FESAS IDE: An Integrated Development Environment for Autonomic Computing

Christian Krupitzer*, Felix Maximilian Roth*, Christian Becker*,
Markus Weckesser†, Malte Lochau†, and Andy Schürr†
*Chair of Information Systems II, University of Mannheim
Schloss, 68131 Mannheim, Germany
Email: {christian.krupitzer, felix.maximilian.roth, christian.becker}@uni-mannheim.de
†Real-Time Systems Lab, Technische Universität Darmstadt
Merckstr. 25, 64283 Darmstadt, Germany
Email: {markus.weckesser, malte.lochau, andy.schuerr}@es.tu-darmstadt.de

***Abstract*—While Autonomic Computing can ease the maintenance of systems through adaptations [1], the development of Autonomic Computing systems itself introduces a high complexity. Literature suggests that reusable processes for the development and reusable components in the adaptation logic can reduce the complexity. Existing approaches aim to reduce this complexity with tools and frameworks for specific tasks in the development of the adaptation logic of Autonomic Computing systems. However, to the best of our knowledge, none of these approaches offer an Integrated Development Environment (IDE) for it.**

In this paper, we extend FESAS – our framework for building reusable adaptation logic components – with Eclipse plug-ins integrated into the FESAS IDE for a simplified development of MAPE components as well as a process for deployment of the components. In this paper, we present these tools. Further, we evaluate their potential to ease the development of self-adaptive systems within five example cases. Last, we discuss the benefits and limitations of the FESAS IDE.

I. Introduction

Today, an increase of the distribution of systems emerge through trends as cyber-physical systems (CPSs) with its growing number of mobile and embedded devices as well as the omnipresence of (wireless) networking. These trends enable new applications, such as autonomous driving, Industry 4.0, or Internet of Things. However, they need integration of all available, highly specialized, and heterogeneous devices, ranging from embedded sensor nodes to servers in the cloud. Further, the inclusion of data streams with web data and sensor data leads to a high complexity in system development. In Autonomic Computing, these issues are addressed with *self-adaptive systems* (SASs) that automatically modify themselves by adjusting parameters or components of the system in response to changes in their operating environment or changes in the system resources itself ([2], [1]). Autonomic Computing includes various domains, such as CPSs, cloud computing, or self-organizing systems/ Organic Computing.

Developing, configuring, and maintaining such systems is a very difficult, error-prone, and time-consuming task [3, p. 2]. As identified in [4, p. 2] "we need [..] systematic development, deployment, management and evolution" of SASs. Often, the development is use case specific ([3, p. 2], [5]). Reusability could help to establish standards. There are various elements of the adaptation logic (AL) or autonomic manager [1], respectively, that can be reused, such as distribution structures (e.g., [6]), communication mechanisms (e.g., publish-subscribe systems), or structures for handling knowledge (e.g., distributed databases). Therefore, we introduced an approach for building reusable ALs in [7]. It separates reusable parts as decentralization structures, communication patterns, and knowledge representation, from parts that need customization, such as reasoning algorithms. The approach offers fine-granular exchange of code within a component of the AL instead of changing the whole component. This enables on-the-fly adaptation without the need to address issues, such as quiescence detection or recovery of the substituted component's state and data.

In this paper, we complement our AL template with an Integrated Development Environment (IDE) for developing and designing SASs: the *FESAS IDE*. Our main contributions are: (i) the FESAS IDE itself which offers two tools for developing and designing SASs, (ii) a standardized process for developing and deploying SASs (as identified in [8]), and (iii) an evaluation of the FESAS IDE in five example cases. We target with the FESAS IDE developers of the aforementioned systems in the domain of self-adaptive CPSs.

The structure of the remaining paper reflects these contributions. Section II introduces reusability within the AL as well as tools for SAS development and examines related work. In Section III, we present the FESAS framework, followed by the development process with FESAS in Section IV. Section V presents the FESAS IDE. There, we describe the integration of two Eclipse plug-ins into the FESAS IDE that enable development and design of SASs. In Section VI, we use the FESAS IDE in five example cases to build self-adaptive systems. We discuss these results in Section VII. Finally, Section VIII concludes the paper with a summary and future work.

II. Reusability in Autonomic Computing

Autonomic Computing systems are divided into two parts: the *adaptation logic* (AL), or *autonomic manager* [1], respectively, and the *managed resources* (MRs). The AL contains a control structure that processes the information gathered

about MRs. In Autonomic Computing, the most prevalent structure for the AL is the MAPE cycle [9]. MAPE stands for monitoring the environment (M), analyzing monitored data (A), plan adaptation actions (P), and control the execution (E) of these plans on the MRs [1]. Whereas the MRs deliver the system's functionality, the AL is responsible for managing the resources and adapting them. This paper focuses on the implementation of the AL, as this makes a system self-adaptive or autonomous, respectively. Often, SASs are composed of various subsystems, which are distributed. In such systems, the MAPE functionality can be distributed, too. Therefore, decentralization must be supported by the AL [5]. In literature, different distribution patterns can be found (e.g., [6]).

In [5], we presented different approaches that can support developers of (self-)adaptive systems in implementing the AL. In the following, we sketch some of these frameworks and present tools that support the development of SASs. For a more complete overview of related work, the interested reader is referred to [5].

Architecture-based solutions (e.g., *Rainbow* [10] or *ArchStudio* [2]) offer components or layers that control the adaptation. Service-oriented frameworks for SAS development, such as the *SASSY* framework [11], support developers in the implementation of service-based SASs. In both approaches, the support is often specific to a given architecture description language (ADL) or the developers have to implement specific components. Additional to architecture models, many authors propose the use of model-based approaches for analyzing and planning, for instance, based on *runtime models* [12] or *Megamodels* [13]. These modeling concepts are reusable, but often, they have to be adjusted to domains. Component-based solutions are present in Autonomic Computing, too. Patouni and Alonistioti describe a framework with a template for generic autonomic components [14]. Other component-based solutions for Autonomic Computing can be found, e.g., *Accord* [15] and *PCOM* [16]. Nevertheless, their focus is on structural adaptation by exchanging components and not on how to build reusable components that offer the exchange of fine-granular parts of the components itself. Besides these frameworks, other approaches address parts of the implementation process. More information can be found in [5].

Some approaches offer development tools for implementation of the AL, for instance, the *SASSY Development Environment* [11], the *Rainbow Development Toolkit* [10], the *MUSIC Studio* [17], the *ArchStudio tool suite* [2], or IBM's *Autonomic Computing Toolkit* [18]. Often, these approaches need configuration effort, force the developer to implement specific components, or have restrictions regarding the used modelling/implementation languages. As we focus on another level of abstraction, in our *FESAS* framework, the developer has a higher degree of freedom and is free in the decision how to implement the functional logic of the AL components. As our framework hides implementation details, such as communication or system deployment, the developer can focus on the system's functionality.

All presented approaches address reusability on a higher level of abstraction. They focus on the exchange of components or the definition of the AL's components. To the best of our knowledge, there is no approach that focuses on the internal implementation of the components which would facilitate a fine granular definition of exchangeable parts of the components. In common component-based adaptation approaches (e.g., [15], [16]), the whole component needs to be exchanged when code should be changed. We shift this to a view of adaptation within a component. For the described scenario of adaptation of the AL, it would be beneficial to exchange only a part of the component. This can be achieved, for instance, through dynamic code reloading/rewriting [19] or aspect-oriented techniques [20]. Such approaches allow to keep the component's structure as well as the communication mechanisms and replace the functionality of the component only. Additionally, this exchange of the functional logic facilitated by a separation of functional logic and the rest of the component for handling communication and data, allows for faster development of components as solely the functional logic needs to be adjusted.

In the next section, we discuss our FESAS framework that enables to build SASs with reusable components for the AL.

III. The FESAS Framework

The FESAS framework focuses on code reuse and simplified exchange of code in the components of an SAS's AL. FESAS complements these functionalities with capabilities for adapting the AL at runtime. In the following, we present the FESAS framework and illustrate its use with a cloud scenario comparable to the Amazon's EC2. Different servers offer resources for running virtual machines (VMs) on them. Each data center has one AL for self-management through performing the following adaptations:

- Start new servers in case of high utilization.
- Migrates VMs from one server to another for distributing the workload or in case of server defects.
- Shut down servers in case of reduced workload.

Next, we present the *FESAS Adaptation Logic Template* [7] that acts as system model for the AL and the runtime support that is offered by FESAS for adapting the AL.

The most prominent model for a control structure in the Autonomic Computing community is the MAPE-K model ([1], [9]). Therefore, we used this model in our approach and in our template for the AL as shown in Figure 1. Elements with round edges are part of the MAPE functionality, namely: *monitor*, *analyzer*, *planner*, and *executor*. The *knowledge* component stores all kind of data which is relevant for the AL, including the *context manager* that handles context information. Hexagonal formed elements are interfaces: *Sensor* and *effector* connect AL and MRs, whereas *physical sensors* and *context actuators* connect MRs and their environment, respectively. The elliptic formed elements represent *probes* and *actuators*. The sensor captures the information from the probes and the monitor offers an aggregated view to the analyzer (cf. probes and gauges in Rainbow [10]). Actuators enable to adapt the managed resources by receiving instructions from an effector.

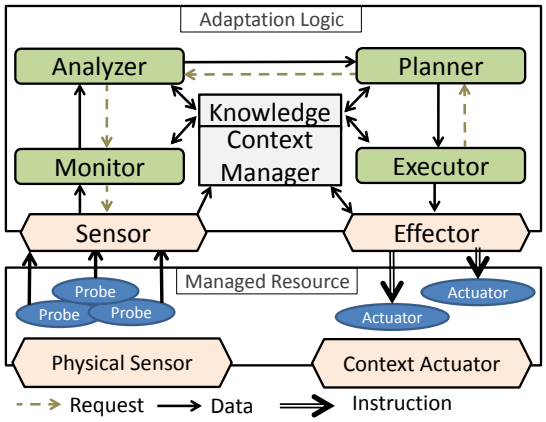

Fig. 1: The FESAS Adaptation Logic Template (cf. [7]).

Figure 1 shows a centralized AL. However, the distributed nature of SASs is often reflected in the AL. Therefore, the AL has to support distribution. Thus, the MAPE components are partially optional for a specific subsystem, but need to be present in the global view of the SAS. This enables the implementation for different settings of distribution, e.g., as the patterns for decentralization control presented in [6].

Additionally to the AL template, we provide a component template for reusable components within the AL. The MAPE components consist of an exchangeable functional logic and logics for communication and data handling. Interfaces for these elements enable reusability as well as interchangeability. This separation of communication, data handling, and functional logic enables reusability of parts of the components. Further, it enables the exchange of algorithms at runtime without the need of changing the whole component. This eliminates issues, such as quiescence detection or recovery of the state and data of the substituted component. However, the template does not make any claims on how to implement the functional logic within the components. Therefore, known approaches as presented in Section II can be used for the implementation of the functional logic. For example, it would be possible to implement a functional logic that uses a specific modeling language, such as the languages presented in Section II (e.g., [12] or [13]). This way, the modeling language supports the reuse of the functional logic as the same logic can be used in different systems that use the same modeling language. For a more detailed description of the component template, the interested reader is referred to [7].

Mapping this to the cloud scenario, the AL would follow the MAPE principle. MAPE components would measure and analyze the workload and plan for starting/shutting down servers. All components can share the same component implementation. In object-oriented implementation languages, they would have a common super class. Components vary only in their properties and functional logic but use the same code for communication and the same component template.

Changes in the MRs, in the environment, or in the system goals might imply an adaptation of the AL itself at runtime. AL adaptation may have several goals, such as satisfying a certain quality level (self-healing), or increasing performance of the system through improved adaptation of the MRs (self-optimization). Thus, in [21] we proposed an approach for adapting the AL. In our approach, an Adaptation Logic Manager (ALM) is added as an additional layer to the SAS that is responsible for adaptation of the AL. As runtime adaptation of the AL is out of scope for this paper, we refer the interested reader to [21] for further information.

IV. System Development with FESAS

In the last section, we introduced the FESAS framework's system model. In this section, we present how developers can use the FESAS framework for building SASs. Having a defined process for SAS development identifies the challenge presented in [8]. Comparable to other frameworks as *SASSY* [11], there are two roles integrated in the development process with FESAS: the *system developer* who writes code and the *system designer* who defines the configuration of the SAS. Further, FESAS defines a process for development as well as deployment of an SAS. This addresses the lack of processes as well as specific implementation guidelines or components as identified in [4, p. 11]. Figure 2 captures the workflow for developing an SAS and shows, how the FESAS framework supports developers of SASs. In the following, we describe the workflow in more detail.

At design time, system developers have to implement the functional logics for SASs, e.g., the monitoring logic for a specific use case. This can be done manually or by using the *FESAS Development Tool*. Further, the developer has to define metadata for these elements. The code including the metadata is saved in the FESAS Repository and, from there, deployed at the target system at runtime. System designers specify the system's configuration for a specific application, i.e., the functional logics for its MAPE elements and the distribution pattern. This is done by either writing a JSON file or using the model-based *FESAS Design Tool*. Both tools, the *FESAS Development Tool* and the *FESAS Design Tool*, are available as Eclipse plug-ins. Section V describes these tools in detail.

At runtime, the FESAS middleware configures the AL based on the information in the system design model as well as the descriptions of the implemented components that the FESAS Repository stores. Therefore, components for the MAPE elements are started. These components constitute an implementation of the FESAS Adaptation Logic Template for a specific system infrastructure, i.e., a specific programming language and/or communication middleware. In [7], we present a reference system implemented in Java. This system offers a publish-subscribe service for information exchange within the AL based on the BASE middleware [22]. It is ready out of the box and only requires minor configuration. When starting the SAS, FESAS uses the FESAS Repository to initialize these components with the functional and communication logics as specified in the configuration files. At runtime, the local repository integrated in the FESAS middleware sends

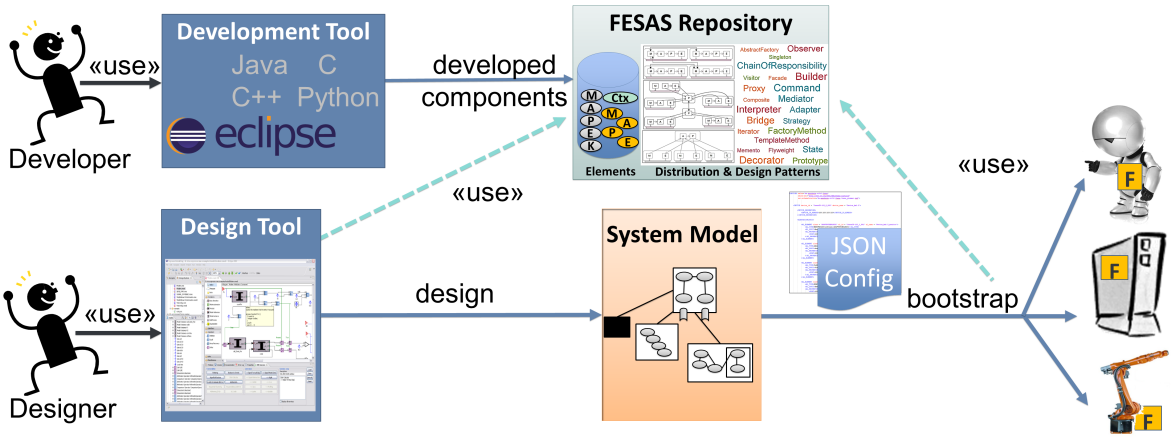

Fig. 2: Workflow with the FESAS framework. The letter 'F' on devices indicate, that these devices run the FESAS middleware.

requests in form of logic contracts for functional and communication logic elements to the FESAS Repository and receives suitable code. For future work, we plan to further formalize the contracts' properties as this can improve the handling of contracts for finding a suitable functional logic. Once the initialization is finished, the AL controls the SAS. Based on the selected algorithms in the functional logics, parameter and/or structural adaptation (including algorithmic adaptation) can be initiated by the AL. Through this modular approach, FESAS offers a lightweight development environment and can be combined with various concepts for building the AL.

For the cloud scenario, the system developer has to implement the functional logics, such as the logic for measuring the servers' workload for a monitor component. The implementation of the AL uses the components and communication structures provided by the reference system. The system designer writes configuration files for the MAPE elements in the AL. In case of multiple data centers with global decision making, the ALs of the data centers exchange information. As a central planner exists, the data centers are connected in the design model. In a decentralized setting, each data center has its own planner, hence, there are no connections in the design model between the data centers. At runtime, the FESAS middleware uses the specified information for setting up the AL using the FESAS Repository as described above.

V. The FESAS IDE

The FESAS middleware offers support for the development of SASs with a simplified exchange of code. However, implementing the functional logics and writing the configuration manually is error-prone as well as time-consuming since the developer has to learn the FESAS syntax for the configuration files. We address these issues with the *FESAS IDE*.

The FESAS IDE is based on the Eclipse IDE. It offers two tools: the *FESAS Development Tool* and the *FESAS Design Tool*. These tools represent the separation of responsibilities in developing SASs using FESAS into the roles of system developer and system designer. The system developer uses the *FESAS Development Tool* for developing the functional logics' code, which is then stored in the FESAS Repository. Using the *FESAS Design Tool*, the system designer configures the components of a specific SAS's AL and specifies, which functions these components should load at runtime. Both tools are implemented as Eclipse plug-ins. System designer and developer can cooperate and jointly define the components as well as their functionality. However, the split in the responsibilities and the contract concept enables to design the SAS and use the FESAS Repository as market for finding suitable MAPE functionality.

Next, we introduce the *FESAS Development Tool* and the *FESAS Design Tool* and illustrate the process of using the tools with the cloud scenario known from the previous sections. We describe how the system designer and system developer can use the plug-ins to speed up the development of an SAS.

A. FESAS Development Tool

The *FESAS Development Tool* offers support for developing the functional logics of the SAS's MAPE components. Through the integration into the Eclipse IDE, developers can use all known features for coding, such as syntax highlighting, on-the-fly compilation, and simplified integration of libraries. Developers can use an existing Eclipse installation and solely have to add the *FESAS Development Tool* feature and import a project, that provides the package structure of the repository as well as configuration files that hold parameters for the plug-in. The process for using the *FESAS Development Tool* is divided into four steps: (i) initializing a logic element and its metadata file, (ii) writing source code and testing the functionality, (iii) preparing the elements for the repository, and (iv) committing the prepared elements to the FESAS Repository. We will illustrate these activities in the following.

First, the developer has to generate a logic element. During the creation process, the developer specifies the metadata of the functional logic. In the cloud scenario, such a logic element could be the functional logic of a monitor for monitoring the workload of servers. The tool initializes a source code file

with the metadata as well as a JSON file that contains the metadata used by the repository. The *FESAS View* extends the perspective in Eclipse and offers an overview of the metadata of the selected source code file. Developers can implement the functions as usual. Further, for testing purposes, developers can start a prototype system, that loads the implemented code into a local test system. As the plug-in is integrated into Eclipse, the test system provides the debugging support known from Eclipse. This allows for detecting errors early in the development process without the need to set up the whole system and its communication. For the cloud scenario, the developer has to specify the format of the data from the MRs (here: the servers) and, if she would like to test the monitor's functionality, add the functional logics' code of a sensor and a monitor, and specify a connection between each other in the test system (substituting the design information).

After finishing the implementation and testing phase, the written code is packed into zip files. Such a zip file contains the metadata as JSON file and the functional logic as well as files containing referenced code, so called dependencies. The plug-in calculates the dependencies. In case of an implementation in Java, the class file of the functional logic is loaded and the plug-in searches recursively in the byte code for any related class that is not part of the Java or the FESAS library. The plug-in adds all identified dependencies to the zip file.

Once the developer has created zip packages for all logic elements, she can commit them to the FESAS Repository. The plug-in marks files that have been previously sent to the repository and excludes them for future sending processes unless the developer did adjustments in its code. An SAS that runs the FESAS middleware can load the code from the repository as described in the previous section. With the help of the *FESAS Design Tool*, which is described in the next section, the system designer writes the configuration files that specify which logic should be loaded at system deployment.

B. FESAS Design Tool

Besides implementing the functional elements of the SAS, the SAS has to be designed. The *FESAS Design Tool* supports the design process of an SAS. The system designer has to write configuration files that specify which (sub)systems host which parts of the MRs and/or the AL. Additionally, the system designer can specify the functional logic that the components implement. Last, the system designer uses the *FESAS Design Tool*, to describe the connections between the components, such as which sensor senses which MRs or from which monitor an analyzer receives information. This enables the implementation of decentralization patterns (cf. [6]).

The *FESAS Design Tool* is an Eclipse plug-in that offers a model-driven development (MDD) approach for designing the system. It provides a graphical editor in which the system designer can specify all information for the AL's configuration. Further, the plug-in is complemented with a generator for JSON files representing the information specified in the editor. These files are used in the system deployment process for configuring the SAS. The *FESAS Design Tool* is based on the *Eclipse Modeling Framework*[1] (EMF), the *Graphical Modeling Framework*[2] (GMF), and the *Acceleo*[3] code generator. Figure 3 depicts the process of configuration file creation using the *FESAS Design Tool*. Next, we highlight the implementation of the editor using EMF and GMF. Subsequently, we describe the JSON file generation process.

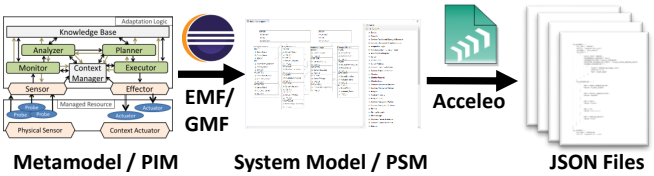

Fig. 3: Process of JSON configuration file generation.

Both, EMF and GMF, can seamlessly be integrated in Eclipse and are available as plug-ins. EMF offers a modeling framework with code generation facilities. As other MDD approaches, EMF specifies a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The PIM represents the metamodel and is implemented as an Ecore model with EMF. It is based on the FESAS Adaptation Logic Template presented in Section III. Elements of the metamodel represent the MAPE components with their properties, including the contracts for functional logics that have to be loaded at runtime. The PIM's elements are the base for the implementation of PSMs. A PSM specifies the components of a specific SAS's AL, its configuration, and the connections between the components. Having the EMF files only, a developer would have to write the PSMs manually (by writing XML files complying to the EMF syntax). For a more convenient use of the plug-in, we implemented a GMF-based graphical editor. GMF is part of the Graphical Modeling Project and supports the development of graphical editors for all types of models and purposes. Often, the combination of EMF and GMF is used to support MDD with the Eclipse IDE. GMF offers the definition of the elements on a canvas, the definition of tools in a palette, as well as the mapping of tools to diagram elements and to elements in an EMF Ecore metamodel. This enables an automatic creation of an element in the PSM, once an element is added to the diagram canvas. Further, the system designer can only use the elements of the palette. This guarantees the structural correctness of the model as EMF/GMF permits to use these elements in another way as specified in the metamodel.

Figure 4 shows the diagram canvas with an example of an SAS with a decentralized AL. System designers can drag and drop elements from the palette into the canvas (e.g., an analyzer component), specify the functional logic of these elements, and add connections (e.g., to a planner or another analyzer). Connections between components are represented by arrows in the canvas and by elements in the palette. Further, arrows between `device` and the `managed element` and `adaptation logic` groups indicate which physical

[1]EMF's website: https://eclipse.org/modeling/emf/
[2]GMP's website: http://www.eclipse.org/modeling/gmp/
[3]Acceleo's website: http://www.acceleo.org/pages/welcome/en

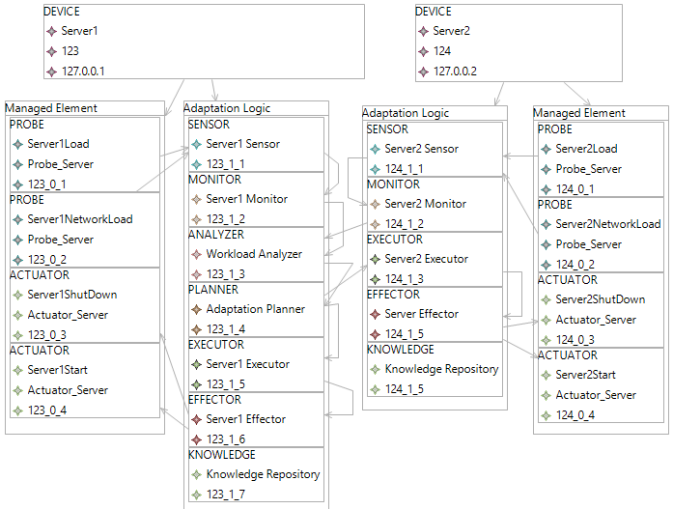

Fig. 4: Diagram canvas with an exemplified SAS.

device deploys which elements. The SAS shown in Figure 4 represents a data center with servers as MRs. Each device is a server complemented with (a part of) the AL. Following the Master/Slave pattern [6], `Server1` has full MAPE functionality, while `Server2` only performs monitoring and executing. `Server1` performs global analyzing and planning.

Once the system is designed in the editor, the next step is the creation of configuration files. For creating the configuration files, the plug-in uses the functionality of the open source code generator *Acceleo*. Acceleo enables the transformation of EMF models (based on an Ecore domain model) to various formats, such as HTML, XML, or JSON. In the following, we present the mapping of our metamodel to JSON files as well as the implementation of the code generation.

Acceleo uses the EMF-based PSM model and templates of the JSON files as input for the configuration file generation. The templates are composed of static JSON data and wildcards. During file generation, the Acceleo generator replaces the wildcards with the values of the PSM represented by the model specified in the editor. For each device, Acceleo creates one JSON file with the information about the components as well as one file with the information regarding connections. FESAS uses these files during the deployment of an SAS as specified in Section IV.

The resulting configuration file for a device has a JSON element for the device as root element. This element contains the device's properties as JSON attributes and JSON elements for the AL, MRs, and the communication. Each JSON element represents an array with the AL components, the probes and actuators, and the communication links, respectively. The AL components have logic contracts for the logic they should load. We have chosen JSON, as it is a light-weight and language-independent representation. Listing 1 shows the structure of a JSON file specifying the device.

A system designer can use the editor to design an AL, for example, for the cloud system. Different system designs can be tested by changing the corresponding elements or connections, respectively. For instance, the designer can change the system design from a decentralized setting by eliminating planners and specifying a central planner pattern (cf. [6]). Connections to the MRs can be enabled by specifying properties in the configuration of sensor elements. The AL can be started and initialized with the FESAS framework and its reference system without additional effort.

Listing 1: Excerpt of a generated JSON file (omitting specific use case details).

```
{
  "DEVICE_PROPERTIES": [...],

  "AL_ADAPTATIONLOGIC": [{
    "AL_ELEMENT": {...
      "AL_LOGIC": [...],
      "AL_PROPERTIES" :[...]}},...
  ],

  "COMM_ADAPTATIONLOGIC": [{
    "COMM_ELEMENT": {...}},
  ],

  "MANAGEDRESOURCES": [{
    "MR_RESOURCE": {...}},
  ]
}
```

The combination of editor and code generator is optimized for the FESAS framework, however, not limited to it. Due to the separation between drawing in the editor and configuration file generation, it is possible to extend the plug-in for other languages, e.g., specific ADLs. The editor captures the elements and their properties in an abstract way. The templates in Acceleo define the syntax of the resulting configuration files. Through changing the Acceleo templates and the code of the generator, it is possible to represent the editor's information in another syntax, hence, using another language for the configuration files. In this case, only the code generation part must be re-implemented for the favored ADL. Vice versa, it is possible to use predefined editors for generating FESAS configuration files by changing the templates to the information representation of the syntax used in the editor.

As the design and the development of the AL is divided into two parts, changes in the design do not necessarily lead to changes in the development of code and vice versa. The need for changes in the code after changes in the design depends on the specific implementation of the logic elements. In case the design switches from a decentralized planner to a central approach, it can be necessary to change the planner's functional logic as it now relies on different analyzers' values and needs to combine their results. For future work, we plan to elaborate this and provide code fragments that support a specific type of design and, therefore, a specific decentralization pattern.

VI. Evaluation

We used the FESAS IDE to build SASs in five use cases. The different characteristics of the use cases reflect the flexibility of the FESAS IDE. Next, we present the implementation

of the example cases. Subsequently, we evaluate the suitability of the FESAS IDE for the example cases' implementations.

A. Example Systems

In the following, we describe the implementation of five SASs in different application domains: (i) autonomic cloud, (ii) adaptive highway, (iii) adaptive tunnel lighting, (iv) smart home, and (v) smart vacuum system. The application areas are focused around the topics autonomous robotics, cloud computing, and CPSs. All implementations use the FESAS framework, including the reference system and FESAS IDE.

1) Autonomic Cloud: Cloud computing and data center management are common applications in Autonomic Computing. We built an AL that can manage data centers. The system is an implementation of the use case presented in Section III. The MRs are composed of servers and VMs; both are simulated. Communication between a data center and an entity that simulates requests from clients for starting a VM is established with the BASE middleware [22]. Further, the MRs are connected to the AL using sockets.

2) Adaptive Highway: In accordance with literature (e.g., [23]), we built different SASs for highway management. The systems enable congestion avoidance through adaptive speed limits (using digital traffic signs), re-routing, and releasing shoulders as well as adaptations based on the current weather situations. Further, we incorporate cooperative driving under the assumption that self-driving cars drive on the highway. So far, the systems are built in isolation. The systems use the same components and communication logics but various functional logics. The traffic simulator *SUMO*[4] simulates the traffic flow as well as the Vehicle-to-Infrastructure (V2I) communication infrastructure. *TraCI*[5], an addition to SUMO, allows to change parameters during the simulation, and therefore, to simulate traffic signs and reactions to V2I communication. In the platooning use case, we use self-driving LEGO Mindstorms robots to simulate self-driving vehicles[6].

Within this paper, we focus on the system for releasing the highway's shoulder and re-routing. The system simulates the German highway network between Mannheim and Stuttgart with real traffic data. In this system, the connection between AL and MRs, or TraCI, respectively, is based on XML-RPC. Currently, we are combining the different systems to an Intelligent Highway Vehicle System.

3) Adaptive Tunnel Lighting: Having a high contrast of the light within a tunnel compared to the outside brightness can dazzle drivers. Therefore, lights in tunnels need to be adapted according to outside light circumstances. Brightness of the environment can change multiple times per day through the change between day and night light or weather influences such as rain, snow, or sunshine. Using FESAS, two Master students implemented adaptive lighting of a tunnel in a 6 weeks student project. The lighting of the tunnel as well as the light sensors as managed resources are simulated on a web server implemented in JavaEE. A light-weight context model simulates the environment on the web server. The connection between the tunnel as MR and the AL is established via HTTP.

4) Smart Home: Smart Homes support residents by automatically adjusting certain parameters of the house according to various measurements and rules. Therefore, an AL receives measurements (such as temperature, light intensity, or movements in the house) and uses actuators to influence such values (e.g., turn on the heater, close the shutter, or perform an emergency call). As part of a 6 weeks student project a Bachelor and a Master student implemented a smart home system. The system supports two actions. If the difference between the current temperature and the desired target temperature exceeds a threshold, the Smart Home system adjusts the air conditioner. Further, the system regularly checks the number of people in a room and can turn off the light, if the room is empty for a specified period of time. The environment of the house is simulated in Java.

5) Smart Vacuum System: In accordance with literature (e.g., [24]), two Master students implemented in a 6 weeks study project a smart vacuum system in form of an autonomic cleaning robot. The distribution of dirt in the environment is unknown to the robot. The robot can determine its current location. Further, it perceives whether the area where it is currently located is dirty or not. Accordingly, it decides to clean or to go left, right, forwards, or backwards. The robot is simulated in an environment that can be configured (size and amount of cells to clean) at the start of the system. Currently, we are working on a solution with LEGO Mindstorms as cleaning robots.

B. Analysis

The example cases were helpful for analysis of different factors. On the one hand, we studied how the FESAS IDE improved the development process with FESAS as well as the usability of the FESAS IDE. We interviewed the students that developed the systems and captured their answers in structured answer sheets. On the other hand, we measured the amount of code that was reused within the different cases as this is an indicator for the degree of support offered by the FESAS IDE and FESAS in the development of SASs. In the following, we present the results of our analysis.

For the tunnel, cleaning robot, and smart home use cases, pairs of students had six weeks for implementation without the use of the FESAS tools. Thus, the students had to implement functional logics and write the configuration files manually. Additionally, the students had to implement SASs once again using the FESAS IDE. All agreed, that the FESAS IDE is helpful as it eliminates the need to learn the FESAS configuration syntax and the structure of the component template. Further, they agreed that the IDE is easy to understand. Table I shows the results of the interviews.

Additionally, Bachelor, Master, and Phd students that used the FESAS IDE for developing the example systems evaluated its usability. For the systems with centralized control, the

[4]SUMO's website: http://www.dlr.de/ts/sumo/en/

[5]TraCI's website: http://sumo.dlr.de/wiki/TraCI

[6]A movie showing the platooning approach can be found at: https://www.youtube.com/watch?v=Nnrbq-4Dn24

TABLE I: Answers of students who used FESAS with and without the FESAS IDE (n=3 teams).

	(Strongly) disagree	Neutral	Agree	Strongly agree
The Design plugin was helpful.	0%	0%	33.3%	66.7%
Using the Design plugin was easy to understand.	0 %	0 %	33.3%	66.7%
The Development plugin was helpful.	0 %	0%	33.3%	66.7%
Using the Development plugin was easy to understand.	0%	0	33.3%	66.7%

students could reuse an existing configuration file. Therefore, as not all of the participants had to use the *FESAS Design Tool*, we focus on the use of the *FESAS Development Tool*, only. The results of the interviews indicate that the *FESAS Development Tool* is easy to use and needs a short training period. Further, it offers reusability of code and simplifies the exchange of algorithms in the MAPE components. Additionally, the development process is fastened by the tool in general and the elimination of implementation of general issues (e.g., communication) in specific. The tool seems to be well integrated in the FESAS workflow, abstracts from FESAS specific activities, and therefore, accelerates the process of development with FESAS. Table II shows the results of the interviews.

TABLE II: Answers of students using the FESAS IDE (n=6 students). One person did not answer question 7.

	(Strongly) disagree	Neutral	Agree	Strongly agree
1) The tool has a short training period.	0%	0%	16.7%	83.3%
2) The tool facilitates using FESAS.	0%	0%	0%	100%
3) The tool is easy to use.	0%	0%	66.7%	33.3%
4) The tool supports reusability of code.	0%	0%	50%	50%
5) The tool support simplified exchange of MAPE algorithms.	0%	0%	33.3%	66.7%
6) The tool eliminates the implementation of general issues.	0%	0%	33.3%	66.7%
7) The tool supports testing in the development phase.	0%	66.7%	16.7%	0%
8) The tool is well integrated into the FESAS development process.	0%	0%	16.7%	83.3%
9) The tool accelerates development with FESAS.	0%	0%	16.7%	83.3%
10) The tool accelerates development in general.	0%	16.7%	16.7%	66.6%

Additionally, we analyzed the degree of reusability on the system level. The degree of reusability shows the amount of code, that FESAS provides or the FESAS IDE generates. This code handles generic issues – such as communication – or integrates the MAPE components (excluding the functional logics). Developers using the FESAS IDE do neither need to implement nor customize this code.

As metric, we performed measurements of the *lines of code* (LOC). We acknowledge, that LOC is not the perfect measurement variable as code can be written differently. However, it can be used as an indicator for showing the amount of code that is offered by FESAS and generated by the FESAS IDE versus the amount the developer has to write. The results are shown in Table III and explained in the following.

TABLE III: LOC that FESAS provides or the FESAS IDE generates versus LOC implemented by the developers (percent values indicate share of systems' LOC).

	Cloud	Highway	Tunnel	Home	Cleaner
AL components	7,608	7,608	7,608	7,608	7,608
Functional logics	243	320	253	333	316
Implemented	101	193	129	216	191
Generated	142	127	124	117	125
Dependencies	238	246	80	283	-X-
Total LOC	8,089	8,174	7,941	8,224	7,924
Generated code	95.81%	94.63%	97.37%	93.93%	97.59%

As all systems can use the FESAS reference system without customization, all implementations have 7,608 LOC for the AL components. Only the configuration files and the functional logics vary. As most use cases have a centralized AL, their configuration files are almost identical (except the logic contracts' properties). We divided the LOC of the functional logics in lines that are generated by FESAS and the FESAS IDE (e.g., for import statements) and lines written by the developers. For the example cases, the developers implemented between 101 and 216 LOC in all functional logic elements whereas between 117 and 142 additional lines are generated. Moreover, all developers have to implement functional logics and their dependencies (between 0 and 283 LOC for the example cases). The rest of the code is offered by FESAS and generated by the FESAS IDE. As a result, the degree of generated code is between 93.93% and 97.59%. Moreover, the reused code handles issues that the developer do not have to cover, such as the communication between the MAPE elements or the deployment of the AL and is reusable. This reduces the complexity in the development of SASs significantly.

The next section discusses the results presented in this section and critically analyzes them.

VII. Discussion

This section discusses the results of the evaluation. We divide the discussion into three parts. First, we discuss the reusability introduced through the FESAS IDE. A high reusability indicates a facilitated development as the reusable code is either generated by the FESAS IDE or its complexity is hidden by it. Next, we consider the appropriateness of the FESAS IDE for supporting the development of SASs. Last, we critically debate the limitations of the evaluation and drawbacks of the FESAS IDE.

The five different use cases show that FESAS is not limited to a specific type of system. For each use case, the AL is

implemented using the FESAS framework and FESAS IDE in Java. However, the implementation of the MRs varies. We used Java, JavaEE, Python, C++, and Lejos (a JavaVM for Lego Mindstorms robots). For some systems, the MRs are simulated in a simplified way. For others, we use fully fledged simulators or robots. The connection between MRs and AL shows different approaches: HTTP requests to web servers, XML-RPC, Java-based method calls, and sockets connecting Java programs or connecting the AL with MRs implemented in other programming languages. Through the generic FESAS Adaptation Logic Template and its interfaces, the connection between AL and MRs can be adjusted by changing the functional logic and/or properties of a sensor/effector component. The FESAS IDE supports this process through defined interfaces for developing an AL's components.

The evaluation results for the example cases show that the reference system can be reused without adjustments. This significantly increases the level of code reuse within the different systems as well as reduces complexity in the development of SASs. The FESAS IDE hides the complexity in using the reference system. For the AL, developers only have to implement functional logic elements. In case of a central AL, the system designer can use the predefined configuration files and adjust the links to the MRs. In other cases, the system designer uses the *FESAS Design Tool* and generates configuration files without the need to learn the syntax. This is supported by the FESAS IDE. Further, none of the developers had to deal with communication issues in the AL, as this is handled by the FESAS reference system. The FESAS framework is responsible for the rest of the process (system deployment, loading the functional logics, establishing communication links). This result supports one of our main claims: The FESAS IDE offers support for reusability and a fast and convenient deployment of an SAS. Hence, it enables a fast and convenient setup of the AL in early stages of a development project as well as enhances exchange of functionality. Further, it is possible to reuse existing code with little effort as the FESAS IDE offers an interface for the functional logic elements and hide complexity.

Using the FESAS framework without the IDE would force developers to learn the syntax for the metadata of the code for the functional logics. The same is true for the system designer who would need to learn the syntax of the configuration files. Having both integrated into Eclipse plug-ins enables to hide the details of the syntax. Further, the integration into Eclipse enables the use of implementation workflows known from IDEs. Developers can build MAPE functionality by using the *FESAS Development Tool* for implementing one method only. The system designer uses the *FESAS Design Tool* and generates configuration files without the need to learn the syntax. According to the interviews, both significantly reduce the time for learning how to use FESAS. Furthermore, as Eclipse is a well-known IDE, the developer or designer, respectively, is already familiar with handling the tools. This reduces the learning time. Additionally, the analysis of the example cases show that the integration of existing code is simplified through the use of the *FESAS Development Tool* as it offers a clear interface for adding code. Furthermore, the interviews show that the use of the FESAS IDE is easy to learn and speeds up the implementation of SASs significantly, as it eliminates general issues, such as communication within the AL, and supports a defined process for the development of SASs. However, the developers of the example systems identify that the integration of the testing process could be improved. We will address this in future work.

We acknowledge that the presented example cases represent rather small and simplified systems. However, the objective of the evaluation is to show the flexibility of the FESAS IDE. This is proven, taking the variability of the use cases into account. Furthermore, the fact that some of the students implemented a first version of the system without the FESAS IDE and then with the FESAS IDE definitely can create some form of bias. However, this does not significantly influence the evaluation of the FESAS IDE as the students had to implement the functional logics and write the configuration files manually for the first versions. Manual implementation differs from using the structured approach with the FESAS IDE. However, we acknowledge that this bias can influence the answers of the students regarding the evaluation of the FESAS framework after the second implementation phase due to learning effects. Another factor influencing the results is the support that the students received. As this evaluation was part of Bachelor/Master theses or study projects, the students had an introduction session to FESAS and, additionally, the possibility to get help from the developers of FESAS. This could shorten the learning phase as problems can be solved by asking, which avoids frustration. It would be interesting to analyze the acceptance and suitability of the tools after making them available for public. We plan this for future work.

Additionally, we acknowledge that LOC is not the perfect measurement for the performance of the development. Ideally, we would analyze the "intelligence" of the code that has been produced with the FESAS IDE. On the one hand, we think that this is difficult to measure. On the other hand, our intention was not to evaluate the performance of systems generated with the FESAS IDE but to show that the developers only have to implement a minor part of the AL. Therefore, we think LOC is an acceptable indicator. Further an evaluation regarding the performance of developed software and the influence of the FESAS IDE on that would be an interesting aspect for future work. Possible metrics can be the quality of the resulting software in terms of performance as well as self-* properties. Especially the degree of achieving self-* properties is an important aspect. These properties are part of the functional logics that are implemented by the developer. We plan to extend the FESAS IDE with a mechanism to offer further support for the decisions of the developer, e.g., regarding self-* properties. With this, we could evaluate the performance of the FESAS IDE for automated development of the resulting SASs. Therefore, the FESAS IDE can be a first step to an IDE for self-* properties of Autonomic Computing systems and could be used to learn, how to capture the relevant properties.

VIII. Conclusion

In this paper, we presented the FESAS IDE. The FESAS IDE offers two tools for engineering SASs: the *FESAS Development Tool* and the *FESAS Design Tool*. The *FESAS Development Tool* offers a simplified development of functional logic elements that can be stored in the FESAS Repository and used for AL construction. The *FESAS Design Tool* offers a graphical editor for the construction of the AL, i.e., the distribution of the MAPE components and their functionalities. System designers can drag and drop elements into the editor (e.g., an analyzer), specify the functional logic of these elements, and add connections (e.g., from an analyzer to a planner). An evaluation with implementations of SASs in five different application domains showed the ease of use of the FESAS IDE, the applicability of the FESAS IDE for fast development, as well as the high degree of flexibility and reusability.

For future work, we plan to improve the FESAS IDE with additional functionality, such as an improved testbed for the developed code, the use of predefined patterns (e.g., [6]) for system development, as well as additional support for automated development, e.g., support of the developer in implementing self-* patterns. This can support the designers as well as the developers. Having such mechanisms enable to test the performance of the FESAS IDE for (automated) system development. These tests could be used for an evaluation with other Software Engineering tools. Additionally, we plan to provide code fragments in the *FESAS Development Tool* that support a specific type of design, e.g., a decentralized versus a centralized planning logic. Further, we will specify the logic contracts with a higher degree of formalism for its properties – e.g., by defining an ontology – as this improves the development process as well as the reusability. Additionally, we plan to extend the FESAS framework and tools for using it in other languages than Java, e.g., Python or C/C++. Last, the process of development could be further improved by shifting activities from design to runtime [8]. The use of the Models@run.time approach [25] could help to achieve this.

Acknowledgments

The authors would like to thank Jannis Bergbrede, Lena Burger, Daniel Flachs, Tobias Höhmann, Johannes Kräft, Kai Schoknecht, Daniel Schopp, and Nils Wilken for their support in the implementation of example systems. Further, the authors say special thanks to Martin Breitbach and Janick Edinger for their valuable contribution.

This work was supported by the Julius-Paul-Stiegler-Memorial-Foundation. More information regarding the FESAS project can be found at http://fesas.bwl.uni-mannheim.de/

References

[1] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intellgent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[3] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. LNCS, 2009, vol. 5525, pp. 1–26.

[4] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson et al., "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, 2013, vol. 7475, pp. 1–32.

[5] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing Journal*, vol. 17, no. B, pp. 184–206, 2015.

[6] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola et al., "On Patterns for Decentralized Control in Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475, pp. 76–107.

[7] C. Krupitzer, F. M. Roth, S. VanSyckel, and C. Becker, "Towards Reusability in Autonomic Computing," in *Proc. of ICAC*, 2015, pp. 115–120.

[8] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla et al., "Software Engineering Processes for Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer Berlin Heidelberg, 2013, vol. 7475, pp. 51–75.

[9] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape & Research Challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, p. Art. 14, 2009.

[10] D. Garlan, S.-W. Cheng, A.-C. 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.

[11] D. Menasce, H. Gomaa, S. Malek, and J. Sousa, "SASSY: A Framework for Self-Architecting Service-Oriented Systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, 2011.

[12] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, "Models@Run.time to Support Dynamic Adaptation," *IEEE Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[13] T. Vogel and H. Giese, "A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels," in *Proc. of SEAMS*, 2012, pp. 129–138.

[14] E. Patouni and N. Alonistioti, "A Framework for the Deployment of Self-Managing and Self-Configuring Components in Autonomic Environments," in *Proc. of WoWMoM*, 2006, pp. 480–484.

[15] H. Liu, M. Parashar, and S. Hariri, "A Component-Based Programming Model for Autonomic Applications," in *Proc. of ICAC*, 2004, pp. 10–17.

[16] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - A Component System for Pervasive Computing," in *Proc. of PerCom*, 2004, pp. 67–76.

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

[18] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM, 2004.

[19] I. Welch and R. J. Stroud, "Kava - Using Byte code Rewriting to add Behavioural Reflection to Java," in *Proc. of COOTS*, 2001, p. 9.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 – Object-Oriented Programming*, ser. LNCS, 1997, vol. 1241, pp. 220–242.

[21] F. M. Roth, C. Krupitzer, and C. Becker, "Runtime evolution of the adaptation logic in self-adaptive systems," in *Proc. of ICAC*, July 2015, pp. 141–142.

[22] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "BASE – a Micro-broker-based Middleware for Pervasive Computing," in *Pro. of PerCom*, 2003, pp. 443–451.

[23] S. Tomforde, H. Prothmann, F. Rochner, J. Branke, J. Hähner, C. Müller-Schloer, and H. Schmeck, "Decentralised Progressive Signal Systems for Organic Traffic Control," in *Proc. of SASO*, 2008, pp. 413–422.

[24] E. M. Fredericks, B. DeVries, and B. H. C. Cheng, "Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty," in *Proc. of SEAMS*, 2014, pp. 17–26.

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