known as self-improvement [48]) to tackle uncertainty [31]. More precisely, we show how it is possible with REACT-ION to make a situation-dependent choice of the feedback loop that is applied for an adaptation.

The remainder of this article is structured as follows. Section 2 reviews related work. Section 3 briefly outlines REACT's architecture. This section also introduces how REACT's implementation supports self-adaptivity. Section 4 proposes REACT-ION. It first presents the concept of situation awareness in more detail. 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. The section further describes how REACT-ION can be used for proactive adaptation and for self-improvement of adaptation logics. Section 5 first presents a comparative evaluation of REACT and Rainbow. Then, it describes experiments that evaluate REACT-ION and its capabilities for proactive adaptation and self-improvement. Finally, Section 6 summarizes our findings and outlines future work.

2 RELATED WORK

In this section, we review related work. In Section 2.1, we present approaches related to REACT, i.e., frameworks for engineering self-adaptive systems. In Section 2.2, we summarize approaches to situation awareness in self-adaptive systems. These are approaches particularly related to REACT-ION—the extension to REACT presented in this article.

2.1 Engineering Self-adaptive Systems

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 Reference [49]. Several related approaches perform adaptations based on architectural models (e.g., References [32, 61, 78]) or specify architecture definition languages for self-adaptive systems (e.g., References [23, 27, 51]). Model-based engineering approaches such as [11, 34, 59, 68] often use **Dynamic Software Product Lines (DSPLs)** with feature models. The models@run.time research proposes to use runtime models that represent the system and environment for reasoning [9, 14]. 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 [45]—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 [47] and HAFLoop [94], 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. In this regard, approaches that support multiple programming languages (e.g., Reference [52]) are easier to use. 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 [36] framework allow for connecting the target system easily to the

adaptation logic, which is especially important for legacy systems. Rainbow, however, belongs to the approaches [19, 20, 36, 54, 80] 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 [92]. 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 [48], or change the deployment at runtime. This is 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 [5, 19, 20, 36, 69, 79], 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 [5, 36, 47, 79, 91, 94] 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 [36] and Zanshin [79] have been compared in Reference [3].

Accordingly, in Reference [67], we proposed REACT, a reusable runtime environment for model-based adaptations in communication systems. REACT contributes to the state of the art thanks 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.

2.2 Situation Awareness

This section briefly summarizes research related to REACT-ION—the extension to REACT introduced in this article. It presents approaches on (i) context awareness, (ii) proactive and history-aware adaptation, and (iii) self-improvement and hybrid planning. We focus on implemented approaches excluding general architectures or methodologies.

2.2.1 Context Awareness. *Context* is "*any information that can be used to characterize the situation of an entity*" [1, p. 304]. The interested reader is referred to References [39, 50] for an overview of context-aware systems in general. Particularly relevant for REACT-ION is *context management*, i.e. the acquisition, modeling, reasoning, and dissemination of context [65]. Prominent approaches such as Aura [37], CARISMA [18], Gaia [71], or PROACTIVE [85] illustrate that context management is a major research focus in the pervasive and context-aware computing domain.

As far as self-adaptive systems are concerned, context awareness and, hence, context management are mostly covered implicitly. Being aware of the context, including the state of the environment and the system itself, is considered as a fundamental property of a self-adaptive system [74]. Context data is often collected via *sensor* interfaces, e.g., in References [36, 38, 47, 79]. The approach by Fredericks et al. [33] is an example where context is considered explicitly. The approach plans adaptations in the large state space of possible contexts using optimization. At runtime, specific situations are identified and used for finding optimal system configurations with clustering techniques. The authors apply the approach successfully in a real-world navigation scenario. In Reference [44], Képes et al. propose a system for modeling the context of software updates in cyber-physical systems. Based on the context, a situation-aware choice of the optimal time for updates is made. Nakahara et al. [60] propose CoSMOS, an approach for self-adaptive offloading

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

	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 [91]	●	●	●	●			●		●	●	
Cetina [20]		●	●								
EUREMA [87, 88]	●		●	●					●		
FESAS [47]	●		●	●		●	●	●	●	●	
Genie [10]	●	●	●				●		●		
GRAF [2]	●	●	●	●		●	●		●		
HAFLoop [94]	●		●	●		●	●	●	●	●	
KX [64]	●	●	●	●				●	●		
Malek [52]	●	●		●	●	●	●		●		
MOSES [19]		●	●								
MUSIC [38]	●	●		●		●	●		●		
Preisler [69]	●	●	●	●			●				
Rainbow [24, 36]	●	●	●	●		●				●	●
REFRACT [80]		●	●						●		
SASSY [54]	●	●	●	●					●		
StarMX [5]	●		●	●		●	●			●	
Tomforde [82]	●	●	●	●		●	●		●		
Zanshin [79]	●	●	●	●		●	●			●	●
REACT [67]	●	●	●	●	●	●	●	●	●	●	●

decisions. In their system, the offloading strategy of applications is adapted at runtime based on the context.

2.2.2 Proactive and History-Aware Adaptation. In this article, we show how REACT-ION's context management module can be used to perform *proactive adaptation* (cf. Section 4.2). Related to this, the PROACTIVE project [85] presents an approach for handling proactive system configurations in pervasive computing systems. A constraint satisfaction problem is solved that takes different application requirements into account. In Reference [16], Cámara et al. propose a formal model and algorithm that considers the time required for an adaptation—such as booting an additional server in a cloud scenario—for proactive adaptation. As an extension, Moreno et al. [55] also consider uncertainty and use model checking at runtime for making adaptation decisions. They apply a **Markov Decision Process (MDP)**. One drawback is that the MDP has to be constructed for each adaptation decision. Consequently, in Reference [56], the authors move a large portion of the MDP construction to the offline phase for increasing the adaptation speed. In Reference [77], Shin et al. propose PASTA—a proactive adaptation approach that uses statistical model checking. The approach tackles state explosion by providing an algorithm, a reference architecture, and an open-source implementation.

Closely related to proactive adaptation, *history-aware adaptation* describes approaches that are time-aware and include time series modeling for reasoning about the next adaptation [73, 84]. In Reference [35], García-Domínguez et al. argue that model-based approaches often use model evolution for integrating changes at runtime. The authors propose an extension of a model query language to keep track of model changes in a scalable way and to annotate "*situations of*

interest" [35]. This enables to easily query the system specifically for annotated situations and can be considered as an indexing approach. Additionally, the approach takes specific model evolution steps into account for determining "*time-aware patterns*" [35]. A time-aware pattern happens, e.g., if a hierarchical model node does not have children at one time stamp and has some children at another. Accordingly, the system enables to directly query such situations. Comparably, in Reference [73], Sakizloglou et al. propose a (memory-efficient) querying scheme for model history. Their approach has the advantage of enhancing a runtime model with history-aware capabilities and has a strong focus on scalability by including pruning strategies.

2.2.3 Self-Improvement and Hybrid Planning. *Self-improvement* is the "*adjustment of the adaptation logic to handle former unknown circumstances or changes*" [48, p. 2] in the environment or the target system. We review self-improvement approaches in Reference [48]. ActivFORMS [91] follows the ***Three-layer Architecture*** **(3LA)** proposed by Kramer and Magee in Reference [46]. The architecture includes an additional goal management layer that adapts the adaptation logic. Hence, the goal management layer represents the self-improvement mechanism. PLASMA [81] uses two models that capture the possible system states and the architecture of the target system. It also uses a specific layer for generating new plans in the case of changing high-level goals or failing components. FUSION [30] is a utility-based approach that adapts the knowledge base of the adaptation logic. The FESAS **Adaptation Logic Manager (ALM)** [72] encompasses an additional MAPE-K-based layer on top of an adaptation logic. This additional feedback loop performs self-improvement of the underlying adaptation logic.

In this article, we use REACT-ION for *hybrid planning*. Hybrid planning describes a form of self-improvement where the feedback loop for adaptation is chosen at runtime (among several alternatives). In References [62, 63], Pandey et al. specifically define the hybrid planning problem and investigate multiple planning alternatives for balancing adaptation quality and adaptation speed. EUREMA [88] proposes a model-based engineering approach for specifying adaptation logics. Additional trigger conditions can be used at runtime to select the feedback loop that should be executed. In Reference [43], the authors propose a reinforcement learning-based approach for improving the decision-making system of a robot, which is similar to hybrid planning. HAFLoop [94] extends FESAS [47] with the explicit option to adapt the adaptation logic component's parameters and structure in a reusable way. This also permits having multiple parallel adaptation logic components and restructuring them at runtime according to changing contexts and system goals.

3 REACT: A REUSABLE RUNTIME ENVIRONMENT FOR ADAPTIVE COMMUNICATION SYSTEMS

We briefly introduce REACT's architecture and internal feedback loop based on Reference [67] in this section, before we present REACT-ION—the extension for situation awareness—in Section 4.

3.1 REACT's Architecture

In contrast to self-adaptation frameworks that 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

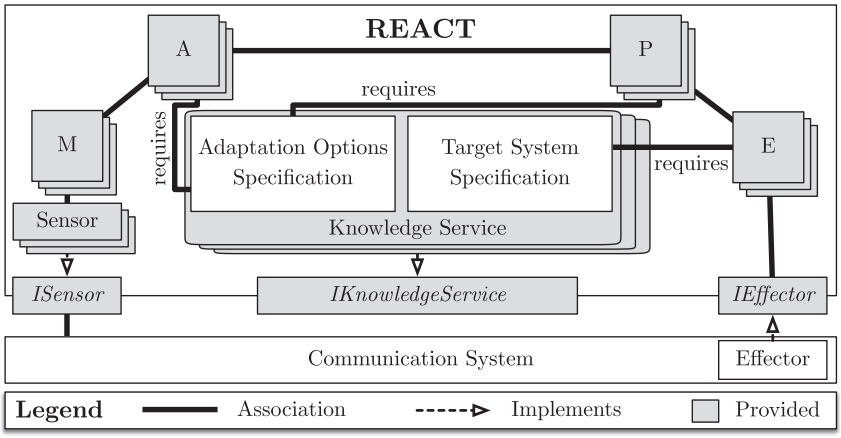


Fig. 1. REACT's architecture in a UML-like notation. It consists of one or multiple MAPE feedback loop(s) connected to instance(s) of the knowledge service with the *adaptation options specification* and *target system specification* provided by the domain expert. REACT's reusable feedback loop uses 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.

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 [45]. The feedback loop uses information stored in 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.

As models provide a sufficient level of expressiveness while being easy to use for domain experts, we selected a model-based approach for REACT's feedback loop. By creating the models at design time, the domain expert tailors the feedback loop to the respective use case. Thus, the domain expert is able to integrate self-adaptivity into the target system by only providing the models used as decision criteria. These models are then used by the readily provided internal feedback loop of the runtime environment. 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 [53]. 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. This may be necessary due to two reasons. First, complexity and uncertainty may lead to situations that were not foreseeable at design time [83]. Second, environmental changes may necessitate model changes. The `IKnowledgeService` interface thus allows, for instance, REACT to be connected to a self-improvement [48] 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 Reference [92], 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** (**CLAss, FEature, Reference**) [8]. There are multiple reasons to use Clafer. First, it is a well-established approach applied in different domains [4, 8], 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 large modeling experience. Third, Clafer provides model verification and validation [7]. By using Clafer, REACT offers the possibility for advanced static analysis as presented in Reference [89]. Thus, we can make sure that no contradictions exist in the Clafer specifications and that each possible sensor input leads to valid adaptation decisions.

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*

```
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.

commonalities" [4] 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 and 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.

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. In many cases, a UML model of a target system might already exist and be ready to use as a *target system specification* for REACT. This considerably decreases development effort. In addition, an automated creation of a UML model from source code can also reduce the time for modeling. REACT parses the UML class diagram as an XML file complying with 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

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

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 solves this problem specification as a **constraint-satisfaction problem (CSP)** with Chocosolver [70], 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 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*. 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 [41]. 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 [29]. 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

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

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 article, 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*" [28]. Accordingly, the three levels of situation awareness consist of (i) perception, (ii) comprehension, and (iii) prediction [28]. Situation awareness has already been applied in the ubiquitous and pervasive computing domain, mainly with a focus on the perception and comprehension levels (e.g., see Reference [93]). Fredericks et al. have studied and discussed the usefulness of situation awareness in self-adaptive systems in Reference [33]. 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 (i.e., self-improvement [48]) to tackle uncertainty [31]. Thus, we exemplarily apply hybrid planning with 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 integration of this context management module has two implications. First, it paves the way toward proactivity as well as self-improvement [48] 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, then REACT-ION may skip the planning process and perform the same adaptation again.

4.1.1 Architecture. In Reference [65], Perera et al. define the *context life cycle*, which describes how context information is processed in a context-aware system. The life cycle consists of four phases: *acquisition*, *modeling*, *reasoning*, and *dissemination*. The acquisition phase describes how the context data is gathered from several sources such as physical or virtual sensors. In the second phase (context modeling), this context data is transformed into a "meaningful" representation, i.e., the context model. This context model "*identifies a concrete subset of the context that is realistically attainable [...] and able to be exploited in the execution of the task*" [42]. Usually, a context model consists of multiple context attributes that contain an identifier, a type, a value, and optional properties. With the help of the context model, the system performs context reasoning—the third phase of the life cycle. In this phase, the system exploits the context model to gain new knowledge or improve its performance [13]. For instance, a situation-aware system may infer the current situation from the context model and adapt accordingly. In the final phase of the life cycle (dissemination),

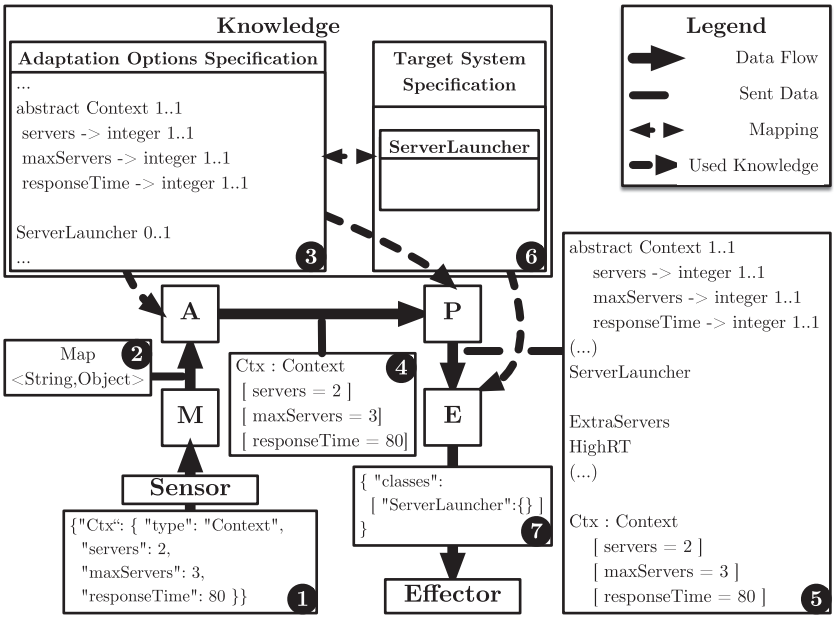


Fig. 2. An adaptation cycle of REACT for the cloud server management example. The analyzer maps the JSON-based sensor information to the *adaptation options specification* in Clafer. The planner evaluates the model and finds a valid instance. Here, it adds a `ServerLauncher` Clafer as starting a new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

the system distributes the context information to external components, either initiated by a query or a subscription.

We design REACT-ION's *Context Manager* along the four phases of the context life cycle. Thus, the context manager consists of four components with clear responsibilities. These four components are *Acquisition*, *Storage*, *Reminiscence*, and *Distribution*. They are responsible for context acquisition, modeling, reasoning, and dissemination, respectively. Figure 3 shows the architecture of REACT-ION including the context manager. In the following, we introduce how REACT-ION performs the four phases of the life cycle.

The context acquisition process starts at REACT-ION's `ISensor` interface that receives sensor data from the target system. This data may originate from multiple physical or virtual sensors. As REACT-ION is a generic runtime environment that is applicable to a wide range of use cases, it has to be able to collect context data from diverse sensors. To achieve this, REACT-ION includes the standardized `ISensor` interface. The data sources connect to REACT-ION via this interface and provide context data in JSON format according to the *adaptation options specification* (cf. Figure 2 and Section 3.2.2). The analyzer forwards the current context information to the knowledge service. The acquisition component of the context manager is now responsible for receiving the context information from the knowledge service and prepares it for storage in the context model.

The *Storage* component contains the context model. It covers the second phase of the context life cycle. Approaches for context modeling range from simple key-value pairs over model-based approaches (e.g., Reference [18]) to ontology-based approaches (e.g., Reference [71]). REACT-ION uses a database for context modeling, which allows for fast and simple storage and retrieval of large amounts of context information [65]. In addition to the context information, the database stores the corresponding adaptation decisions of REACT-ION.

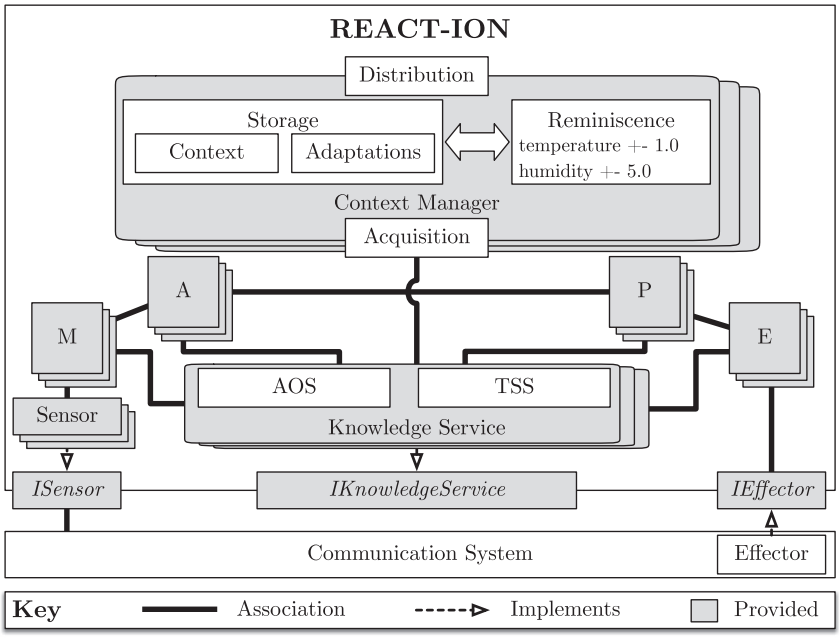


Fig. 3. Architecture of REACT-ION including the optional context manager module. The context manager encompasses four components: *Acquisition*, *Storage*, *Reminiscence*, and *Distribution*. AOS: *adaptation options specification*, TSS: *target system specification*.

As far as the third phase of the context life cycle (reasoning) is concerned, REACT-ION offers two options. 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, then the context is added to the context model 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 (cf. 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 information. For this purpose, REACT-ION is able to disseminate the context information to external components, which is the last phase of the context life cycle.

In REACT-ION, the *Distribution* component is responsible for context dissemination. This component communicates with external software that, e.g., reasons on the context information to detect the current situation. REACT-ION requires a flexible solution that is applicable in many use cases. Thus, the *Distribution* component is able to work both subscription- and query-based. It disseminates (new) context information and planned adaptations via a publish/subscribe system. External components may also query the *Storage* component—and, hence, also the context model—via this publish/subscribe system.

4.1.2 Implementation. For context acquisition, the context information 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 one or multiple tables for the context and one table for the 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. In the example shown in Figure 2, this would lead to a single context table named `Context` with three integer attributes representing the `servers`, `maxServers`, and `responseTime` attributes as columns. In addition to simple data types, the Clafer language also includes abstract Clafers that describe complex types, similar to classes in object-oriented programming. REACT-ION creates a separate table for each abstract Clafer if such Clafers are part of the context information. The separate tables include a foreign key relation to the main context table. The adaptations table contains an entry for each parameter of all components in the *target system specification*. A foreign key relation in the adaptations table references the corresponding context in the context table. In the example in Figure 2, the component column of the adaptations table would either be empty or contain the String *ServerLauncher*.

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 (cf. Section 3.1). This is beneficial in use cases where the frequency 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 (cf. Section 3.1), then 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 REACT-ION achieves proactive adaptation based on the context management module. Second, we show in Section 4.3 how context-awareness can be used as foundation for self-improvement with REACT-ION.

4.2 Proactive Adaptation with REACT-ION

Self-adaptive systems either perform *reactive* or *proactive* adaptation [49]. 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 [39, 49]. In general, for performing proactive adaptation, the system context has to be known [85]. REACT-ION's context management module covers this requirement. In this section, we show how proactive adaptation can be achieved with REACT-ION.

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

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 [96]. In Section 5.2, we therefore show how to connect a REACT-based self-adaptive system to the *Telescope* [95] 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 (cf. Section 3.1). When combining both, domain experts are able to modify the feedback loop based on the current situation, i.e. to apply self-improvement. Self-improvement is important as complexity and uncertainty may lead to situations that were not foreseeable at design time [83]. Examples for such situations include a significant change in 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 (cf. 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 (cf. 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. The approaches differ in their adaptation speed, ease of use, applicability for many use cases, and memory consumption. We propose to combine several reasoning approaches and to choose the suitable feedback loop based on the current situation. The reconfiguration behavior of a

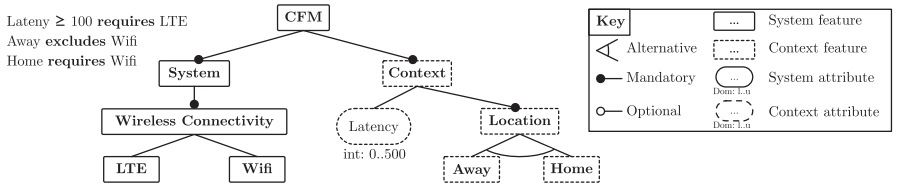


Fig. 4. Exemplary CFM that represents the reconfiguration space of the wireless connectivity of a smartphone. Depending on the constraints, it determines the selection of LTE or Wifi for providing wireless connectivity.

REACT-based system is modeled in Clafer and UML. Planning happens by solving a constraint-satisfaction problem (cf. 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)** [40, 75] for specifying the problem space. A CFM is a hierarchical tree-like model. It specifies the reconfiguration space of a self-adaptive system including the adaptations based on the 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, if using the Wifi connection results in a higher latency than 100 ms, then the LTE connection is enforced for decreasing the latency.

CFMs can be translated into Boolean **satisfiability (SAT)** problems or **mixed-integer linear programming (MILP)** problems [89]. 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 [76], 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 subtree 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 Reference [89].

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.

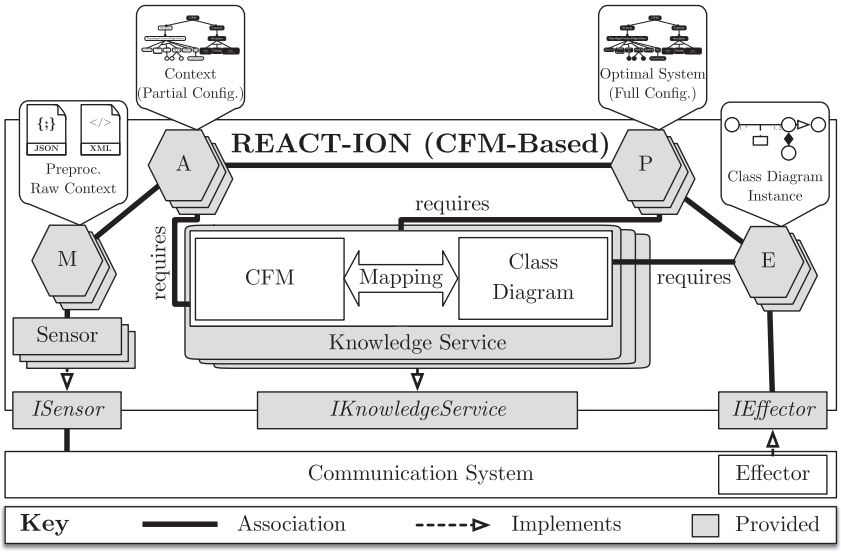


Fig. 5. Architecture and functionality of REACT-ION using the CFM-based feedback loop. The knowledge consists of a CFM representing the problem space, a class diagram representing the solution space, and an explicit mapping between both. The boxes attached to the MAPE functionalities show the results of each component [89]. The context manager is omitted here.

5 EVALUATION

We evaluate REACT and its extension REACT-ION in three experiments. First, Section 5.1 compares REACT with Rainbow [36]—a well-known and frequently applied framework for model-based adaptation. This section summarizes the essential findings from the evaluation of REACT in Reference [67]. Second, Section 5.2 builds upon the extensions of this article and evaluates proactive adaptations with REACT-ION in a smart grid use case. Third, Section 5.3 outlines the evaluation of applying different feedback loops as part of REACT-ION which enables the system to select feedback loops at runtime for self-improvement. Finally, Section 5.4 discusses potential threats to validity.

5.1 Comparison of REACT and Rainbow

In our first experiment, we compare REACT with the well-known Rainbow framework [36] in terms of development effort, performance, and features.

Rainbow—the baseline approach: 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 target system that update the model via gauges, which abstract and aggregate low-level information to detect architecture-relevant events and properties. The adaptation manager, on receiving the adaptation trigger, chooses the "best" adaptation 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 [24]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [23], is a multi-step pattern of adaptations in which each

Table 2. SLOC Measurements of the Modeling in Rainbow and REACT

Rainbow			REACT		
Artefact	**SLOC**	**Language**	**Artefact**	**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**	

step evaluates a set of condition-action pairs and executes an action, namely, a tactic, on the target system with variable execution time. 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 [15], predictive proactive adaptation [57], and human-machine cooperation [17].

SWIM—the use case: We deploy Rainbow and REACT with the SEAMS exemplar SWIM (Simulator for Web Infrastructure and Management) [58], which represents a cloud system. SWIM [58] offers a reproducible way for evaluating adaptation logics in a web server environment. 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 percentage of the requests with optional content is described as dimmer value. 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, represented by the dimmer value.

In this experiment, we compare Rainbow and REACT from the perspective of a domain expert. To provide the best experience for the domain expert, Rainbow and REACT should (i) be usable with low effort, (ii) lead to fast adaptations, and (iii) provide sufficient capabilities to introduce the desired self-adaptive behavior. Thus, we answer three research questions in the following that evaluate the (i) development effort (RQ1.1), (ii) performance (RQ1.2), and (iii) features (RQ1.3) of both Rainbow and REACT.

RQ1.1: How does REACT compare to the state of the art in terms of development effort?

As far as development effort is concerned, two metrics influence the domain expert's experience: the **source 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.

As shown in Table 2, we observe that specifying the adaptive behavior with REACT requires considerably fewer SLOC. The domain expert has to write 152 SLOC in two files with clear responsibilities. To achieve the same behavior with Rainbow, the domain expert has to write 593 SLOC in six files using various programming/specification languages. Next, we assess the development effort for the interface implementation (cf. Table 3). We measure 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

Table 3. SLOC Measurements of the Interface Implementations of Rainbow and REACT

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

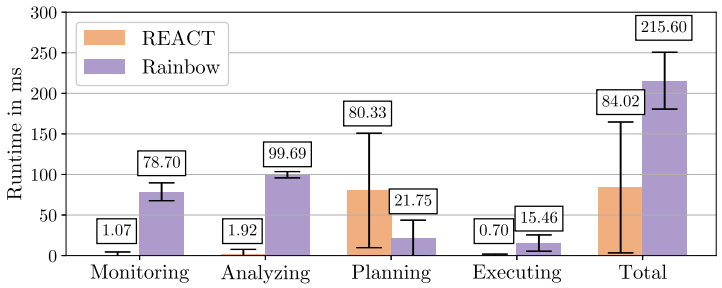


Fig. 6. Average run times of the MAPE activities of REACT and Rainbow.

use their preferred language. We acknowledge that SLOC as a metric might have its shortcomings, however, it is frequently applied as a metric to provide an estimation of the development effort (e.g., in References [24, 47, 86]).

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

We run the "1998 World Cup Web Site Access Logs" trace provided by SWIM 10 times with REACT and Rainbow. Figure 6 presents the average runtimes per MAPE activity as well as their average sum. REACT considerably outperforms Rainbow in the monitoring and analyzing phase with regards to execution time. The design considerations for Rainbow—to (i) hold an exact architecture model of the target system, (ii) update the model when new sensor data is available, (iii) periodically check for issues including an analysis where the problem is located in the model, and (iv) trigger an adaptation accordingly—allow a more complex analysis of the target system architecture at the cost of slower adaptations. 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 do REACT and Rainbow differ 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 is 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 overly 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, then REACT is a good candidate.

5.2 Proactive Adaptation

In the next experiments, we evaluate REACT-ION. First, we apply proactive adaptation with REACT-ION in a smart grid scenario as prediction and proactive adaptation is an important aspect for situation awareness [28].

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 require more power than available, then the required power is taken from the general power lines outside of the smart grid. Analogously, excess power is transferred to the surrounding power lines. A domain expert uses REACT-ION 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 min. We execute 30 runs with different household power profiles, each once with reactive adaptation and once with proactive adaptation.

We implement proactive adaptation based on REACT-ION's context management module as described in Section 4.2. We connect REACT-ION to Telescope [95]—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 min—is predicted.

In this experiment, we answer two research questions. First, we investigate whether proactive adaptation with REACT-ION leads to benefits compared to reactive adaptation (RQ2.1). As the overall goal of REACT-ION is to achieve situation awareness, we discuss how proactivity relates to situation awareness in this case study (RQ2.2).

RQ2.1: How does the system performance improve when applying proactive adaptations using REACT-ION?

Figure 7(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. Reducing the number of overloads is the primary goal in this experiment. We observe that proactive adaptation with REACT-ION is able to decrease the number of overloads from 230 to 203 on average in comparison to reactive adaptation. We therefore conclude that proactive adaptation is superior to reactive adaptation in this use case. Figure 7(b) shows the excess power produced in

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

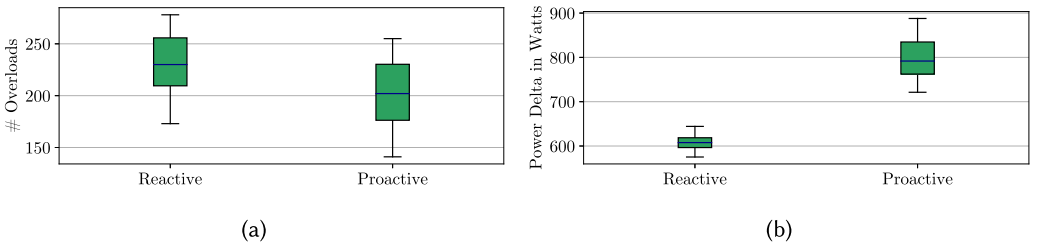


Fig. 7. Average number of overloaded simulation steps (a) and the average power delta (b) of the reactive and the proactive approach.

the smart grid. Since activating power sources increases the production step-wise and not continuously, proactive adaptation leads to more excess power.

In addition to proactive and reactive adaptation, it is also feasible to apply a static configuration with a fixed number of power sources. With such a static configuration, it is trivial to optimize either the number of overloads or the excess power. For instance, a high number of power sources may reduce the number of overloads to 0. The other extreme would be to produce no power at all, which would optimize the amount of excess power. Balancing the two goals with a static configuration, however, is not feasible due to two reasons. First, choosing a suitable static configuration manually is challenging itself. Second, such a configuration would be tailored to a certain situation and would not adapt to context changes. Overall, we therefore conclude that proactive adaptation with REACT-ION 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.

RQ2.2: How does proactive adaptation contribute to situation awareness?

According to Reference [28], 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 8 depicts an excerpt of an exemplary simulation run. The first 5 h 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. In these situations, it is important to decide if an adaptation decision should be reactive or proactive. When integrating proactive adaptation, the reliability of the predictions/forecasts is a relevant metric. If the predictions are not reliable in a specific situation, then reactive adaptation might still be the better option. The reason for this is that—in this case—the best known adaptation for the situation is performed rather than applying an adaptation for a situation that might not happen. This potentially decreases system performance even stronger as a reactive, delayed adaptation would impact performance. Further, reactive adaptation is required as backup for unknown situations. For the smart grid scenario, the current situation of COVID-19 lockdowns would be such an example, because it creates completely different situations—people stay at home at times when they would usually be at work/school—that the prediction/forecasting framework did not encounter and hence was not able to learn. Consequently, it is important to integrate a situation-aware choice whether to apply proactive or reactive adaptation, depending on the reliability of the prediction.

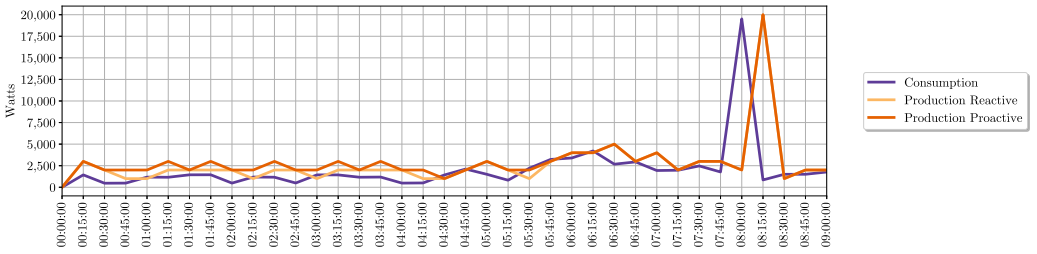


Fig. 8. Excerpt of the timeline of an exemplary run of the proactivity evaluation. Note: Starting at 5:45 reactive and proactive production are equal.

5.3 Self-Improvement with REACT-ION and Multiple Feedback Loops

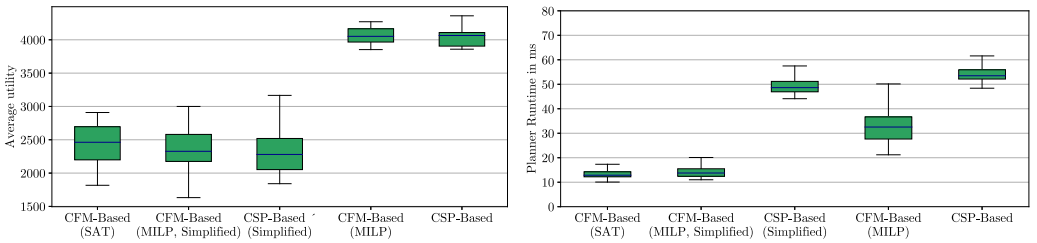
In this experiment, we deploy five MAPE-K feedback loops with different reasoning approaches as proposed in Section 4.3. Similar to Section 5.1, REACT-ION adapts a cloud server deployment provided by the SWIM exemplar in this case study.

Experimental Setup: We deploy five MAPE-K feedback loops with REACT-ION. These feedback loops either use the planning approach that solves a CSP (cf. Section 3) or the planning approach that uses a CFM as introduced in Section 4.3. The feedback loops differ in their solver implementations and in the specification of the reconfiguration options. The first feedback loop (`CFM-based (SAT)`) is CFM-based. It transforms the CFM into a Boolean satisfiability problem and solves it with the SAT4J [12] solver. The second feedback loop (`CFM-based (MILP, Simplified)`) is also CFM-based but transforms the CFM into a MILP problem. It solves the MILP problem with the CPLEX[9] solver. For reasons of comparability, we use the same problem specification for this feedback loop as for the first feedback loop. This means that several additional modeling features that the MILP solver is able to handle (in comparison to the SAT solver) are not used here. The third feedback loop (`CSP-based (Simplified)`) uses REACT's original reasoning approach that solves a CSP. Similar to the previous loop, we use the same problem specification as for `CFM-based (SAT)`. Again, this leaves several strengths of the CSP-based approach (e.g., extensive modeling capabilities) unused but makes it possible to compare the pure execution time of the planning approaches. The fourth feedback loop (`CFM-based (MILP)`) is CFM-based and transforms the CFM into a MILP problem. Thus, it is similar to the second feedback loop. However, we now apply a specification of the reconfiguration behavior that exhausts the additional capabilities and expressiveness of a MILP problem in comparison to a SAT problem. For instance, there is no restriction on Boolean variables only, which was required for the SAT solver. The fifth feedback loop (`CSP-based`) is CSP-based. In contrast to the third feedback loop, it uses a more sophisticated problem specification that exploits the additional features of Clafer.

In the scenario, the 30-min ClarkNet [26] trace provided with SWIM is used. Every run is repeated 20 times, and the context of the system is fetched every 10 s. By comparing the different feedback loops with regards to adaptation quality and adaptation speed (RQ3), we show the benefits of choosing the reasoning approach based on the current situation. Therefore, we measure the average planning time and the average utility. The utility is provided by SWIM and describes the quality of the adaptation decisions.

RQ3: Is self-improvement with REACT-ION able to combine the advantages of several reasoning approaches with regards to adaptation quality and adaptation speed?

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



(a) (b)

Fig. 9. Average utility per feedback loop type (a) and corresponding average planning times (b).

Figure 9 shows the results of the experiment with regards to the utility (Figure 9(a)) and planning time (Figure 9(b)). As far as the quality of the adaptation decisions is concerned, we observe in Figure 9(a) that `CFM-based (MILP)` and `CSP-based` considerably outperform the three other planning approaches that use the basic problem specification. We conclude that the improved modeling capabilities of `CFM-based (MILP)` and `CSP-based` lead to better adaptations. When comparing the remaining three planning approaches that use the same simple specification, we observe that `CFM-based (SAT)` leads to the highest utility. We argue that the faster adaptation is the reason for this. Figure 9(b) shows that `CFM-based (SAT)` leads to the fastest adaptations. It is 10 % faster than `CFM-based (MILP, Simplified)` and 73 % faster than `CSP-based (Simplified)`, even though they are using the same specification. The difference is even larger for more sophisticated specifications in `CFM-based (MILP)` and `CSP-based`. Therefore, the feedback loop that uses `CFM-based (SAT)` reacts more promptly to changes in the load and is thus able to achieve higher utilities than the other approaches with the simple specification. Still, more sophisticated specifications lead to even higher utility values.

Answering *RQ3*, we therefore conclude that there is a clear tradeoff between planning time and adaptation quality. If fast adaptations are required, then employing `CFM-based (SAT)` or a MILP solver with a restricted SAT-based specification (`CFM-based (MILP, Simplified)`) is possibly the better choice. This, however, comes at the cost of worse adaptation decisions in comparison to `CFM-based (MILP)` and `CSP-based`. As there is no planning approach that dominates the others with regards to both planning time and adaptation quality, the optimal choice depends on the current situation. With REACT-ION, such a change of the planning approach based on the situation at runtime is possible. REACT-ION's context management module distributes context information to external components that determine the current situation. Due to REACT-ION's options for self-improvement, it is possible to change the reasoning approach based on the determined situation at runtime. We therefore argue that situation awareness with REACT-ION increases the flexibility of planning. In situations where fast adaptations are important (e.g., in a critical state of the system), the system may use the fastest planning approach. When fast adaptations are less important, the system may in contrast use a more sophisticated planning approach to achieve a higher adaptation quality. A third option would be to use multiple loops at the same time, 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. The system may choose this option when the current situation allows the additional planning effort, e.g., in terms of memory footprint and computational load. We acknowledge that the integration of multiple feedback loops could lead to conflicts. Still, this may be a valuable option in a situation-aware system that chooses the planning approach depending on the current situation. Selecting, handling, and coordinating multiple feedback loops that run in parallel is an interesting field for future research.

5.4 Threats to Validity

We identify the following threats to validity for our evaluation results. 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 References [24, 47, 86]), a future user study with domain experts who apply REACT in different scenarios would strengthen validity. This work is further limited to a comparison with Rainbow. Future research may include a comparison to other frameworks such as SASSY [54] or StarMX [5] in additional use cases from the communication systems domain. As far as situation awareness with REACT-ION is concerned, only Telescope [95] as external prediction approach has been applied, and we omit a comparison of several approaches for prediction. Moreover, we focus in this work on compositional self-improvement. As future work, we aim at evaluating further situation-aware self-improvement options 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. Finally, we acknowledge that situation awareness, proactive adaptation, and self-improvement are broad terms. REACT-ION covers only a part of the spectrum that these terms potentially contain.

6 CONCLUSION

In this article, 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. REACT-ION extends REACT with a context management module, which can be used for providing situation awareness capabilities, executing proactive adaptations, and performing 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 specific 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 [22] instead of Clafer. As **verification and validation (V&V)** is an important research challenge [22, 25], 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 that further investigates the development effort. Focussing on such empirical evidence with practitioners has been identified as general challenge for further self-adaptive systems research [90]. For prediction, we integrated the Telescope [95] framework as an external prediction approach. We plan to investigate possibilities to provide a standardized solution for prediction/forecasts as part of REACT-ION itself. For example, it might be possible to integrate a recommendation system for time series forecasting (e.g., Reference [96]), which autonomously chooses the best suitable algorithm depending on the data characteristics.

ACKNOWLEDGMENTS

We thank Veronika Lesch (University of Würzburg) for the initial implementation of the Java-R-Bridge and our former students Marc Guldner, Erik Penther, and Johannes Schaum for their valuable contributions.

REFERENCES

[1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. 1999. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC'99)*. Springer, Berlin, 304–307. https://doi.org/10.1007/3-540-48157-5_29

[2] 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

[3] Konstantinos Angelopoulos, Vitor E. Silva Souza, and João Pimentel. 2013. Requirements and architectural approaches to adaptive software systems: A comparative study. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*. IEEE, Piscataway, NJ, 23–32. https://doi.org/10.1109/SEAMS.2013.6595489

[4] Michal Antkiewicz, Kacper Bak, Krzysztof Czarnecki, Zinovy Diskin, Dina Zayan, and Andrzej Wasowski. 2013. Example-driven modeling using clafer. In *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS'13)*. ACM/IEEE, New York, NY/Piscataway, NJ, 32–41.

[5] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. 2009. StarMX: A framework for developing self-managing Java-based systems. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09)*. IEEE, Piscataway, NJ, 58–67. https://doi.org/10.1109/SEAMS.2009.5069074

[6] Kacper Bak. 2013. *Modeling and Analysis of Software Product Line Variability in Clafer*. Ph.D. Dissertation. University of Waterloo, Ontario, Canada.

[7] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. 2010. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10)*. Springer, Berlin, Heidelberg, 102–122. https://doi.org/10.1007/978-3-642-19440-5_7

[8] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2016. Clafer: Unifying class and feature modeling. *Softw. Syst. Model.* 15, 3 (2016), 811–845. https://doi.org/10.1007/s10270-014-0441-1

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

[10] 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 *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 811–814. https://doi.org/10.1145/1368088.1368207

[11] 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 *Proceedings of the 12th International Software Product Lines Conference (SPLC'08)*. IEEE, Piscataway, NJ, 23–32.

[12] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *J. Satisfiabil. Boolean Model. Comput.* 7, 2-3 (2010), 59–6. https://doi.org/10.3233/SAT190075

[13] Antonis Bikakis, Theodore Patkos, Grigoris Antoniou, and Dimitris Plexousakis. 2007. A survey of semantics-based approaches for context reasoning in ambient intelligence. In *Proceedings of the European Conference on Ambient Intelligence (AmI'07)*. Springer, Berlin, 14–23. https://doi.org/10.1007/978-3-540-85379-4_3

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

[15] 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 *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15)*. ACM, New York, NY, 428–435. https://doi.org/10.1145/2695664.2695680

[16] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2014. Stochastic game analysis and latency awareness for proactive self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, New York, NY, 155–164. https://doi.org/10.1145/2593929.2593933

[17] Javier Cámara, Gabriel A. Moreno, and David Garlan. 2015. Reasoning about human participation in self-adaptive systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. ACM, New York, NY, 146–156. https://doi.org/10.1109/SEAMS.2015.14

[18] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. 2003. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Softw. Eng.* 29, 10 (2003), 929–945. https://doi.org/10.1109/TSE.2003.1237173

[19] 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. Softw. Eng.* 38, 5 (2012), 1138–1159. https://doi.org/10.1109/TSE.2011.68

[20] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. 2008. A model-driven approach for developing self-adaptive pervasive systems. In *Proceedings of the 3rd Workshop on Models@run.time*. 97–106.

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

[22] Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. 2011. Using models at runtime to address assurance for self-adaptive systems. In *Models@run.time—Foundations, Applications, and Roadmaps*. Springer, Berlin, 101–136. https://doi.org/10.1007/978-3-319-08915-7_4

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

[24] Shang-Wen Cheng. 2004. *Rainbow: Cost-effective software architecture-based self-adaptation*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA.

[25] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, Joao Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. 2010. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, Berlin, 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

[26] John Dilley. 1996. Web server workload characterization. *HP Lab. Tech. Report* 24 (1996), 1–16. https://doi.org/10.1145/233008.233034

[27] Jim Dowling and Vinny Cahill. 2001. The K-component architecture meta-model for self-adaptive software. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION'01)*. Springer, Berlin, 81–88. https://doi.org/10.1007/3-540-45429-2_6

[28] 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

[29] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. 2007. iPOJO: An extensible service-oriented component framework. In *Proceedings of the International Conference on Services Computing (SCC'07)*. IEEE, Piscataway, NJ, 474–481. https://doi.org/10.1109/SCC.2007.74

[30] Naeem Esfahani, Ahmed M. Elkhodary, and Sam Malek. 2013. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Trans. Softw. Eng.* 39, 11 (2013), 1467–1493. https://doi.org/10.1109/TSE.2013.37

[31] 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, Berlin, 214–238. https://doi.org/10.1007/978-3-642-35813-5_9

[32] Jacqueline Floch, Svein O. Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. 2006. Using architecture models for runtime adaptability. *IEEE Softw.* 23, 2 (2006), 62–70. https://doi.org/10.1109/MS.2006.61

[33] Erik M. Fredericks, Ilias Gerostathopoulos, Christian Krupitzer, and Thomas Vogel. 2019. Planning as optimization: Dynamically discovering optimal configurations for runtime situations. In *Proceedings of the 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'19)*. IEEE, Piscataway, NJ, 1–10. https://doi.org/10.1109/saso.2019.00010

[34] Nadia Gámez, Lidia Fuentes, and José M. Troya. 2015. Creating self-adapting mobile systems with dynamic software product lines. *IEEE Softw.* 32, 2 (2015), 105–112. https://doi.org/10.1109/MS.2014.24

[35] Antonio García-Domínguez, Nelly Bencomo, Juan Marcelo Parra Ullauri, and Luis Hernán García Paucar. 2019. Querying and annotating model histories with time-aware patterns. In *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS'19)*. ACM/IEEE, New York, NY/Piscataway, NJ, 194–204. https://doi.org/10.1109/MODELS.2019.000-2

[36] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Comput.* 37, 10 (2004), 46–54. https://doi.org/10.1109/MC.2004.175

[37] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste. 2002. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Comput.* 1, 2 (2002), 22–31. https://doi.org/10.1109/MPRV.2002.1012334

[38] Sven Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George A. Papadopoulus. 2012. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *J. Syst. Softw.* 85, 12 (2012), 2840–2859. https://doi.org/10.1016/j.jss.2012.07.052

[39] Marcus Handte, Gregor Schiele, Verena Majuntke, Christian Becker, and Pedro José Marrón. 2012. 3PC: System support for adaptive peer-to-peer pervasive computing. *ACM Trans. Autonom. Adapt. Syst.* 7, 1 (2012), 1–19. https://doi.org/10.1145/2168260.2168270

[40] 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'08)*. IEEE, Piscataway, NJ, 12–21. https://doi.org/10.1109/SPLC.2008.15

[41] Michi Henning. 2004. A new approach to object-oriented middleware. *IEEE Internet Comput.* 8, 1 (2004), 66–75. https://doi.org/10.1109/MIC.2004.1260706

[42] Karen Henricksen. 2003. *A framework for context-aware pervasive computing applications*. Ph.D. Dissertation. The University of Queensland, Brisbane, Australia.

[43] Christopher-Eyk Hrabia, Patrick Marvin Lehmann, and Sahin Albayrak. 2019. Increasing self-adaptation in a hybrid decision-making and planning system with reinforcement learning. In *Proceedings of the 43rd Annual Computer Software and Applications Conference (COMPSAC'19)*. IEEE, Piscataway, NJ, 469–478. https://doi.org/10.1109/COMPSAC.2019.00073

[44] Kálmán Képes, Frank Leymann, and Michael Zimmermann. 2020. Situation-aware updates for cyber-physical systems. In *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC'20)*. Springer, Berlin, Heidelberg, 12–32. https://doi.org/10.1007/978-3-030-64846-6_2

[45] 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

[46] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: An architectural challenge. In *Proceedings of the Future of Software Engineering Symposium (FOSE'07)*. IEEE, Piscataway, NJ, 259–268. https://doi.org/10.1109/FOSE.2007.19

[47] 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 *Proceedings of the International Conference on Autonomic Computing (ICAC'16)*. IEEE, Piscataway, NJ, 15–24. https://doi.org/10.1109/ICAC.2016.49

[48] Christian Krupitzer, Felix Maximilian Roth, Martin Pfannemüller, and Christian Becker. 2016. Comparison of approaches for self-improvement in self-adaptive systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'16)*. IEEE, Piscataway, NJ, 308–314. https://doi.org/10.1109/ICAC.2016.18

[49] 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

[50] Brian Y. Lim and Anind K. Dey. 2010. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th International Conference on Ubiquitous Computing (UbiComp'10)*. ACM, New York, NY, 13–22. https://doi.org/10.1145/1864349.1864353

[51] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. 1995. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*. Springer, Berlin, Heidelberg, 137–153. https://doi.org/10.1007/3-540-60406-5_12

[52] Sam Malek, George Edwards, Yuriy Brun, Hossein Tajalli, Joshua Garcia, Ivo Krka, Nenad Medvidovic, Marija Mikic-Rakic, and Gaurav S. Sukhatme. 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

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

[54] Daniel A. Menascé, Hassan Gomaa, Sam Malek, and João Pedro Sousa. 2011. SASSY: A framework for self-architecting service-oriented systems. *IEEE Softw.* 28, 6 (2011), 78–85. https://doi.org/10.1109/MS.2011.22

[55] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. 2015. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 1–12. https://doi.org/10.1145/2786805.2786853

[56] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. 2016. Efficient decision-making under uncertainty for proactive self-adaptation. In *Proceedings of the International Conference on Autonomic Computing (ICAC'16)*. IEEE, Piscataway, NJ, 147–156. https://doi.org/10.1109/ICAC.2016.59

[57] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. 2018. Flexible and efficient decision-making for proactive latency-aware self-adaptation. *ACM Trans. Autonom. Adapt. Syst.* 13, 1 (2018), 3:1–3:36. https://doi.org/10.1145/3149180

[58] 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 *Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'18)*. ACM, New York, NY, 137–143. https://doi.org/10.1145/3194133.3194163

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

[60] Flávio Akira Nakahara and Delano Medeiros Beder. 2018. A context-aware and self-adaptive offloading decision support model for mobile cloud computing system. *J. Ambient Intell. Humaniz. Comput.* 9, 5 (2018), 1561–1572. https://doi.org/10.1007/s12652-018-0790-7

[61] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. 1999. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst. Appl.* 14, 3 (1999), 54–62. https://doi.org/10.1109/5254.769885

[62] Ashutosh Pandey, Gabriel A. Moreno, Javier Cámara, and David Garlan. 2016. Hybrid planning for decision making in self-adaptive systems. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO'16)*. IEEE, Piscataway, NJ, 130–139. https://doi.org/10.1109/SASO.2016.19

[63] Ashutosh Pandey, Ivan Ruchkin, Bradley R. Schmerl, and Javier Cámara. 2017. Towards a formal framework for hybrid planning in self-adaptation. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17)*. IEEE, Piscataway, NJ, 109–115. https://doi.org/10.1109/SEAMS.2017.14

[64] Janak J. Parekh, Gail E. Kaiser, Philip Gross, and Giuseppe Valetto. 2006. Retrofitting autonomic capabilities onto legacy systems. *Cluster Comput.* 9, 2 (2006), 141–159. https://doi.org/10.1007/s10586-006-7560-6

[65] Charith Perera, Arkady B. Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. Context aware computing for the Internet of Things: A survey. *IEEE Commun. Surv. Tut.* 16, 1 (2014), 414–454. https://doi.org/10.1109/SURV.2013.042313.00197

[66] Martin Pfannemüller, Martin Breitbach, Christian Krupitzer, Christian Becker, and Andy Schürr. 2020. 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'20)*. IEEE, Piscataway, NJ, 228–229. https://doi.org/10.1109/ACSOS-C51401.2020.00062

[67] 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 *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS'20)*. IEEE, Piscataway, NJ, 65–74. https://doi.org/10.1109/ACSOS49614.2020.00027

[68] Martin Pfannemüller, Christian Krupitzer, Markus Weckesser, and Christian Becker. 2017. A dynamic software product line approach for adaptation planning in autonomic computing systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'17)*. IEEE, Piscataway, NJ, 247–254. https://doi.org/10.1109/ICAC.2017.18

[69] Thomas Preisler, Tim Dethlefs, and Wolfgang Renz. 2015. Middleware for constructing decentralized control in self-organizing systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'15)*. IEEE, Piscataway, NJ, 325–330. https://doi.org/10.1109/ICAC.2015.56

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

[71] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. 2002. A middleware infrastructure for active spaces. *IEEE Pervasive Comput.* 1, 4 (2002), 74–83. https://doi.org/10.1109/MPRV.2002.1158281

[72] Felix Maximilian Roth, Christian Krupitzer, and Christian Becker. 2015. Runtime evolution of the adaptation logic in self-adaptive systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'15)*. IEEE, Piscataway, NJ, 141–142. https://doi.org/10.1109/ICAC.2015.20

[73] Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. 2020. A scalable querying scheme for memory-efficient runtime models with history. In *Proceedings of the 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS'20)*. ACM, New York, NY, 175–186. https://doi.org/10.1145/3365438.3410961

[74] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Trans. Autonom. Adapt. Syst.* 4, 2 (2009), 1–42. https://doi.org/10.1145/1516533.1516538

[75] 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'13)*. ACM, New York, NY, 106–113. https://doi.org/10.1145/2499777.2500716

[76] Thomas Schnabel, Markus Weckesser, Roland Kluge, Malte Lochau, and Andy Schürr. 2016. CardyGAn: Tool support for cardinality-based feature models. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'16)*. ACM, New York, NY, 33–40. https://doi.org/10.1145/2866614.2866619

[77] Yong-Jun Shin, Eunho Cho, and Doo-Hwan Bae. 2021. PASTA: An efficient proactive adaptation approach based on statistical model checking for self-adaptive systems. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science, Vol. 12649)*. Springer, Berlin, 292–312. https://doi.org/10.1007/978-3-030-71500-7_15

[78] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. 2008. Using components for architecture-based management: The self-repair case. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 101–110. https://doi.org/10.1145/1368088.1368103

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

[80] 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 *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 377–388. https://doi.org/10.1145/2635868.2635915

[81] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. 2010. PLASMA: A plan-based layered architecture for software model-driven adaptation. In *Proceedings of the International Conference on Automated Software Engineering (ASE'10)*. ACM, New York, NY, 467–476. https://doi.org/10.1145/1858996.1859092

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

[83] 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

[84] Juan Marcelo Parra Ullauri, Antonio García-Domínguez, Luis Hernán García Paucar, and Nelly Bencomo. 2020. Temporal models for history-aware explainability. In *Proceedings of the 12st System Analysis and Modelling Conference (SAM'20)*. ACM, New York, NY, 155–164. https://doi.org/10.1145/3419804.3420276

[85] 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'13)*. IEEE, Piscataway, NJ, 131–140. https://doi.org/10.1109/SASO.2013.28

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

[87] Thomas Vogel and Holger Giese. 2012. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*. IEEE, Piscataway, NJ, 129–138. https://doi.org/10.1109/SEAMS.2012.6224399

[88] Thomas Vogel and Holger Giese. 2014. Model-driven engineering of self-adaptive software with EUREMA. *ACM Trans. Autonom. Adapt. Syst.* 8, 4 (2014), 1–33. https://doi.org/10.1145/2555612

[89] Markus Weckesser, Malte Lochau, Michael Ries, and Andy Schürr. 2018. Mathematical programming for anomaly analysis of clafer models. In *Proceedings of the 21st International Conference on Model Driven Engineering Languages and Systems (MODELS'18)*. ACM, New York, NY, 34–44. https://doi.org/10.1145/3239372.3239398

[90] Danny Weyns. 2019. Software engineering of self-adaptive systems. In *Handbook of Software Engineering*. Springer, Berlin, 399–443. https://doi.org/10.1007/978-3-030-00262-6_11

[91] Danny Weyns and M. Usman Iftikhar. 2019. ActivFORMS: A model-based approach to engineer self-adaptive systems. Retrieved from https://arXiv:1908.11179.

[92] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. 2010. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, Berlin, 76–107. https://doi.org/10.1007/978-3-642-35813-5_4

[93] 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

[94] 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

[95] Marwin Züfle, André Bauer, Nikolas Herbst, Valentin Curtef, and Samuel Kounev. 2017. Telescope: A hybrid forecast method for univariate time series. In *Proceedings of the International Work-Conference on Time Series Analysis*.

[96] 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 *Proceedings of the International Conference on Autonomic Computing (ICAC'19)*. IEEE, Piscataway, NJ, 167–176. https://doi.org/10.1109/ICAC.2019.00028

Received January 2021; revised August 2021; accepted September 2021