Towards Reusability in Autonomic Computing

Christian Krupitzer*, Felix Maximilian Roth*, Sebastian VanSyckel*, and Christian Becker*
*University of Mannheim
Schloss, 68131 Mannheim, Germany
Email: {christian.krupitzer, felix.maximilian.roth, sebastian.vansyckel, christian.becker}@uni-mannheim.de

***Abstract*—Reusability of software artifacts reduces development time, effort, and error-proneness. Nevertheless, in the development of autonomic systems, developers often start from scratch when building a new system instead of reusing existing components. Many frameworks offer reusability on a higher level of abstraction, but neglect reusability on the lower component implementation level. In this short paper, we present a reusable adaptation logic by separating the generic structure and mechanisms of Autonomic Computing systems from its custom functionality. That is, we provide a reusable communication architecture with abstract component templates that enables a faster development and easier runtime adaptation. We evaluate our approach in a case study with two implementations.**

I. Introduction

Modern information systems experience various developments that lead to tremendous changes and increase the complexity of development, deployment, as well as maintenance. Developing, configuring, and maintaining such systems is a very difficult, error-prone, and time-consuming task. Different research directions address the need of self-adaptation of systems. In Autonomic Computing, this issue is addressed with *self-adaptive systems* (SASs) that are able to automatically modify themselves in response to changes in their operating environment ([1], [2]). The modification of their behavior is done by adjusting attributes (parameters) or artifacts of the system in response to changes in the system itself or in its environment. Complexity is not only a reason for using SASs, it is itself a challenge in SAS development [3]. Complexity in the development of a system can be reduced by reusing elements. Further, the development of these systems can be fastened by improved reusability of components and processes ([4], [5]). There are various elements of the adaptation logic (AL), or autonomic manager [2], respectively, that can be reused, such as communication structures, common elements of the AL (e.g., the MAPE functionality [2]), or structures for handling knowledge.

In this paper, we introduce an approach for building reusable ALs. It separates reusable parts as decentralization structures, communication patterns, and knowledge representation, from parts that need customization, such as reasoning algorithms. Our main contributions are: First, we provide a template for the AL based on the MAPE [2] concept. This template is enhanced by a context manager, which simplifies interaction with the surrounding environment, as well as reasoning on its information [6]. Second, we present a reusable, decentralized communication structure for communication within the AL based on the publish-subscribe concept. Last, we evaluate our approach and show, how reusability can speed up development.

The structure of the remaining part reflects these contributions. Section II introduces reusability within the AL and examines related work. In Section III, we discuss our system model. Section IV presents our case study. There, we describe the implementation of two systems using the templates and evaluate the degree of reusability. Finally, the outlook in Section V sketches how we will use our results for further supporting developers in the implementation of ALs, up to partly automating the development [7].

II. Reusability in Autonomic Computing

Autonomic Computing systems are divided into two parts: the *adaptation logic* (AL), or *autonomic manager* [2], respectively, and the *managed resources* (MRs). Whereas the MRs deliver the system functionality, the AL is responsible for managing the resources and adapting them. Both are connected via interfaces that enable *self-awareness* and *context-awareness* [8]. Self-awareness describes the ability of a system to be aware of itself, i.e., to monitor its resources, state, and behavior. Context-awareness means that the system is aware of its operational environment, the so-called *context*. Within the AL, a control structure processes the information gathered about MRs. One of the most prominent examples for such a control structure is the MAPE cycle, where 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 [2].

Often, an SAS is a system-of-systems. In such systems, the MAPE functionality can be distributed and, therefore, decentralization must be supported by the AL [6]. Different patterns are suitable [9], which basically differ in the distribution of the MAPE functionality: (i) each subsystem has full MAPE functionality with local decision making (no coordination and exchange of information), (ii) each subsystem has full MAPE functionality and there is coordination within one MAPE functionality (e.g., plan components coordinate their activity for global adaptation planning), or (iii) not every subsystem has full MAPE functionality (e.g., the regional planning pattern [9], where monitoring, analyzing, as well as execution is done locally and planning is done by central planners). These different blueprints for the AL, as well as decentralization patterns (such as [9]), are two examples for reusable elements in the fields of Autonomic Computing and SAS, respectively.

Service-oriented frameworks for SAS development, such as the *SASSY* framework [10], support developers in the implementation of service-based SASs. Architecture-based solutions, such as the *Rainbow* framework [11], the *ArchStudio* tool suite [1], or Kramer and Magee's layer approach to self-management [12] offer components or layers that control the adaptation. However, in such approaches the support for the developer is often reduced to an approach for building the AL, lacking processes for design as well as specific implementation guidelines or components ([4], [5]). Many SASs use model-based approaches for analyzing and planning, e.g., based on *runtime models* [13]. These modeling languages are reusable, but often, they have to be adjusted to domains. Reusability of generic components for different services is not addressed in the approaches above. In [14], the authors describe a framework with a template for generic autonomic components. Nevertheless, they focus on the deployment of the components rather than on reusing (parts of) their functionality. King *et al.* presented a reusable object-oriented structure for self-testing autonomic software [15]. Here, the focus is on testing different policies. Component-based solutions are present in Autonomic Computing, too (e.g., [16], [17]). Nevertheless, their focus is on structural adaptation by exchanging components and not on how to build reusable components where fine-granular parts of the components itself can be exchanged.

All of the presented approaches address reusability on a higher level of abstraction focusing on which components should be present in the AL or how to exchange components or policies dynamically. To the best of our knowledge, there is no approach that focuses on the internal implementation of the components. For illustration purposes why this would be beneficial, we sketch a cloud service scenario comparable to the Amazon EC2 cloud. Different servers offer resources for running virtual machines on them. If the resource utilization exceeds a threshold, additional server instances can be started. If the start is after the identification of exceeding the threshold, the adaptation is reactive. However, adaptation can also be proactive, which lowers delays introduced by reconfiguration [18]. So changing to a proactive mode would be beneficial, but would make proactive analysis necessary. The rest of the system does not need to change. In common component-based adaptation approaches (e.g., [16], [17]), the whole component needs to be exchanged. By introducing adaptation within a component, e.g., through dynamic code reloading/rewriting [19] or aspect-oriented techniques [20], we can keep the component's structure as well as the communication and data handling mechanisms, but only replace the functionality of the reactive analysis algorithm with a proactive one. Additionally, this simplified exchange of the functional logic facilitated by a separation of functional logic and the component's communication and data handling mechanisms enables faster development of components, as the functional logic only must be adjusted and the rest can be reused. In the next section, we discuss our approach for reusability of the AL's components, by offering an approach for building generic components that are customizable through exchangeable subcomponents.

III. Our Model of a Reusable Adaptation Logic

In this section, we present our system model for a reusable AL. The system model is part of the FESAS framework. The goal of the FESAS framework is to provide tools for supporting the development and automating the maintenance of SASs. Therefore, we divide design activities – which are performed by the developer at design time – and runtime activities – which are in the responsibility of FESAS [7]. This section is threefold. First, we present a template for a reusable AL based on the MAPE concept. Second, we detail the structure of a single AL component and explain, how we increase reusability there. Last, we introduce a reusable communication approach based on the publish-subscribe concept.

A. The FESAS Adaptation Logic Template

For building the AL, there are different approaches present in literature. The MRs and the AL can be integrated (*internal approach*) or separated (*external approach*). As we aim at a reusable AL, the AL should be separated from the MRs for improving reusability [6]. All approaches have in common that the AL is composed of ([21], [6]): (i) control structure components (including components for the connection to the MRs) and (ii) supporting components. These components are accompanied with patterns for the self-* properties and distribution/communication. Figure 1 shows the elements of our AL template. In the following, we introduce these elements.

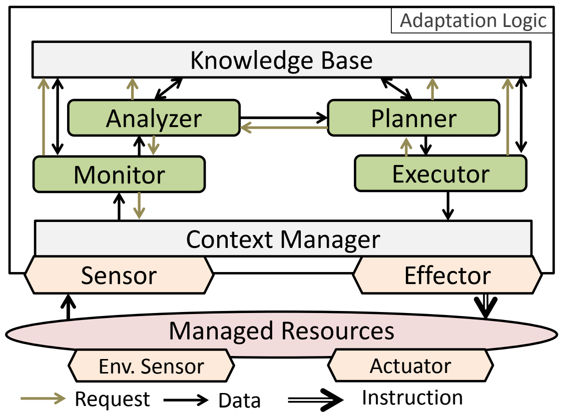

Fig. 1. The FESAS Adaptation Logic Template.

Within the Autonomic Computing community, the MAPE-K model [2] is the most prominent approach for an AL control structure [8]. In our system model, the MAPE functionality is represented by the components *monitor*, *analyzer*, *planner*, and *executor*. The *knowledge* component acts as a central repository for all kinds of data used in the AL, and supports the MAPE functionality. Often, an SAS is a system-of-systems and the AL is distributed in these cases. Therefore, the AL template must offer flexibility for decentralization and distribution of its components. This requirement is addressed in our system model, as the components for the MAPE functionality are optional for a specific subsystem, but must be present in the global view of the SAS. Furthermore, one subsystem can have multiple MAPE loops, e.g., for supporting different self-* properties [2].

Following [2], our system model provides sensors and effectors for interaction with the MRs. *Sensors* capture information about the resources themselves (self-awareness), information about the environment (context-awareness) – the MRs collect the information with *environment sensors* – and information about the users. An *effector* is used by the AL to adapt the MRs or to adapt the environment via *actuators* of the MRs.

Additionally to the MAPE functionality, we introduce the *context manager*. In [6], from a pervasive computing perspective, we motivate a better integration of context (adaptation) into the traditional model of self-adaptive systems, as the adaptation of resources can influence the context, which again can make adaptation necessary. The context manager receives context information (which is gathered through environment sensors of the MRs), aggregates the data of sensors, and maintains a context model. Furthermore, the context manager integrates an architectural model of the MRs with information about the status of these resources. It fulfills the functionality of an aggregator as well as a broker [21]. The monitor works with the data of the context manager. The executor interacts with the context manager and transmits the execution instructions to it. The context manager maps the execution instructions to its context model and determines which effectors are responsible for performing an instruction. By performing the instructions, the MRs or the system's environment via the MRs actuators, are adapted. This way, the MAPE functionality is decoupled from the low level data of the MRs and the context. Therefore, the MAPE components are separated from the sensor and effector data and can operate on a higher level of abstraction. This improves the reusability of the MAPE components and algorithms within these components, as they can operate on a higher abstraction and do not need to be customized for each use case.

B. The FESAS Component Template

In this section, we present the template for one reusable component within the AL, such as a component having MAPE functionality. Our template is independent from a specific implementation. Based on the *Template Method* pattern [22], the MAPE components are composed of an exchangeable logic (e.g., for an analyzer, this would be an algorithm for analyzing the monitored data) and logics for communication and data handling (knowledge in [2]). Interfaces enable receiving and sending data to other components as well as requesting data from other components. The communication logic offers an implementation of these functionalities. Figure 2 shows the template for a component.

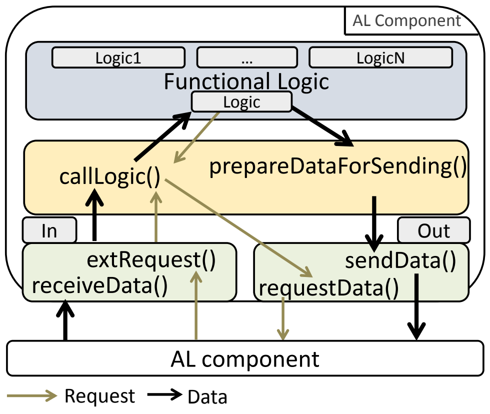

Fig. 2. The FESAS Component Template.

The split of the functionalities in reusable communication and data handling subcomponents, as well as customized functional logic enables already reuse of subcomponents. In object-oriented programming languages, inheritance offers increased reusability of code. In our approach, an `AdaptationLogic` class can act as superclass for all other elements in the AL. Specific components as the MAPE components can inherit most methods, e.g., sending/receiving methods or the basic component logic that controls the workflow and calls the subcomponents. So it is possible to reuse the `AdaptationLogic` superclass as well as the communication structure as base for all AL components within a system. Further, it can be reused without modifications in any system implemented in the same programming language and that is using the same communication approach. This basic component set up is accompanied by a specific implementation for the functional logic. Functional logic elements can inherit most methods from an abstract logic. Only the `callLogic()` method has to be customized.

The component template enables reusability in different aspects. First, the component itself is reusable. It offers a skeleton of methods for calling the functional logic, communication, and knowledge handling. Furthermore, exchanging the functionality is simplified as the interfaces stay stable. Additionally, it is possible to use the same component skeleton with implementations for knowledge and communication for all MAPE functionalities. Only the functional logic must be customized to its purpose. Second, reusability is offered as the interfaces provide generic mechanisms for communication and knowledge handling. Last, reusability is enabled through the context manager. As the context manager can abstract from a specific use case, algorithms can be reused in various use cases. Of course, it is possible to use the component template for the context manager itself in order to simplify its development.

C. A Reusable Publish/Subscribe Communication Approach

The heterogeneity, dynamism, and the distributed nature of an SAS pose several challenges for communication. An SAS needs to integrate different implementations of components as well as different implementation languages. These requirements are fulfilled by using the FESAS AL template and by separating logical and physical communication [6]. As the AL can be decentralized and the location of functionality can change at runtime, the communication shall not be coupled to specific elements, but instead to the information itself (requires *space decoupling*). Furthermore, the different subsystems of the AL may have various MAPE loops. Their activities are not synchronized (requires *synchronization decoupling*) and have varying execution times due to different hardware, algorithms, and purposes (requires *time synchronization*). In line with [23], we propose to use a *publish/subscribe* (pub/sub) approach for communication, as it offers decoupling from space, time, and synchronization, and is not limited to a specific implementation approach or language [24]. Pub/sub approaches enable the decoupling from sender and receiver. If a component wants to receive some specific data, it can register itself at the pub/sub service and becomes a *subscriber*. The registration is done based on topics (predefined keywords), content (e.g., event properties), or event types and can be event-specific or for a pattern of events [24]. If a component wants to make data available, it sends the data to the pub/sub service and becomes a *publisher*. The pub/sub service can be a central component or decentralized.

The concept of our reusable AL structure makes it possible to reuse communication structures. In the following, we present a reusable pub/sub approach. In our approach, components register based on *information categories* and *information types*. Categories (CONTEXT, MONITORING, ANALYZING, PLANNING, EXECUTING, SENSOR, and EFFECTOR) relate to the components of the AL template. Types are a fine granular division of an information category – e.g., for analyzing in the server example in Section II, it could be ANALYZING_WORKLOAD or ANALYZING_VMREQUIREMENTS – which allows to reduce the likelihood of receiving events that are not relevant for a component, but requires more knowledge about the domain and the type of data that is available at runtime. This is a topic-based pub/sub approach [24], as information category and type are keywords for filtering. Our pub/sub service can be configured to use either information categories, information types, or both. Furthermore, our approach divides between registration of events of a specific system (compared to event-specific registration) vs. registration for all systems (event patterns). Besides other properties, the event has a `knowledge ID` which enables subscribers to load data that is related to the event. Subscribers may first receive the event and request relevant data with this `knowledge ID`. The process of registration and publication of events is shown in Figure 3. Naming is in accordance with typical pub/sub systems [24]. In the following, we discuss subscription and publication of events.

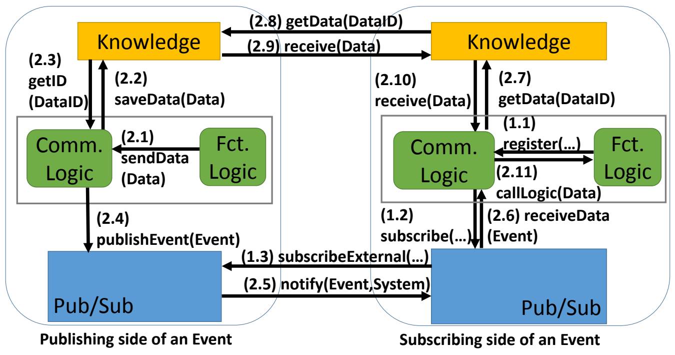

Fig. 3. A Reusable Pub/Sub Approach for Communication.

Subscription for events: The functional logic of a component specifies which data it needs *(1.1)*. The communication logic uses the appropriate `subscribe()` method *(1.2)*, depending on whether it uses the information type or category and whether the scope of the registration is one specific subsystem or the entire system. The pub/sub service adds the subscriber to its list for the specified events. If the subscription is for a local component, the subscription is completed at this stage. For a remote component, the local pub/sub service subscribes for relevant events at the remote pub/sub service(s) of the corresponding subsystem(s), or all systems, respectively *(1.3)*. Unsubscribing for events follows the same process.

Publishing an event: If a functional logic calls the `sendData()` procedure *(2.1)*, the communication logic first saves the data *(2.2/2.3)*, and, then calls the `publishEvent()` procedure *(2.4)*. The pub/sub service notifies all pub/sub services that have subscribed for the event *(2.5)*, and transmits the event. After receiving an event, the pub/sub service hands the information to the subscribed components *(2.6)* which use the communication logic for reading the data. If the event is relevant for a component, the communication logic loads the event's data *(2.7 - 2.10)*, and starts the `callLogic()` procedure with the data *(2.11)*.

The configuration of the pub/sub service (using information type vs. category for filtering, and the scope of subscription) can be adapted at runtime. The decentralized nature of our pub/sub system enables high flexibility. So far, we implemented basic functionality in the pub/sub system. Further functionality for improved data security and backup mechanisms will be integrated later on.

D. Automated System Deployment with Configuration Files

Configuration files determine system deployment. These files specify which subsystem starts which AL components. Furthermore, a configuration file determines the communication between components (initialization of the pub/sub system). Distribution as well as communication structures are saved in XML files. At start, each subsystem reads the specified configuration files, starts the corresponding AL components, adjusts settings of the components based on the files, and loads the specified functional logic elements. The communication files can be reused in different settings with minor changes, only. Furthermore, they support the reusability as these files configure the generic components, e.g., specify which data is used in a component and which functional logic is started.

IV. CASE STUDY

In the preceding section, we presented our system model for reusability in SASs. In this section, we show the feasibility of our concept in a case study. The focus is on the implementation of the AL, rather than on the communication.

A. Use Cases

We implemented an SAS for controlling the traffic flow on a highway and an SAS for data center management using the FESAS AL template and FESAS AL component template. In the following, we present the two use cases.

1) Smart Highway Use Case: In accordance with literature (e.g., [25]) we built an SAS for traffic management. In the scenario, digital traffic signs are installed on a highway. The SAS's AL controls the signs. The signs (= MRs) with its controlling software (= AL) form an SAS. The signs show the speed limit depending on the amount of cars and the traffic flow. As high utilization can lead to traffic jams, if the flow is not regulated, the objective is to analyze such situations and react accordingly by showing speed limits. We assume that self-driving cars drive on the highway. As the AL control these cars indirectly, self-driving cars are manageable resources, too. The traffic simulator *SUMO*[1] simulates the traffic flow as well as the infrastructure. *TraCI*[2], an addition to SUMO, allows to change parameters during simulation, and, therefore, to simulate the traffic signs. The track is divided into four sections. For each section, one subsystem with AL components gets data from simulated cameras, and controls the traffic signs in its section. In the first setting, each section has its own dedicated AL. In the second setting, we grouped two sections to a region. The regional planning pattern [9] enables region-wide planning with a single planning component receiving data from both analyzers, but not system-wide planning as the planners do not communicate with each other.

[1]http://www.dlr.de/ts/sumo/en/
[2]http://sumo.dlr.de/wiki/TraCI

2) Data Center Use Case: For showing the degree of reusability, we implemented a second use case. The second use case is a simulation of a cloud environment as introduced in Section II. Different data centers host servers. A load balancer distributes requests for hosting a virtual machine (VM) on these servers. Each data center has one AL for self-management, i.e., the AL starts new servers in case of high utilization or server defects, transfers VMs from one server to another in the same or another data center for distributing the workload, and shut down servers in case of reduced workload. We simulated four data centers and the servers. In the first setting, each data center has its own dedicated AL and the data centers do not collaborate. In the second setting, we implemented two groups with two data centers each. The regional planning pattern [9] enables group-wide planning without collaboration of the planners.

B. Implementation and System Deployment

We implemented both use cases with our templates. A large part of the code is independent from a specific use case. Next, we describe how to use our templates for development, and how to deploy the components.

1) Implementation Using the AL Templates: The prototype is service-based, as many SASs use service-oriented techniques [8]. The BASE middleware [26] enables communication between the AL components. BASE is a lightweight, service-oriented middleware for pervasive systems implemented in Java. We implemented each component as BASE services. As we use inheritance, most of the methods are implemented in the `AdaptationLogic` service. For a specific component, such as monitoring, we only have to specify which functional logic has to be loaded, and which type of data has to be processed. The same holds true for the functional logic elements. As most of the functionality is inherited from the `AbstractLogic` element, for a specific logic, only the `callLogic()` function has to be implemented. We built the data center use case based on the highway use case. That is, we reused the communication logic, knowledge handling, and the component structure itself without any modification. We only had to change the functional logic of the AL components and do minor changes in sensor, effector, and context manager. Furthermore, the data types and properties used in the AL must be customized in most cases (or defined in a generic usable notation on a high level of abstraction, such as key-value pairs).

2) System Deployment with Configuration Files: We use the same configuration files for system deployment in both use cases. Only the information type, the name of the functional logic that should be used, and the amount of sensors and effectors vary. Both, the distribution of the MAPE functionality as well as the communication structure can be reused.

C. Evaluation

In this section, we provide and discuss some metrics that show the level of reusability of our approach.

1) Reusability on the Component Level: First, we investigate a specific component and analyze the degree of reusability. As an example, we detail the `Analyzer` of the highway use case. The component is composed of: (i) an `AdaptationLogic` superclass with a corresponding interface, (ii) the `CommunicationLogic` with a corresponding interface, (iii) a reusable `Analyzer` implementation with a corresponding interface, and (iv) the functional logic, which is composed of an `AbstractLogic` with a corresponding interface, and a subclass with a customized `callLogic()` implementation. The implementation has in total 538 lines of code (LOC). The superclass with the communication logic and the abstract logic has 439 LOC (81.7% of the component's code) all together. The structure of these three elements is the basis for a component and reusable for all components. The analyzer subcomponent adds additional 29 LOC (5.3%), but this structure is further reusable for all analyzer components. The functional logic itself has 70 LOC (13%). Only this small amount of code is customized to the use case. The customization influences the `callLogic()` method, as well as the setting of some parameters in the `Analyzer` and the functional logic. The rest of the component can be reused for analyzing components or (excluding the additional 29 LOC for the analyzer function) for any other type of AL component.

2) Reusability on the System Level: Second, we analyze the degree of reusability on the system level. The results are shown in Table I and explained in the following.

	Smart Highway Use Case		Data Center Use Case	
	Setting 1	Setting 2	Setting 1	Setting 2
Reusable AL components	7,281	7,281	7,289	7,289
Functional logics (without P)	390	390	254	254
Planner functional logic	84	96	74	74
Functional logics data	279	279	133	133
Total LOC	8,042	8,054	7,750	7,750
Share of reused code	90.6%	90.5%	94.1%	94.1%

TABLE I. EVALUATION RESULTS (NUMBERS INDICATE LOC; PERCENT VALUES INDICATE SHARE OF SYSTEM'S LOC).

The implementation of the reusable parts of the AL has 7,281/7,289 LOC including reusable data types. In theory, this part can be reused for any use case where the BASE middleware is used for communication within the AL. It offers an implementation for the AL components, except the functional logic elements. The context manager needs minor customization to adjust for the context model used (modifications of the context manager leads to the small difference in LOC). Further standardization through using present context models can improve reusability even further. The functional logics of all components (including the planner and context manager logic) have 474 LOC (5.9% of the system's code) for highway setting 1 and 486 LOC (5.9%) for highway setting 2. Only the functional planning component has to be changed for a different functionality (adaptation for small sections vs. adaptation of (large) areas). In the data center use case, the functional logics have 328 LOC (4.2%). There is no difference for the two settings as the implementations are the same. In addition, developers have to specify the data that is used within an AL. The data of the functional logics adds 279 LOC (3.5%) in the highway scenario, and 133 LOC (1.7%) in the data center scenario, respectively.

3) Reusability in the System Deployment: As written in the implementation part, we use the same configuration files for both use cases. We only change the name of the logic identifiers, the type of data, and the configuration of effectors and sensors. Therefore, it is possible to describe decentralization patterns with the configuration files and reuse them. So, the configuration files are a key element for enabling reusability of the components, as they contain the parameters for the AL.

V. Conclusion and Future Work

According to White *et al.* [21], an autonomic system is composed of (i) control structure components, (ii) supporting components, and (iii) design patterns for self-management (including patterns for distribution/communication [6]). In this work, we introduced a reusable AL template. Furthermore, we presented a template for reusable components. Both is accompanied by a reusable communication mechanism based on the pub/sub paradigm. We used the templates and the communication approach for building two adaptive systems in different use cases. We showed, that we can reuse up to 94% of the source code in the use cases. Furthermore, the reusable design in combination with the configuration files enabled adjustment of the use cases to the alternative settings, by only changing one procedure. So far, the integration of patterns is restricted to specifying communication/distribution patterns in the configuration files. In future work, we will integrate design patterns for the self-* properties.

So far, loading functional logics is hard-coded with an if/else structure. In future work, the information of the XML configuration file will be used for further automating the system deployment. Therefore, a central FESAS deployment tool will control the deployment, initialize the start of the AL components on the subsystems (load the specified functional logic), and implement the communication structure. We will provide a generic mechanism for loading the logic from a repository based on contracts using relevant logic metadata [17], which will enable runtime adaptation, as well. This reduces the customization effort even further, as the mechanism for dynamic code loading will support all types of AL, and the loading of code does not need to be specified and customized in the components. Additionally, tools like Eclipse plug-ins will support the creation of configuration files.

In general, we showed that our approach supports reusability on three levels: the component level, the system level, and the system deployment. These levels are of relevance in the development and for starting the system. However, our approach to reusability also supports exchange of components at runtime. The future integration of reflective mechanisms enables the use of the template as a basis for runtime adaptation.

Further, we will evaluate the communication mechanisms in detail. We will focus on the different possible configurations as well as how to establish synchronization of pub/sub data with the MAPE functionality.

References

[1] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Int. Sys.*, vol. 14, no. 3, pp. 54–62, 1999.

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

[3] C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds., *Organic Computing – A Paradigm Shift for Complex Systems*. Springer, 2011.

[4] 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.

[5] 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.

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

[7] C. Krupitzer, S. VanSyckel, and C. Becker, "FESAS: Towards a Framework for Engineering Self-Adaptive Systems," in *Proc. of SASO*, 2013, pp. 263–264.

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

[9] 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.

[10] 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.

[11] 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.

[12] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *Proc. of FOSE*, 2007, pp. 259–268.

[13] 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.

[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*. IEEE, 2006, pp. 480–484.

[15] T. M. King, A. Ramirez, P. J. Clarke, and B. Quinones-Morales, "A Reusable Object-Oriented Design to Support Self-Testable Autonomic Software," in *Proc. of SAC*, 2008, pp. 1664–1669.

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

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

[18] S. VanSyckel, D. Schäfer, G. Schiele, and C. Becker, "Configuration Management for Proactive Adaptation in Pervasive Environments," in *Proc. of SASO*, 2013, pp. 131–140.

[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 et al., "Aspect-oriented programming," in *ECOOP'97 – Object-Oriented Programming*, ser. LNCS, 1997, vol. 1241, pp. 220–242.

[21] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart, "An architectural approach to autonomic computing," in *Proc. of ICAC*, 2004, pp. 2–9.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[23] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime Software Adaptation: Framework, Approaches, and Styles," in *Proc of ICSE*, 2008, pp. 899–910.

[24] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comp. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[25] S. Tomforde, H. Prothmann, F. Rochner, J. Branke, J. Hähner et al., "Decentralised Progressive Signal Systems for Organic Traffic Control," in *Proc. of SASO*, 2008, pp. 413–422.

[26] 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.