REACT: A Model-Based Runtime Environment for Adapting Communication Systems

Martin Pfannemüller*, Martin Breitbach*, Christian Krupitzer‡, Markus Weckesser†, Christian Becker*, Bradley Schmerl§, Andy Schürr†
*Universität Mannheim, Germany
Email: {martin.pfannemueller, martin.breitbach, christian.becker}@uni-mannheim.de
†Technische Universität Darmstadt, Germany
Email: {markus.weckesser, andy.schuerr}@es.tu-darmstadt.de
‡Universität Würzburg, Germany
Email: christian.krupitzer@uni-wuerzburg.de
§Carnegie Mellon University, USA
Email: schmerl@cs.cmu.edu

Abstract—**Trends such as the Internet of Things or edge computing lead to a growing number of networked devices. Hence, it is becoming increasingly important to manage communication systems at runtime. Adding self-adaptive capabilities is one approach to reduce administrative effort and cope with changing execution contexts. Existing frameworks for building self-adaptive software can help to reduce development effort in general. Yet, they are neither tailored towards the use in communication systems nor easily usable without profound knowledge in self-adaptive systems development. In this paper, we propose REACT, a reusable, model-based runtime environment to complement communication systems with adaptive behavior. It addresses the heterogeneity and distribution aspects of networks and reduces development effort. REACT empowers developers of communication systems to add adaptive behavior without having experience in self-adaptive systems development. We show the effectiveness and efficiency of our prototype in an experimental evaluation based on two distinct use cases from the communication systems domain: cloud resource management and software-defined networking. The first use case includes a comparison with Rainbow, which represents a state-of-the-art model-based framework for building self-adaptive systems. The second use case applies REACT in a sophisticated, real-world communication system scenario.**

I. 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 [1]. 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 [2]. 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 [3], SASSY [4], or MUSIC [5] 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 propose REACT, a **R**untime **E**nvironment for **A**dapting **C**ommunication Sys**T**ems. REACT allows domain experts to specify adaptation behavior in a model-based fashion with Clafer [6] 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 [7, 8], we implement REACT, make it available as an open source project[1], and guide domain experts with a well-defined development process. We evaluate REACT by (i) comparing it with the state-of-the-art Rainbow framework in a cloud resource management scenario and (ii) applying it in a real-world use case from the SDN domain.

[1]Available here: https://github.com/martinpfannemueller/REACT.

978-1-7281-7277-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ACSOS49614.2020.00027

The remainder of this paper is structured as follows. Section II reviews related work. Section III outlines the challenges we address with our approach as well as REACT's architecture. This is followed by Section IV that introduces how REACT's implementation supports self-adaptivity. Section V presents the development process to apply REACT. Section VI evaluates our approach in two distinct use cases comparing it to the state of the art. Finally, Section VII summarizes our findings and outlines future work.

II. Related Work

Engineering of self-adaptive systems is a prominent research area with a large body of excellent related work that we can build upon. We review the research landscape in [9]. Several related approaches perform adaptations based on architectural models (e.g., [1, 10, 11]) or specify architecture definition languages for self-adaptive systems (e.g., [12–14]). Model-based engineering approaches such as [15–18] often use DSPLs with feature models. The models@run.time research proposes to use runtime models that represent the system and environment for reasoning [19, 20]. 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 I.

TABLE I
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 [33]	●	●	●	●			●			●	
Cetina [25]		●	●								
FESAS [22]	●		●	●		●	●	●	●	●	
Genie [35]	●	●	●				●		●		
GRAF [36]	●	●	●	●		●	●		●		
HAFLoop [23]	●		●	●		●	●	●	●	●	
Malek [24]	●	●		●	●	●	●		●		
MOSES [26]		●	●								
MUSIC [5]	●	●		●		●	●		●		
Preisler [30]	●	●	●	●			●				
Rainbow [3, 37]	●	●	●	●		●				●	●
REFRACT [27]		●	●						●		
SASSY [4]	●	●	●	●					●		
StarMX [31]	●		●	●		●	●			●	
Tomforde [38]	●	●	●	●		●	●		●		
Zanshin [32]	●	●	●	●		●	●			●	●
REACT	●	●	●	●	●	●	●	●	●	●	●

First, an approach that optimally assists domain experts should support all self-* properties [21]—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 [22] and HAFLoop [23] for instance, provide excellent support with reusable MAPE components, but do not integrate a decision engine.

We aim to support the domain expert during the development process. Especially in the heterogeneous communication systems landscape, an approach is easy to use if it supports multiple programming languages such as the approach by Malek *et al.* [24]. 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 [3] framework allow connecting the target system easily to the adaptation logic, which is especially important for legacy systems. Rainbow, however, belongs to the approaches [3, 4, 25–27] 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 [28]. 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 [29], or change the deployment at runtime. This holds true for communication systems in particular, where, e.g., new components or subsystems may join or leave the system at any time. In several related approaches [3, 25, 26, 30–33], 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 [3, 22, 23, 31–33] 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 [3] and Zanshin [32] have been compared in [34].

In this paper, we propose REACT, a reusable runtime environment for model-based adaptations in communication systems. REACT contributes to the state of the art due to its focus on communication systems and domain expert support. None of the existing approaches offers multi-language support, enables decentralized control as well as distributed deployments, and is available as an open source project. We make the source code of REACT's implementation available and compare our approach with Rainbow.

III. REACT- A Reusable Runtime Environment for Adaptive Communication Systems

In this section, we first outline the challenges that shape REACT's design. Second, we introduce REACT's architecture.

A. Challenges

Introducing self-adaptivity to modern communication systems poses challenges regarding *distribution and decentralization*, *heterogeneity*, and *development effort*. We design REACT to address these challenges.

C1 Distribution & Decentralization: First, we adapt communication systems, i.e., inherently distributed systems. Thus, REACT has to collect sensor information from various distributed components and subsystems. Since adaptation may affect different parts of the distributed system, REACT must identify and adapt the affected parts. Second, REACT must support decentralized control, which has been identified as an open research challenge recently [7]. Typically, decentralized feedback loops are deployed distributed [28]. This makes it possible to share the computational overhead of self-adaptivity across multiple nodes.

C2 Heterogeneity: We face heterogeneity in two ways. First, REACT must be applicable to a wide range of heterogeneous communication systems. Therefore, domain experts must be able to use their preferred language to connect their (legacy) communication system to REACT. Second, the specification of the adaptation must be independent from the target communication system. This allows REACT to apply the same self-adaptive behavior to different target systems easily.

C3 Development Effort: Building self-adaptive systems is still a complicated task [9]. For adding adaptive behavior to a system, domain experts require specific knowledge, e.g., about feedback loops and adaptation planning. In practice, a domain expert typically does not have the required expertise. Thus, REACT simplifies and minimizes the development effort. This comprises a clear and easy, model-based development process with low programming overhead.

B. REACT's Architecture

In contrast to self-adaptation frameworks which offer a standard way to build self-adaptive applications, we refer to REACT as a runtime environment, i.e., a platform that is additionally able to plan and execute adaptations based on user-specified adaptation behavior. REACT includes a feedback loop as well as interfaces for connecting target systems. Potential target systems in the communication systems domain are overlay networks such as peer-to-peer systems and underlay networks, e.g., in SDN scenarios. However, REACT could possibly be used in other application domains as well. The feedback loop follows the MAPE-K architecture that consists of components for (i) **M**onitoring the system and the environment, (ii) **A**nalyzing the monitored data for necessary adaptations, (iii) **P**lanning the adaptations, and (iv) **E**xecuting the adaptations in the target system as well as (v) a shared **K**nowledge base [21]. The feedback loop uses information stored in an instance of the knowledge for reasoning. It

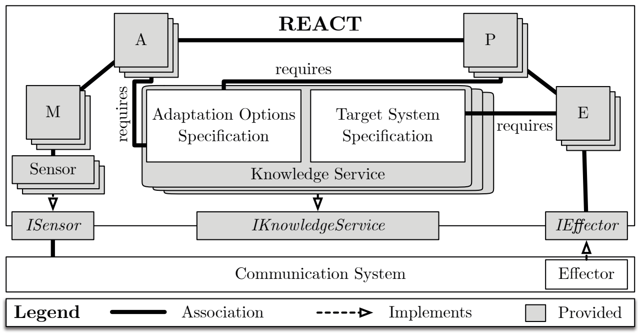

Fig. 1. REACT's architecture in a UML-like notation. It consists of one or multiple MAPE feedback loop(s) connected to instance(s) of the knowledge service with the adaptation options specification and target system specification provided by the domain expert. REACT's reusable feedback loop uses the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

receives sensor information from the communication system as an input and determines the required adaptations as an output via interfaces (Challenge C2). 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 (Challenge C3). The white boxes represent the specifications and the effector implementation that have to be provided by the domain expert.

Considering the specifications, self-adaptive systems can use models, rules, goals, utility functions, or combinations of the former as decision criteria [9]. As models provide a sufficient level of expressiveness while being easy to use for domain experts, we select 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 [39]. The separation of the two models decouples the specification of the reconfiguration behavior from the target system and its architecture. Separating the models is largely inspired by models@run.time [19] research, which distinguishes between problem and solution space. 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 (Challenge C1). 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 [40]. Second, environmental changes may necessitate model changes. The `IKnowledgeService` interface thus allows, for instance, REACT to be connected to a self-improvement [29] 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, we achieve high scalability and allow distributed deployments and decentralized control. Fully decentralized or hybrid patterns, as described in [28], are realizable (Challenge C1).

IV. Enabling Self-Adaptivity with REACT

In this section, we present 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.

A. Modeling

An essential part of REACT are the models of the adaptation behavior (adaptation options specification) and of the target system (target system specification). The domain expert provides these models at design time and may update them at runtime. REACT uses the models at runtime to adapt the target system. REACT supports adaptation options specifications in the structural specification language Clafer (**cla**ss, **fe**ature, **r**eference) [6]. There are multiple reasons to use Clafer. First, it is a well-established approach applied in different domains [6, 41], 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 [42]. By using Clafer, REACT offers the possibility for advanced static analysis as presented in [43]. 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 commonalities" [41] like a class in object-oriented programming. Hence, a Clafer can inherit from an abstract Clafer and use abstract Clafers like a type. The lines 2-5 describe an abstract entity of type *Context* with integer attributes. A solution of this problem space requires to have exactly one instance of this Clafer with all attributes set. Lines 6 and 7 define the auxiliary Clafers `ExtraServers` and `HighRT` that state whether it is possible to start an additional server and whether the response time is high. In addition, a Clafer model may contain constraints in brackets. Lines 8-9 specify constraints that set the auxiliary Clafers `ExtraServers` and `HighRT` according to the context. Line 10 is the adaptation rule stating that the `ServerLauncher` Clafer should be present in a model instance if the response time is high and more servers are available.

```
 1  ServerLauncher 0..1
 2  abstract Context 1
 3      servers −> integer 1
 4      maxServers −> integer 1
 5      responseTime −> integer 1
 6  ExtraServers 0..1
 7  HighRT 0..1
 8  [ 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.

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 to the *UML 2 Abstract Syntax Metamodel* by the Object Management Group. Due to this standardized format, the domain expert can create the XML file manually or use a graphical editor that offers an export in this format such as Papyrus[2]. 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.

B. Integrated Feedback Loop

The previous section describes the modeling of the adaptation options specification in Clafer and the target system specification in UML. Now, we show how REACT autonomously leverages these use case dependent models to achieve self-adaptivity. Figure 2 shows the behavior of REACT's integrated MAPE-K feedback loop in the aforementioned cloud server management example. The feedback loop starts as soon as new sensor information is received via the sensor interface in JSON format. In the example, this sensor data ❶ is context information about the cloud system. The received information is handed over to the monitoring component.

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

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

REACT's planner merges the generated Clafers with the adaptation options specification to the problem specification. The problem specification thus contains the global constraints of the adaptation options specification and the current constraints imposed by the sensor data. Now, REACT solves this problem specification as a constraint-satisfaction problem with

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

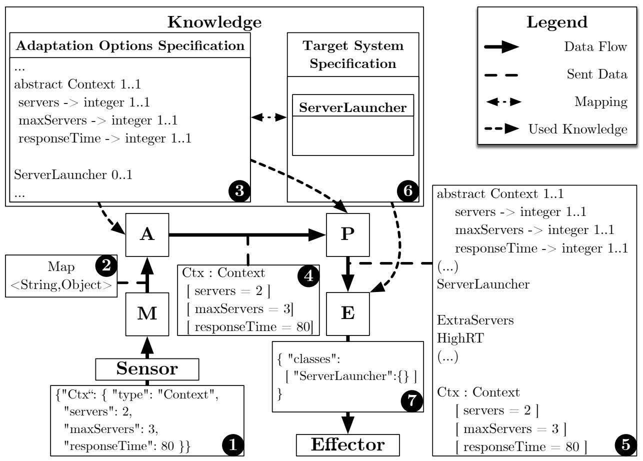

Fig. 2. An adaptation cycle of REACT for the cloud server management example. The analyzer maps the JSON-based sensor information to the adaptation options specification in Clafer. The planner evaluates the model and finds a valid instance. Here, it adds a `ServerLauncher` Clafer as starting a new server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

Chocosolver [44], a Java-based library for constraint programming. Hence, the solver finds a model instance ❺ that satisfies all constraints. In the exemplary case, this model instance would either contain or not contain the `ServerLauncher` Clafer, which constitutes the adaptation decision.

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

C. 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 [45]. 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 [46]. 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[3] registry for automatic setup, or manual IP address and port specifications.

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

V. REACT's Development Process

Defining a process for development, deployment, and operation is a major challenge in the self-adaptive systems domain [8]. This section shows how a domain expert can use REACT with a three-step development process. Figure 3 depicts the three development steps: 1) Modeling, 2) Connecting, and 3) Configuring.

1. Modeling		**2. Connecting**		**3. Configuring**	
1.1	Adaptation Options Specification	**2.1**	Effector Implementation	**3.1**	Key-Value-Based Configuration Files
1.2	Target System Specification	**2.2**	Sensor Implementation		

Fig. 3. Development process for using REACT. After modeling the self-adaptive behavior, the domain expert connects REACT to the target system. With a simple configuration step, the domain expert is able to deploy REACT.

In the first step, the domain expert creates the adaptation options specification (*1.1*) and the target system specification (*1.2*) for REACT's feedback loop with Clafer and UML. The separation into adaptation options specification and target system specification allows REACT to reuse the same adaptation options specification for different communication systems. For instance, a shared repository with reusable specifications could offer other domain experts the possibility to readily use these specifications for their system. In this case, they could only specify the target system specification and reuse an existing adaptation options specification. With the Clafer executable[4], it is possible to check the adaptation options specification as well as to test the specification with sample inputs at design time for assuring a correct behavior.

After the modeling part, the domain expert connects REACT to the target system. First, she implements the effector interface (Listing 2) to the target system (*2.1*). As it is common in the field of self-adaptive systems to have parameter and architectural adaptation [39], the interface encompasses a method for each adaptation type. For parameter changes, all available attributes of the UML instances are transmitted to the target system in REACT's current version instead of only the values which changed. Similar, complete UML instances are transferred instead of deltas for component changes. Future versions of REACT may only transfer changes.

```
interface IEffector {
    void sendParameterChanges(ParameterChange p);
    void sendComponentChanges(ComponentChange c); }
interface ISensor {
    void receiveSensorData(SensorData s); }
```

Listing 2. `IEffector` and `ISensor` interfaces.

After this step, the domain expert implements the sending of sensor information from the target system to REACT (*2.2*). The `ISensor` interface, shown in Listing 2, specifies the function implemented in REACT's sensor component for receiving sensor information. Hence, the target system calls the sensor function periodically to trigger REACT. After the implementation step, the interfaces can be used with REACT and any kind of specification, i.e., arbitrary adaptation options specification and target system specification. Hence, they can be reused if the specification changes.

Finally, the domain expert provides REACT with a minimal set of configuration information (*3.1*). This configuration step enables a distributed deployment of REACT's components. One configuration file for each MAPE component allows configuring the component type, a unique identifier, the successor component, and the corresponding knowledge component. REACT then creates and deploys a respective component for each key/value-based configuration file.

VI. Evaluation

This section experimentally evaluates the prototypical implementation of REACT[5]. First, we compare REACT with Rainbow, a well-known and frequently applied framework for model-based adaptation. For doing so, we implemented the simulation-based SEAMS exemplar SWIM (Simulator for Web Infrastructure and Management) [47], which represents a cloud system. Second, this section presents the application of REACT in an emulated communication system in the field of Software-Defined Networking (SDN).

A. Cloud Server Management

In our first experiment, we compare REACT with the well-known Rainbow framework [3] in terms of development effort, performance, and features.

Rainbow Framework: 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

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

[4]https://gsd.uwaterloo.ca/clafer-tools-binary-distributions.html, v. 0.4.5

[5]Supplementary material including a dockerized version of REACT, evaluation scripts, and documentation can be found here: https://github.com/martinpfannemueller/ACSOS2020-Supplementary-Material.

properties. This separation means that the same code for Rainbow can be used across multiple deployments of the system by only changing probes (and effectors). Evaluators check for satisfaction of constraints and properties in the model and triggers adaptation if any problems are found, (e.g., the response time falls below some threshold or the cost of deployment becomes too high). The adaptation manager, on receiving the adaptation trigger, chooses the "best" adaption plan to execute, and passes it on to the strategy executor, which executes the strategy on the target system via effectors.

The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as reflected in the models. The underlying decision making model is based on decision theory and utility [37]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [13], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the target system with variable execution time. A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the target system. 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 [48], predictive proactive adaptation [49], and human-machine cooperation [50].

Scenario: In this experiment, REACT and Rainbow adapt a cloud server deployment providing a web application. This experiment uses SWIM [47], which offers a reproducible way for evaluating adaptation logics in this web server environment. It is a simulation environment based on OMNeT++. The SWIM exemplar consists of multiple simulated web servers connected to a round-robin load balancer. The load balancer distributes simulated requests and the corresponding server simulates the execution. Each web server response can contain optional content (e.g., advertisements) which increases the response time but also leads to additional revenue for the web site operator. The overall goal of the system is thus continuously reaching a fixed response time goal, maximizing the revenue with the optional content, and minimizing the cost for the servers. Accordingly, there are two ways of adapting the running system: 1) Adding or removing servers, and 2) controlling the percentage of responses with optional content. We use the "1998 World Cup Web Site Access Logs" trace provided by SWIM for comparison.

Experimental Setup: In accordance with [37, 51], we measure the required source lines of code (SLOC) for implementing the SWIM use case with REACT and Rainbow. The lines of codes comprise the specification files and the interface implementation for connecting the respective approach to SWIM. Further, we measure the cycle time for executing an adaptation in REACT and Rainbow as well as the processing time of each MAPE activity. We conduct 10 evaluation runs each for REACT and Rainbow on a machine equipped with an Intel Core i7-8700k and 32GB of RAM. Both approaches have been executed in Docker containers. For better comparability, REACT and Rainbow perform similar adaptations, leading to the same response times and simulated costs for the web site operator in SWIM.

Results: In this experiment, we answer three research questions regarding REACT's development effort, performance, and capabilities in comparison to the Rainbow framework.

RQ1: 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 lines of code required to achieve self-adaptivity and the number of different programming languages, tools, and technologies she needs to be familiar with. Both metrics apply to i) specifying the adaptive behavior and ii) implementing the interfaces to SWIM. Table II shows the lines of code for the specification of the adaptive behavior.

TABLE II
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**	

We observe that specifying the adaptive behavior with REACT requires considerably fewer SLOC. The domain expert has to write 152 SLOC in 2 files with clear responsibilities. To achieve the same behavior with Rainbow, the domain expert has to write 593 SLOC in 6 different files using various languages. Next, we assess the development effort for the interface implementation. In Table III, we observe that REACT requires 200 SLOC and Rainbow requires 204 SLOC. However, REACT requires fewer (configuration) files for setting up the connection. In addition, due to its language-independent interfaces, domain experts can use their preferred language.

TABLE III
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**	

RQ2: How does REACT compare to the state of the art in terms of performance?

Figure 4 presents the average runtimes per MAPE activity as well as their average sum. REACT considerably outperforms Rainbow in the monitoring and analyzing phase. Since Rainbow holds an exact architecture model of the target system, it updates the model when new sensor data is available, periodically checks for problems including an analysis where the problem is located in the model, and triggers an adaptation. This design choice thus allows a more complex analysis of the target system architecture at the cost of slower adaptation. The total execution time of an adaptation cycle in REACT is determined to a very high degree by the planner component. This is not surprising, as the planner executes Chocosolver to find a valid model instance. Clafer itself scales well with increasing problem size even with models of several thousand Clafers [42, 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.

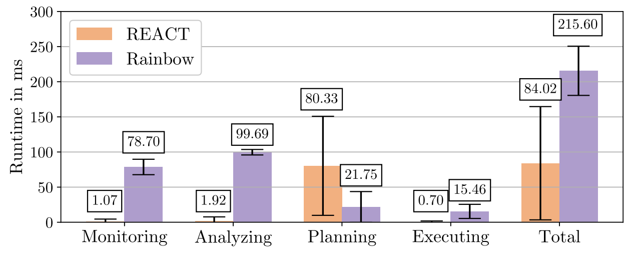

Fig. 4. Average run times of the MAPE activities of REACT and Rainbow.

RQ3: 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 works utility-based with the possibility to weight optimization goals, which may considerably reduce a domain expert's effort in scenarios with multiple goals. REACT, however, offers runtime modifications of the adaptation behavior, decentralized control, and multi-language support. Accordingly, if there is the need for weighted optimization and a central deployment without too strict timing requirements, Rainbow is a good choice. If there is no need for weighted optimization, and the requirement for decentralized deployments and fast execution, REACT is a good candidate.

B. SDN-Based Wifi Handover

In the second experiment, we show REACT's focus on communication systems in a real-world SDN-based use case adding adaptive behavior to an underlay network. Sensor information from two distributed hosts is pushed to a decentralized adaptation logic following the *regional planning* pattern [28]. **Scenario**: A car receives a live stream from a streaming server via a wireless network connection (see Figure 5). With each handover between the wireless network towers along the road, the user in the car experiences packet loss. The goal is to improve the quality of experience by minimizing the packet loss during the handover. SDN *"is a paradigm where a central software program, called a controller, dictates the overall network behavior"* [52]. The controller manages a set of controllable switches. These switches deal with incoming packets according to flow rules. A flow rule can, for instance, forward a packet to a specific port, change or add packet headers, or implement firewall functionality by rejecting a packet. The SDN controller offers an API that allows domain experts to write applications for the controller. In our case, we apply these capabilities by monitoring and adapting the flow rules with REACT for seamless handovers. A specific adaptation means that there should be flow rules for the current wireless tower, as well as flow rules duplicating the streaming traffic to the next tower. This duplication should only take place when the car is going to leave the radio range of the first tower soon. Achieving this behavior continuously requires a recurring adaptation of the flow rules.

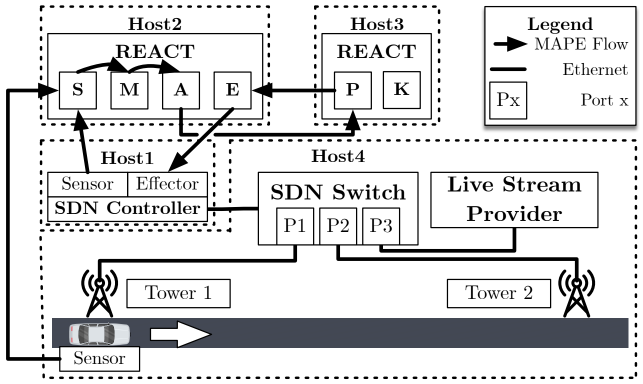

Fig. 5. SDN handover setup with two access points in a live streaming scenario with multiple sensors. The SDN sensor sends topology information while the car sends its distance to the currently connected radio tower. REACT creates flow rules for adaptively duplicating the live stream traffic.

Experimental Setup: REACT receives sensor data from two sources. First, the sensor SDN application sends the host location, which contains addressing information as well as the currently connected network tower, to REACT. An additional sensor in the car sends the distance to the currently connected access point to REACT every second to minimize the network traffic duplication. We apply REACT's built-in *aggregation* monitoring strategy to combine the data for reasoning. REACT's integrated MAPE components are distributed according to the regional planning pattern [28]. Monitor, analyzer, and executor are deployed on a separate machine. A powerful and stable resource runs the computationally intensive planner and the knowledge service.

We use the ONOS [53] SDN controller in this evaluation, which runs on another separate machine. The network was emulated with Mininet-Wifi [54] on a forth machine. In a pre-test, we used the VLC player for streaming a 4K video. However, for better controllability and reproducibility of the experiment, we run Iperf[6] in UDP mode with $25\,\mathrm{Mbit/s}$, the bandwidth recommendation of Netflix for 4K video streams. The ethernet connections have a bandwidth of $100\,\mathrm{Mbit/s}$. We emulated four access points, one moving wireless node as the car, and a static host representing the live stream provider.

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

Results: We answer the following research question:

RQ4: Can REACT be implemented and used effectively in a real-world communication system?

As shown in Figure 6, self-adaptivity with REACT reduces the packet loss considerably. The aggregated mean packet loss of the overall simulation time improves from $4.87\,\%$ in the reactive forwarding case to $0.48\,\%$ with REACT.

Hence, we observe that REACT can be applied effectively in a real-world communication system (*RQ4*). In addition, REACT makes it possible to efficiently change the behavior of the SDN controller by changing the adaptation options specification. It further allows to port the specified behavior to different SDN controllers by only implementing the effector interface and sending sensor data accordingly. Thus, we achieve portability of the specified behavior which is not available in SDN in general, where each SDN controller needs specific SDN applications with different interfaces to the controller for applying a certain behavior in the network.

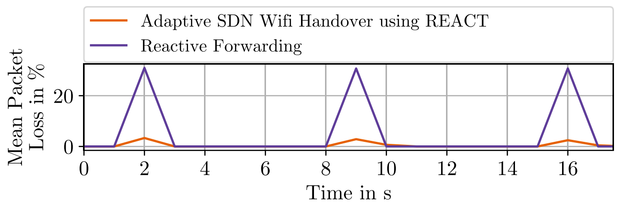

Fig. 6. Average packet loss in % over time. The duplication flows are added as soon as the distance to the connected access point is below a threshold.

C. Threats to Validity

REACT's current implementation is limited in both its features and its performance. For instance, a weighted optimization of multiple adaptation goals, which is possible in Rainbow, is not supported by Clafer. Thus, we plan to integrate an easy-to-use API for domain experts to forward multiple, weighted optimization goals to REACT's underlying Choco-solver in future work. In the evaluation, we measure SLOC and the number of different languages to show REACT's low development effort for domain experts. Even though SLOC are frequently used as a metric (e.g., in [22, 37, 51]), a future user study with domain experts who apply REACT in different scenarios would strengthen validity. In the second experiment, we adapt an underlay network with REACT, showing its capabilities for decentralized control, distributed deployment, and multi-sensor support. It would be interesting to observe the scalability of our approach as far as (i) large Clafer and UML models and (ii) larger system sizes are concerned. This work is further limited to a comparison with Rainbow and to two use cases only. Future research may include a comparison to other frameworks such as SASSY [4] or StarMX [31] in additional use cases from the communication systems domain.

[6]https://iperf.fr/

VII. Conclusion

In this paper, we propose REACT, a reusable runtime environment for model-based adaptations in communication systems. REACT integrates a MAPE-K feedback loop that leverages a Clafer and a UML model provided by the domain expert to autonomously achieve self-adaptivity. We implement REACT and make it available for domain experts. 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 compare 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.

In future work, we will integrate additional interfaces that allow developers to use own analyzing and planning techniques such as machine learning for proactivity or a different specification language such as Stitch [55] instead of Clafer. As verification and validation (V&V) is an important research challenge [8, 55], we plan to add verification of dynamic properties such as runtime V&V techniques and guarantees according to costs into REACT, e.g., using model-checking methods. This will ensure the correctness of the models and REACT will give certain runtime guarantees. Future work additionally includes a user study with domain experts which further investigates the development effort. Focussing on such empirical evidence with practitioners has been identified as general challenge for further self-adaptive systems research [7].

Acknowledgment

This work has been co-funded by the German Research Foundation (DFG) as part of project A4 within CRC 1053 – MAKI.

References

[1] P. Oreizy et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems and their Applications*, vol. 14, no. 3, pp. 54–62, 1999.

[2] M. Charalambides, G. Pavlou, P. Flegkas, N. Wang, and D. Tuncer, "Managing the future internet through intelligent in-network substrates," *IEEE Network*, vol. 25, no. 6, pp. 34–40, 2011.

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

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

[5] S. O. Hallsteinsen et al., "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2840–2859, 2012.

[6] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: unifying class and feature modeling," *Software and System Modeling*, vol. 15, no. 3, pp. 811–845, 2016.

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

[8] R. de Lemos et al., "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2010, pp. 1–32.

[9] C. Krupitzer, M. Breitbach, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems (extended version)," 2018.

[10] S. Sicard, F. Boyer, and N. D. Palma, "Using components for architecture-based management: the self-repair case," in *Proc. of ICSE*, 2008, pp. 101–110.

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

[12] J. Dowling and V. Cahill, "The k-component architecture meta-model for self-adaptive software," in *Proc. of REFLECTION*, 2001, pp. 81–88.

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

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proc. of ESEC*, 1995, pp. 137–153.

[15] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace, "Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems," in *Proc. of SPLC)*, 2008, pp. 23–32.

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

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

[18] M. Pfannemüller, C. Krupitzer, M. Weckesser, and C. Becker, "A dynamic software product line approach for adaptation planning in autonomic computing systems," in *Proc. of ICAC*, 2017, pp. 247–254.

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

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

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

[22] C. Krupitzer, F. M. Roth, C. Becker, M. Weckesser, M. Lochau, and A. Schürr, "FESAS IDE: An Integrated Development Environment for Autonomic Computing," in *Proc. ICAC*, 2016, pp. 15–24.

[23] E. Zavala, X. Franch, J. Marco, and C. Berger, "Hafloop: An architecture for supporting highly adaptive feedback loops in self-adaptive systems," *Future Gener. Comput. Syst.*, vol. 105, pp. 607–630, 2020.

[24] S. Malek et al., "An architecture-driven software mobility framework," *J. Syst. Softw.*, vol. 83, no. 6, pp. 972–989, 2010.

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

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

[27] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the rainbow: self-adaptive failure avoidance in configurable systems," in *Proc. of SIGSOFT*, 2014, pp. 377–388.

[28] D. Weyns et al., "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2010, pp. 76–107.

[29] C. Krupitzer, F. M. Roth, M. Pfannemüller, and C. Becker, "Comparison of approaches for self-improvement in self-adaptive systems," in *Proc. of ICAC*, 2016, pp. 308–314.

[30] T. Preisler, T. Dethlefs, and W. Renz, "Middleware for constructing decentralized control in self-organizing systems," in *Proc. of ICAC*, 2015, pp. 325–330.

[31] R. Asadollahi, M. Salehie, and L. Tahvildari, "StarMX: A framework for developing self-managing Java-based systems," in *Proc. of SEAMS*, 2009, pp. 58–67.

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

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

[34] K. Angelopoulos, V. E. Silva Souza, and J. Pimentel, "Requirements and architectural approaches to adaptive software systems: A comparative study," in *Proc. SEAMS*, 2013, pp. 23–32.

[35] N. Bencomo, P. Grace, C. A. Flores-Cortés, D. Hughes, and G. S. Blair, "Genie: supporting the model driven development of reflective, component-based adaptive systems," in *Proc. of ICSE*, 2008, pp. 811–814.

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

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

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

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

[40] S. Tomforde and C. Müller-Schloer, "Incremental Design of Adaptive Systems," *J. Ambient Intell. Smart Environ.*, vol. 6, no. 2, pp. 179–198, 2013.

[41] M. Antkiewicz, K. Bak, K. Czarnecki, Z. Diskin, D. Zayan, and A. Wasowski, "Example-Driven Modeling using Clafer," in *Proc. of MoDELS*, vol. 1104, 2013, pp. 32–41.

[42] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled," in *Proc. of SLE*, 2010, pp. 102–122.

[43] M. Weckesser, M. Lochau, M. Ries, and A. Schürr, "Mathematical Programming for Anomaly Analysis of Clafer Models," in *Proc. of MODELS*, 2018, pp. 34–44.

[44] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*, TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. [Online]. Available: http://www.choco-solver.org

[45] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, 2004.

[46] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *Proc. of SCC*, 2007, pp. 474–481.

[47] G. A. Moreno, B. R. Schmerl, and D. Garlan, "SWIM: an exemplar for evaluation and comparison of self-adaptation approaches for web applications," in *Proc. of SEAMS*, 2018, pp. 137–143.

[48] J. Cámara, D. Garlan, B. R. Schmerl, and A. Pandey, "Optimal planning for architecture-based self-adaptation via model checking of stochastic games," in *Proc. of SAC*, 2015, pp. 428–435.

[49] G. A. Moreno, J. Cámara, D. Garlan, and B. R. Schmerl, "Flexible and efficient decision-making for proactive latency-aware self-adaptation," *ACM TAAS*, vol. 13, no. 1, pp. 3:1–3:36, 2018.

[50] J. Cámara, G. A. Moreno, and D. Garlan, "Reasoning about human participation in self-adaptive systems," in *Proc. of SEAMS*, 2015, pp. 146–156.

[51] T. Vogel, "Model-Driven Engineering of Self-Adaptive Software," Ph.D. dissertation, University of Potsdam, 2018.

[52] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[53] P. Berde et al., "ONOS: towards an open, distributed SDN OS," in *Proc. of HotSDN*, 2014, pp. 1–6.

[54] R. dos Reis Fontes and C. E. Rothenberg, "Mininet-WiFi: A Platform for Hybrid Physical-Virtual Software-Defined Wireless Networking Research," in *Proc. of SIGCOMM*, 2016, pp. 607–608.

[55] B. Cheng et al., "Using models at runtime to address assurance for self-adaptive systems," in *Models@run.time - Foundations, Applications, and Roadmaps*. Springer, 2011, pp. 101–136.