**High-Quality Specification of Self-Adaptive Software Systems**

Markus Luckey, Gregor Engels
University of Paderborn, Germany
{lucky,engels}@upb.de

**Abstract**—Today's software systems have to cope with changing environments while at the same time facing high non-functional requirements such as flexibility and dependability. Recently, these non-functional requirements are addressed using self-adaptivity features, that is, the system monitors its environment and adjusts its structure or behavior in reaction to changes.

In classical model-driven software engineering approaches, self-adaptivity introduces additional complexity since self-adaptation features are distributed in a cross-cutting manner at various different locations in the models, resulting in a tightly interwoven model landscape that is hard to understand and maintain. A particular solution to cope with this problem is the separation of concerns (SoC) to focus on the specific concern of self-adaptivity and allow in-depth analyses. Applying SoC requires suitable development processes, languages, and techniques, e.g., for quality assurance, to be available.

In this paper, we present a method for the specification of self-adaptive software systems using a UML based concern-specific modeling language called Adapt Case Modeling Language (ACML) that allows the separated and explicit specification of self-adaptivity concerns. Based on formal semantics we show how to apply quality assurance techniques to the modeled self-adaptive system, which enable the provisioning of hard guarantees concerning self-adaptivity characteristics such as adaptation rule set stability and deadlock freedom. Further, we show how the language and techniques integrate with existing software development processes.

**I. INTRODUCTION**

Today's systems face high requirements regarding flexibility, dependability, fault-tolerance and context-awareness at the same time. These requirements are recently addressed by introducing self-adaptivity features. Thereby, these systems—called self-adaptive systems—are capable to autonomously react to changes in their environment. Examples for self-adaptation capabilities include mobile devices that switch from wireless LAN to GPRS, rack server systems that outsource data into the cloud, or service-oriented business information systems that replace a service that exhibits low quality.

While self-adaptation capabilities always existed in software systems to a certain degree, their increase poses new challenges to software engineering, e.g., making it hard to show correctness of these systems early during design time. Modern software engineering addresses this problem using constructive and analytical methods. **Constructive methods** aim at improving the quality during creation of specification documents. To this end, common approaches utilize the principle of separation of concerns, e.g., by using concern-specific modeling languages (CSML) for separating self-adaptation concerns from business logic concerns. **Analytical methods** help improving the quality by analyzing the existing models for specific properties such as adaptation rule set stability or deadlock freedom. Analytical methods include static and dynamic approaches. Since dynamic approaches such as testing require an implementation of the system to be present, they cannot be applied in early system design phases. In contrast, static analysis such as model checking allows the early quality assurance since it relies on the design models.

**Contribution.** We propose a specification method for self-adaptive systems consisting of constructive and analytical methods that may be integrated with existing UML-based software engineering methods. In earlier work [1], we motivated the need for explicit separated specifications of self-adaptivity in software systems and provided the basic concepts needed therefore. Based on and extending the concepts of [1], in this paper, we propose a concrete UML-based CSML named **Adapt Case Modeling Language** (ACML) that allows the specification of structural and behavioral concerns of self-adaptive software systems. We further advance the ideas of [1], by utilizing a semantics specification language for meta-model based languages (DMM) [2] to enable the quality assurance (i.e. model checking) of self-adaptive software systems early in the development process. A distinctive feature of our quality assurance approach is that model checking is performed transparently for the modeler. Thereby, our approach helps the system modeler to understand why the system model violates certain quality properties, and how the system model can be fixed, without any knowledge of formal quality assurance techniques such as model checking.

**Outline.** In Section II, we will sketch a scenario—used as running example—which is inspired by a real-life project conducted in the Software Quality Lab (s-lab) at the University of Paderborn. Section III briefly lists the supported modeling dimensions of our approach's language. In Section IV, we will describe the proposed concern-specific language for self-adaptive systems and in Section V, we describe how the language may be integrated with existing software engineering methods. In Section VI, we will show how we fully automatically apply model checking techniques to assure high model quality. In Section VII we briefly describe what exactly the system designer has to do to use our approach. Section VIII
presents evaluations of our approach, and finally, in sections IX and X, we discuss related work and conclude the paper.

II. Scenario

As an example, we use the scenario of a rack server system supported by cloud resources. The main purpose of the system is to provide the infrastructure for delivering web pages to end users. In case of resource bottlenecks additional computation resources may be obtained from the cloud. A non-functional requirement in this scenario is the operation of the rack server system at minimum cost, i.e. in particular the number of used cloud resources (pay-per-use) must be minimized at all times. Other requirements demand for an automatic fan management depending on the number of used servers as well as an automatic server management depending on the current workload. The functionality of server management, fan management, and cloud resource integration can be considered as self-adaptive behavior, while the load that is generated by user requests is part of the uncertain environment that has to be monitored.

The challenge when specifying this scenario without explicit modeling techniques for self-adaptivity are as follows: During the first operationalization of the requirements (e.g., using plain UML use cases and activity diagrams) this self-adaptation functionality must be woven into the software specification at many different locations which does not only spread related information over the system but possibly hides it, too. Compared to a non-adaptive system where the number of active servers is fixed, the specification of the self-adaptive rack server system exhibits increased complexity, which on the one hand hardens the comprehensibility, and on the other hand abets the introduction of severe specification flaws. If for example the functionality that activates and deactivates servers in the rack is insufficiently specified, the continuous activation and deactivation of servers for different reasons might lead to an unstable system behavior (as shown in Section VI). If the required self-adaptation features are more complex (e.g., considering the adjustment of business processes) the situation gets even more difficult, and thus, error-prone. Hence, even this small scenario clarifies the need for dedicated techniques to specify self-adaptive systems.

III. Modeling Dimensions of Self-Adaptivity

To give a brief characterization of our proposed modeling language upfront, we will enumerate modeling dimensions for self-adaptive systems in the following, few of which have been borrowed from the Modeling Dimensions by Andersson et al. presented in [3]. The modeling dimensions pose requirements to a comprehensive concern-specific modeling language for self-adaptive systems. For each dimensions described below, we will state how it is supported by our proposed language, the ACML. We distinguish the three main dimension categories Adaptation Reasons, Adaptation Subject, and Modeling Mechanisms.

Adaptation Reasons. Adaptation Reasons can be described by self-* properties. Aligned with the definition in [4], we further distinguish self-optimizing, self-protecting, self-configuring, and self-healing. Since the ACML is a general purpose adaptation language, all four self-* properties can be addressed with the ACML.

Adaptation Subject. The Adaptation Subject is further divided into scope, model type, and artifact & granularity. Considering the scope, the ACML allows to specify the adaptation of the type (permanent change) and of instances (temporary change) of a model. The ACML defines strategies to cope with existing instances of changed types. Allowing the specification of changes on type level supports a clean model design since explicit dealing with sets of instances, etc. is avoided. Considering the model type, the ACML allows the specification of adaptation of the system’s structure, behavior and even the adaptation specification itself. Finally, concerning the artifact & granularity of the adaptation subject, the ACML allows for parameter and state changes (e.g., changing a server’s running state), as well as for compositional changes (e.g., replacing a particular component by several others).

Modeling Mechanisms. Finally, considering the modeling mechanisms, the characteristics are timing, triggering, anticipation, automation, and type of specification. Regarding the timing, the primary purpose of the ACML is to allow reactive adaptation, whereas proactive adaptation is partially supported by the use of specific histories. ACML allows the specification of time and event triggered adaptation, while non-anticipated change is not supported. The ACML is designed for the specification of fully automatic self-adaptation. However, human interaction could be introduced with concepts such as proposed with BPEL4PPL. Finally, the type of specification using the ACML is explicit and imperative.

IV. Language for Self-Adaptive Systems: ACML

To come up with a good language, in [1] we analyzed different adaptivity specification and implementation methods, which are used during the engineering process. Most of the methods we investigated are either using or at least compatible with control-loops. The well-known control-loop model MAPE-K proposed by IBM [5] provides a process-oriented reference model for realizing adaptive behavior. The MAPE-K control loop model is shown in Figure 1. The model consist of a monitor that observes a managed element (i.e. the system and the environment) using sensors, an analyzer that analyzes the monitoring results, a planning component that decides for an appropriate adaptation action, and an execution component that executes the chosen adaptation action using effectors. All phases use a shared knowledge base. The ACML takes up these concepts. It is divided into a behavioral part, the Adapt Cases, and a structural part, the Adaptation View. Adapt Cases model the four phases monitor, analyze, plan, and execute, while the first two phases as well as the last two
A. Behavioral Adapitivity Modeling: Adapt Cases

The behavioral part of the ACML, the Adapt Cases, allows the modeling of adaptation rules on a logical platform-independent level. Adapt Cases are based on traditional use cases and follow the model of control-loops [5]. That is, the information provided by the Adapt Cases is used to monitor the system, analyze the gathered information, plan an appropriate adaptation action, and finally execute this adaptation action. We map those concepts first identified in [1] to the UML as well as extend the UML to support a UML-based specification of self-adaptive systems. In the following, we briefly describe the abstract and concrete syntax as well as the language’s semantics.

Abstract Syntax. Figure 2 shows the excerpt from the meta model which defines Adapt Cases and their features. AdaptCases inherit from use cases and classes. Typed as use cases, Adapt Cases inherit all options of use cases, such as the association with an actor, include, extend, and similar relationships such as adapt, as well as the mechanism of extension points (not shown in Figure 2). We inherited from class, too, in order to allow the definition of signal receptions in Adapt Cases. That is, an Adapt Case might react to a signal that is received from the environment.

An Adapt Case separates the description of monitoring and adaptation activity. Both the monitoring and the adaptation activity are modeled with UML activities. Being a BehavioredClassifier, an Adapt Case may have Behaviors. An activity is a Behavior, which we further specialized to AdaptationActivity and MonitoringActivity to explicitly demonstrate the concern-specific concepts. An activity may contain a variety of different actions. We reused some existing UML actions for instance manipulation of UML models and added further actions for type manipulation, i.e. to allow specifying the adaptation of the system’s type. All actions operate on the adaptation interfaces (i.e. sensors and effectors) that are defined in the Adaptation View (see Section IV-B).

Additional concepts, such as adaptation triggering events, have been defined and are described in detail in [6].

Concrete Syntax. Figure 3 shows an example Adapt Case for autonomous server management. The upper activity diagram reflects the monitoring definition. Each 5 minutes, it checks the load and eventually starts an adaptation. The monitoring definition contains the CallAdaptationActivityAction (circled A) that triggers adaptation if the system’s load is out of range. The lower activity diagram contains the adaptation specification. Based on the availability of local servers, either local or cloud servers are activated or deactivated. For more sophisticated Adapt Cases, please refer to [6].

Semantics. Basically, the semantics of Adapt Cases are based on the UML activity semantics. Additionally introduced actions were equipped with new semantics descriptions. Adapt Cases use activities and actions within the monitoring and adaptation activity definitions. In other words, Adapt Cases embrace activity diagrams. This bracket is responsible for relating monitoring definitions to adaptation activities and to trigger both monitor and adaptation activity if applicable. Another important semantic definition is the precedence of adaptation logic over business logic. That is, whenever an Adapt Case is applicable, it is assumed that there is always enough time to execute the Adapt Case prior to any change in the environment or system. This definition is due to the absence of explicit timing definitions for Adapt Cases and avoids endless traces of system and environment changes without any adaptation, although possible. A detailed elaboration on the semantics is given in [1], [7].

Now that we have specified the behavioral part of self-adaptation (i.e. the adaptation rule itself), we describe how to specify the structural part of self-adaptation in the next section.

B. Structural Adapitivity Modeling: Adaptation View

The Adaptation View allows the specification of structural information of a system from an adaptation point of view. It
allows the designer to specify how exactly the environment of a self-adaptive system might change (and the system itself if underspecified). Among others, the Adaptation View enables the description of properties which are sensible and effectible by the self-adaptive system. Further, the Adaptation View allows the system designer to indicate, which elements are likely to change and which not, as well as the range of change. For instance, considering a rack-server system with a fan in its environment, the fan’s range of change might consider the fan’s speed ranging from 1 to 5. Among others, this information is used to analyze system and environmental changes. In the following, we briefly present the abstract syntax, concrete syntax, and semantics of the Adaptation View Model.

**Abstract Syntax.** To support the effective separation of concerns, we reuse the concepts of UML components, interfaces, and properties, as illustrated in Figure 4. In the UML, components are used to express concepts without the need to specify a concrete realization. Since we want to express a similar concept, we reuse and specialize the UML component in our language. As illustrated in Figure 4 we introduced two specialized component classes: system component and environment component. Where the prior represents the system itself, the latter represents an entity outside of the system’s borders. System and environment components can provide adaptation interfaces to enable the sensing and effecting of owned properties. Moreover, an environment component may asynchronously send environment signals to spontaneously trigger the system’s adaptation.

In the Adaptation View, knowledge (not shown in the meta model excerpt in Figure 4) can be considered as a kind of library which includes additional information that is relevant for the analysis and planning phase of the adaptation.

**Concrete Syntax.** The concrete syntax of Adaptation View diagrams follows mostly the standard UML notation for
Semantics. Apart from several additional static semantic constraints, the semantics of the Adaptation View match the ones defined in the UML for component diagrams. Additional semantics are defined for the so-called unbound variables, i.e., properties that do not have a specification. Basically, the semantics of these unbound variables is that they may adopt any value within the specified range with the given step size. This is especially crucial for analysis, where these unbound variables have to be simulated with every possible value.

So far, we have described the language that can be used to specify adaptive behavior separated from the core business logic. Therefore, we propose the use of an Adaptation View Model that decorates the standard system model using adaptation specific information such as sensors, effectors, and knowledge. Adapt Cases can then be used to specify monitoring and adaptation routines which refer to sensors, effectors, and knowledge. The complete language is formally defined, allowing for formal analysis and thus early feedback for the system designer as described in Section VI. In the next section, we describe how the ACML can be used in the development process.

V. DEVELOPMENT PROCESS FOR SELF-ADAPTIVE SOFTWARE SYSTEMS

Figure 6 shows how the ACML integrates in UML-based engineering processes. The figure focuses on the first two phases of a typical software development process: Requirements Specification and Analysis & Design. The standard development process shown at the bottom (using the UML for specification), includes the formulation of textual requirements in the beginning usually accompanied by a problem domain model which relates important concepts that occur in the respective domain. From these requirements, use cases are inferred which are refined by flows, e.g., in terms of activity diagrams. In the next step, the use cases are further refined with sequence charts describing the interaction with the system and finally result in an analysis class diagram which models and relates the concepts of the software system that is to be implemented. In the last shown phase, an architecture is designed (e.g., using component diagrams) and the lifecycle of important objects is modeled using state charts. Of course, different development processes may vary in terms of terminology and ordering of the described tasks. Especially the architecture design (by the use of components) is often performed earlier in the process.

For the separate specification of the system’s self-adaptation logic (see the top of Figure 6), we propose to use a concern-specific modeling language, called Adapt Case Modeling Language (ACML), that integrates with the UML [8]. The ACML consists of an Adapt Case Model that describes adaptation rules with an extended Activity Diagram, as well as an Adaptation View Model. The Adaptation View Model is a view on the designed software system particularly focusing on aspects that are important for adaptation (cf. Figure 7). Examples include the specification of sensors, effectors, and adaptation knowledge (e.g., aggregated values). The Adaptation View Model reuses the standard component specifications that are specified during business logic specification and attaches necessary sensors, effectors, value aggregations, etc. In fact, the Adaptation View Model is a decorating view onto the business logic component diagram. Since Adapt Cases operate on the Adaptation View only, the Adaptation View provides the interface between the business logic and the adaptation logic. This allows a clean separation of the two tasks of specifying the business logic and specifying the adaptation logic. For instance, in the development process, the business logic can be changed without considering the adaptation logic and vice versa.

The two process actions at the top of Figure 6 describe the usage of the ACML to describe self-adaptive systems. First, high-level Adapt Cases are used to model the adaptation requirements on a high level of abstraction. These high-level
Adapt Cases correspond to high-level use cases (or business use cases) and are usually given by a name and a natural text description, only. In UML-based development processes, use cases and Adapt Cases are used to provide a first (textual) operationalization of functional requirements, including adaptation requirements. High-level Adapt Cases are supported by a high-level Adaptation View Model that defines first sensors and effectors as well as environment components. Often at this level, the Adaptation View consists of only one single system component and zero to many environment components.

Figure 7 shows a high-level ACML model that reflects the scenario from Section II. The figure shows the high-level Adaptation View Model with two high-level Adapt Cases. The model has been created during late requirements engineering and will be refined later during the design phase. The model defines a system that contains local servers and a cloud provider that offers cloud servers. Further, the model defines four adaptation interfaces, namely two sensors for the system’s temperature and its load and two effectors for activating and deactivating local and cloud servers. Finally, the model shows which Adapt Case uses which adaptation interface (sensors and effectors) for monitoring and adapting the system and its environment. On this abstraction level, the Adapt Cases are described using natural language text as known from use cases.

Next, these high-level models are refined to low-level Adapt Cases with concrete monitoring and adaptation routines which relate to the refined Adaptation View Model (cf. Figure 6 top right). The monitoring and adaptation routines are specified using specific UML activities as shown in Figure 3 and Figure 9 and described in Section IV-A. The Adaptation View Model is refined with detailed interface descriptions and knowledge as shown in Figure 5 and described in Section IV-B.

Since both the Adapt Case Model and the Adaptation View Model are heavily UML based, and moreover, are strongly aligned with UML modeling practices, the ACML can be used with any UML based software development process, and in particular, standard well-known UML refinement approaches may be used (cf. [9] for use case techniques).

Besides the constructive methods such as concern-specific languages, analytical methods help engineering good software systems. In the next section, we describe how we apply quality assurance techniques to models that have been constructed using the ACML early in the engineering process.

VI. Quality Assurance for Self-Adaptive Systems using the ACML

The basic idea of the quality assurance task is depicted in Figure 8. Adapt Cases and Adaptation View are formally defined and taken as input for a model checker. Same applies to quality properties which are formally defined using temporal logic.

The model checker’s result is used to create a quality analysis report that is provided to the system designer—already during design-time. Examples for quality constraints include absence of deadlocks and stability of the adaptation rule set: A set of rules is free of deadlocks if no adaptation rule puts the system in a state where system progress (e.g., adaptation) is required but no other rules can apply. A set of rules is stable if no two or more rules continuously redo their adaptation actions without any change in the environment or the system.

To exemplify how to check stability in a modeled self-adaptive system, we consider our example scenario from Section II. In particular, we consider the Adaptation View model given in Figure 5 and two Adapt Cases the first of which activates servers if the system’s load increases (cf. Figure 3) and the second of which deactivates servers if the temperature exceeds a given level (cf. Figure 9).

As stated in Section II, the given system model gives rise to the following, problematic situation: When the load (for a single server) increases, the overall computed system load increases. Therefore, the Adapt Case from Figure 3
applies and activates local servers. This in turn leads to an increasing system temperature making the Adapt Case from Figure 9 apply which deactivates local servers. Deactivating local servers ends up in an increasing system load again and therefore, the adaptation cycle starts over. This situation is depicted in Figure 10.

Apparently, this is not the desired behavior, since the system constantly switches servers on and off without solving the actual problem of too high system load or temperature. The problem is that the Adapt Cases indeed react to changes of the system and its environment, but these changes are indirectly caused by other Adapt Cases. The result is a system which ends up in an endless loop of adaptations without improving the system health. In the following, we will show how to automatically detect problems such as described above, and how to support the system designer in fixing these problems.

Fig. 10. Sketched Excerpt from the System’s State Space

Therefore, we use DMM Rules to formalize the semantics for the ACML. DMM is a graph-transformation language with the purpose to define execution semantics for meta-model based languages [2]. The semantics of the Adaptation View describes how the system and its environment change over time. For instance, in our scenario, the semantics simulate the change of the system load or the arrival of environment signals. The semantics for Adapt Cases describes the reaction of Adapt Cases to a change in the Adaptation View and the effect of applying adaptation actions. With the specified DMM rules, we automatically create a labeled transition system (LTS) which reflects the state space of our modeled system.

States in our transition system describe possible instances of our modeled system. Transitions are applications of DMM rules, which either change the state of the system and its environment (e.g., create signals or change the system load in our scenario), or apply Adapt Cases which in turn adapt the system via AdaptationInterfaces. Note that the modeler does not have to create any DMM rule by herself. Instead, these DMM rules are manually created once by the language designer to reflect the languages semantics and are used later to interpret the models created within this language by the modeler. The interpretation’s result is the LTS. For our scenario, the generated LTS would contain the trace that is shown in Figure 10. Now, we can formulate and check temporal logic formulas which range over the transition labels using the model checker Groove [10].

Let LTS be the LTS computed from the set of DMM rules applied to our scenario model. The latter consists of the Adapt Cases (figures 3 and 9) and the adaptation view model (Figure 5). Further, let \( \mathcal{A} \) be the set of DMM rules that evaluate Adapt Cases and \( \mathcal{C} \) the set of DMM rules that simulate system and environment changes.

**Definition 1: (Rule Set Stability)** An adaptation rule set \( \mathcal{A} \) is stable if the LTS does not contain paths such that rules out of \( \mathcal{A} \) are applied infinitely often, but no rule \( c \in \mathcal{C} \) is applied in between.

Before we can finally provide the LTL formula which will verify our system for stability, we need to briefly investigate how the Groove model checker works. Groove gives rise to labeled transition systems (LTS) where states are typed graphs and transitions are applications of graph transformation rules; the transitions of the LTSs are labeled with the according applied rule’s name. As a result, the Groove model checker can process LTL formulas where the atoms are rule names. For instance, if the LTS contains a state \( s \) with an outgoing transition labeled \( l \), the property \( l \) is true for \( s \). One consequence of this is that we can verify whether a rule out of a set of given rules is applied by creating the disjunction over the rules’ names. We make use of this by defining the property \( A := \forall a \in \mathcal{A} \). Given a state \( s \), \( A \) will be true iff at least one of the adaptation rules can be applied to \( s \).

We now turn to the LTL property used to verify rule set stability:

\[
\neg \diamond \Box A
\]

Here, the formula is true if there is no trace in the LTS such that from one point in time on, only adaptation rules out of \( \mathcal{A} \) are applied, and therefore it realizes the requirements of Def. 1.

Using this formula (and similar ones) we can support the designer with direct feedback on the quality of the model at hand. If the LTS fails to fulfill one of our properties, the model checker will provide a counter example (i.e., a trace of states showing which sequence of rule applications led to a state violating the LTL property). In an integrated modeling environment for self-adaptive systems, we use that information to help the designer to understand why the system model.
violates a quality property, and how the system model can be fixed. For this, we use the DMM simulation capabilities to simulate the counter example and thereby giving the user an intuition of the problem on the same abstraction level that he modeled the system at. In our scenario, the problem is that the server management Adapt Case activates local servers on high system load without considering the overall system temperature. The problem could be approached by inserting an additional condition to the Adapt Case, such that it does not apply if the system temperature is already high.

The State Space Explosion Problem. A problem that usually occurs when applying model checking, is state space explosion. The reason for an exploding state space is the multiplication of possible system states. To address the huge number of system states, we use a multi-staged model checking approach. At each stage a number of model-checking runs are performed where earlier stages use more abstract models resulting in smaller state spaces. Therefore, we intelligently adjusted the semantics and models to reduce the number of possible states in first validation iterations. For instance, in a first model checking run, we abstract from parallel executed adaptations. An error found in this abstracted setting will occur in the non-abstracted setting as well and thus the validation is stopped in the first run providing fast feedback to the user. In this manner, we build the multi-staged model-checking approach for ACML models that takes the specific semantics of self-adaptive systems into account to reduce state spaces. Details about how we address the state space explosion problem within our approach have been published in [7].

VII. THE ENGINEERING APPROACH – DESIGNER’S TASKS

The goal of our modeling and quality assurance approach is to support designers in engineering high quality self-adaptive systems without having the knowledge of a meta-model or model checking expert. Therefore, our approach hides the quality assurance complexity from the user and provides a user-friendly domain-specific language which is based on the well-known UML. Thereby, we focus on the early design time of a software engineering process.

From an end-user perspective, the designer

• models the system’s business logic using the UML,
• creates an adaptation-specific view onto the system using the Adaptation View Model, and
• identifies and models adaptivity using Adapt Cases.

The modeling workbench automatically performs hidden quality checks and provides direct feedback to the user in form of natural language error messages (e.g., the execution of Adapt Case X prevents Adapt Case Z to ever apply again).

Since our workbench is eclipse-based, these validation results are given in the familiar eclipse validation dialog and error view. Of course, for expert users, a detailed view with the erroneous system state and the corresponding trace to that state exists.

In our approach, the designer

• does not need to specify formal DMM rules,
• does not need to understand model checking, but
• only needs to model in a UML-like (thus familiar) language.

We have implemented a fully functional Adapt Case and Adaptation View editor as well as the quality assurance approach using DMM and the Groove model checker.

VIII. EVALUATIONS

We have conducted several evaluations for the ACML. For the first evaluation [11], several quality criteria were identified from different sources in current research (e.g., Andersson et al. [3]). Using these criteria, particular language features were evaluated using the rack server system example. This approach lead to several findings which were used to further improve the ACML. For instance, it was identified that the ACML lacks suitable language elements to reflect the delayed effect of an adaptation especially in mechatronic systems. Therefore, we introduced a language feature that makes Adapt Cases wait for the adaptation effect before a possible reapplication.

The second evaluation [11] involved a group of 10 students who used the ACML to model the adaptivity of a real-world web-based learning system. The goal of the system’s adaptation is to adapt the tasks’ difficulty to the user’s skill level. Again, the language proved to be sufficient in modeling such systems. This evaluation’s main result was that the ACML’s operations seemed to be too low-level. Therefore, we introduced customizable compound operations that reflect typical adaptation pattern.

A further evaluation was performed by a project group, a group of 10 students who together conducted a project that lasted a full year in cooperation with two industrial partners. The task of the project group MePaso [12] was to create a modeling and quality assurance workbench for process-based service compositions. Therefore, the students reused Adapt Cases to model the adaptation of the process itself and the service bindings. In this context, the ACML was extended to specify the adaptation of processes, i.e., the behavioral description of software systems. The implemented ACML workbench was evaluated using a realistic software project that was given by one of the industrial project partners and originates from the domain of car insurance. The project was particularly interesting because of its large size and therefore the findings related to the model complexity and the state space explosion problem. The two industrial partners involved in the project group plan to utilize the results generated by the project group. Further evaluations regarding the state space explosion problem and a solution particularly for the ACML have been presented in [7].

Finally, a larger case study has been presented at the Comparing Modeling Approaches Workshop 2012 (CMA’12) [6]. All evaluation approaches prove the applicability of the ACML and the corresponding tooling. The evaluation results proving language weaknesses were taken to further improve the ACML.
IX. RELATED WORK

Recent research provides various approaches that support the engineering of adaptive systems in different phases of the design process. Cheng et al. present a combined, goal-based modeling approach in [13] using the KAOS methodology that explicitly considers uncertainty and the context the system is acting in. Identified uncertainty is modeled using the RELAX notation. The system context is defined by a domain model and referenced in the goal-model. This approach takes care of adaptivity in the requirements phase and thus is input for Adapt Cases in the analysis & design phase.

Use cases [14] are a well-known modeling technique for the logical system design. Using "traditional" use cases, the described adaptivity of the system is hidden within the attached activity or sequence diagrams. As a result, adaptation is not sufficiently taken into account during the logical design. Adapt Cases extend use case modeling by making the explicit specification of adaptation actions available.

Extended use case approaches for software product lines [15] provide the specification of variations which can be related to simple condition concepts. The description of adaptation actions are much more complex than the modeling of variations since additional aspects like monitored elements in the system and the environment need to be considered and are not supported by these approaches. Adapt Cases support the detailed modeling of the system and the environment. Thereby, complex monitoring and adaptation activities considering changes in the system and its environment can be defined.

Existing architectural approaches for adaptive systems like MAPE-K [5] and the three-layer reference architecture proposed in [16] describe abstract logical concepts for realizing adaptivity. These approaches specify aspects and characteristics of adaptive behavior on a high level of abstraction and cannot be used for specifying concrete adaptation actions. Nonetheless, these approaches lay down the foundations for the specification of adaptivity in Adapt Cases.

Fleury and Solberg are presenting a domain-specific modeling language in [17] which separates adaptation and system logic. Their decoupling to the context is made through context variables. Dependencies of these context variables can be described using first order logic expression. This approach is not integrated into the UML. Further, they use constraints, which can be checked against the system model. Their constraints may be considered as application-specific properties. However, they have no support for global properties (e.g. rule set stability), which would check the quality of the adaptation rules themselves.

Hebie et al. [18] propose a UML profile for explicitly model control loops of adaptive system and identify dependencies between those. This approach extends the UML but focused on one particular aspect being the dependencies between component interfaces considering the adaptation. Adapt Cases are much more detailed and cover the complete system design.

Adler et al. [19] and Schneider et al. [20] propose new modeling concepts and their formalization for self-adaptive systems. Thereby, they support the verification of modeled self-adaptive systems using model checking. However, their languages are not based on standard languages such as the UML making their use and integration into software development processes rather burdensome. Another approach [21] supports modeling and verification of self-configuring behavior, however, is limited to a small application domain, i.e. robotic plants. These and similar approaches [22] are focused on the area of embedded systems and therefore do not well integrate with existing methods for software engineering in general.

Several approaches exist that deal with the formalization and analysis of adaptation and self-adaptive systems, respectively [23]. Recent research includes FORMS [24], a formal reference model for self-adaptation. While FORMS is fully capable of precisely modeling self-adaptive systems on a formal basis, it does not define any notion of execution semantics, that is how the adaptation of the system takes place. Therefore, it does not allow the analysis of the particular adaptation rules and their properties.

Another approach uses CSP to model and analyze self-adaptive systems [25]. However, this approach lacks a user-friendly modeling language for self-adaptive systems and in turn requires the system designer to have deep knowledge in formal methods.

Zhang and Cheng [26] extend LTL with an adaptive operator that allows the description of adaptive behavior using time logic. Programs are state machines, the program specification are LTL formulas. The authors describe three types of adaptation semantics using their adaptive operator. They show that safety and lifeness properties are preserved under the adaptive operator. However, the context is not considered and the description via LTL seems rather unhandy to be used in practice. Also, Their approach does not support the quality assurance of the adaptation rules themselves but only supports the description of adaptation behavior and its compliance with the modeled system.

Finally, comparing our approach to standard control theory modeling and reasoning approaches such as MRAC [27] or MIAC [28], the distinctive features are an integration into standard software engineering methods using the UML, and the full transparency of formal model checking techniques for the modeler. However, the adoption of control theory techniques into the software engineering of self-adaptive systems should be investigated further (cf. [29]).

X. CONCLUSIONS & FUTURE WORK

Conclusions. In this paper, we propose an integrated modeling and quality assurance approach for self-adaptive systems. By the use of our concern specific modeling language—the Adapt Case Modeling Language (ACML)—which fosters the principle separation of concerns, we hide complexity from the designer and allow the concentration on the adaptive behavior in early design. Based on the well-known standard UML, our
language is close to methods usually used in software engineering, making the adoption of our approach into software engineering processes easy. Capturing the semantics of self-adaptive systems on this high level of abstraction enables us to provide the designer with precise and immediate feedback on certain quality properties of his models, such as adaptation rule set stability. Hence, applying our approach results in high quality models for self-adaptive system that are not only intuitive, explicit, and well-structured, but are also guaranteed to fulfill certain important quality properties, such as deadlock freedom, stability, fairness, etc. The creation of high-quality models usually leads to software products of higher quality. Hence, our approach helps engineering self-adaptive systems of high quality.

Future Work. We are currently working in several directions. First, to achieve further methodological support, we are investigating common adaptation pattern that may lead to an extension of the ACML. Second, we are investigating different quality properties and their relevance for self-adaptive systems. Furthermore, we are working on a holistic approach towards simulation and execution of the modeled self-adaptive systems. Another open issue is the inclusion of timing. In future, we want to enable the explicit specification of adaptation durations in the models. Finally, we are investigating the impact of Adapt Cases on other requirements and use cases with a special focus on non-functional performance requirements.

REFERENCES