UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Adaptivity Engineering

Modeling and Quality Assurance for Self-Adaptive Software Systems

Markus Luckey, M.Sc.

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering, and Mathematics
of the University of Paderborn
in partial fulfillment of the requirements
for the degree of Dr. rer. nat.

September 2013

To Stefanie,
and my children Thilo & Milian.

Acknowledgments

Writing this thesis would never have been possible without being accompanied and supported by many people. Hence, I would like to thank the following people for sharing their experience, time, and patience.

I deeply appreciate my advisor and mentor, Prof. Dr. Gregor Engels, for his efforts to drive me to success with candid advice, patient guidance, and kind encouragement. Gregor, thanks for your trust and all the opportunities to grow into many different directions and the fruitful ground to take these opportunities. Talking about opportunities, I further thank Dr. Stefan Sauer who allowed me with a great portion of confidence to take part at the s-lab – Software Quality Lab that he is managing. I would also like to thank my committee members Prof. Dr. Danny Weyns, Prof. Dr. Uwe Kastens, Jun. Prof. Dr. Steffen Becker, and Dr. Theodor Lettmann for the friendly guidance and thought-provoking suggestions while carefully reviewing my work to strengthen and elevate its quality.

I would like to thank my fellow doctoral students and co-workers and especially Jan-Christopher Bals, Matthias Becker, Dr. Fabian Christ, Masud Fazal-Baqaie, Barış Güldalı, Benjamin Nagel, Yavuz Sancar, Hendrik Schreiber, and Henning Wachsmuth as well as Friedhelm Wegener and Beatrix Wiechers for their support, feedback, and friendship.

I am especially grateful to two of my friends and colleagues, Dr. Christian Gerth and Dr. Christian Soltenborn. Chris, many thanks for sharing and discussion so many scientific, professional, and personal topics that influenced my decisions the last 4 ½ years and for always taking the opposite part in our discussions. In a similar vein, thank you, Solti. Not only that you helped me with my scientific results by laying the grounds for my thesis with your work on Dynamic Meta Modeling and your steadily willingness to proceed your approach to support mine. Thanks also for the numerous talks where you helped me taking another point of view!

Finally, the ones who closest accompanied the venture of my thesis are part of my family. I want to thank my parents for being my greatest supporters, for loving, supporting, understanding me, and for giving me a chance to thrive. Also, many thanks to my parents-in-law for the warm welcome and support. Finally, I am eternally grateful to my loved partner Stefanie and my children Thilo and Milian who allowed me with a great deal of understanding, patience for my different moods, and the right degree of distraction to finish my thesis. Thanks for reminding me of the most important things in my life!

Adaptivity Engineering
Modeling and Quality Assurance for Self-Adaptive Software Systems

Abstract

Modern software engineering introduces self-adaptivity features to perform automatic maintenance and make software systems more flexible and resilient. Unfortunately, introducing the additional self-adaptivity features makes software design bloated and complicated. As a consequence, software design models are often prone to errors. The literature proposes constructive approaches such as MDE, patterns, etc. as well as analytical approaches such as testing or model checking to solve the problem of complexity in general. However, there is no sufficient adaptivity-specific support throughout the engineering process, i.e. no approaches that support the creation of self-adaptivity specification models and their quality assurance.

In this thesis, we will propose an integrated modeling and quality assurance environment for designing self-adaptive software systems. Therefore, we will propose constructive methods (e.g., languages) and analytical methods (e.g., model-checking) to support the engineering of these systems. Both types of methods are integrated into standard software engineering techniques and tools. As a result, the designer is supported in modeling self-adaptive software systems using concern-specific languages and receives immediate feedback about the quality of his models. This way, software engineering for self-adaptive systems is getting supported starting at the early design phase leading to less errors produced, and thus, to better software, overall.

Adaptivity Engineering
Modeling and Quality Assurance for Self-Adaptive Software Systems

Zusammenfassung

Moderne Softwareentwicklung nutzt Techniken der Selbstadaptation, um Wartung von Softwaresystemen zu automatisieren und diese somit flexibler und robuster zu gestalten. Allerdings führt die Einführung solcher Techniken zu größeren und komplizierten Softwareentwürfen. Die Konsequenz sind Fehler im Entwurf. In der Literatur werden konstruktive Methoden wie MDE oder Patterns und analytische Methoden wie Testen oder Model Checking vorgeschlagen, um das Komplexitätsproblem zu verringern. Allerdings werden die Techniken der Selbstadaption von solchen Methoden bisher noch wenig unterstützt, d.h. dass es wenige integrierte Ansatze für die explizite Modellierung und Qualitätssicherung von Selbstadaptation gibt.

In dieser Arbeit schlagen wir einen integrierten Modellierungs- und Qualitätssicherungsansatz für den Entwurf selbstadaptiver Softwaresysteme vor. Es werden sowohl konstruktive Methoden (z.B. Sprachen) als auch analytische Methoden (z.B. Model Checking) für die Unterstützung der Entwicklung solcher Systeme vorgeschlagen. Beide Typen von Methoden sind in Standardtechniken und Werkzeuge integriert. Im Ergebnis wird der Entwickler in der Modellierung selbstadaptiver Softwaresysteme durch den Einsatz von adaptionsspezifischen Sprachen unterstützt. Durch die dazu passenden Qualitätssicherungsverfahren erhält der Entwickler unmittelbare Rückmeldung über die Qualität seiner Modelle. Somit wird die Entwicklung selbstadaptiver Systeme bereits in frühen Phasen des Entwicklungsprozesses unterstützt, Entwurfsfehler werden vermieden und somit bessere Software gebaut.

Contents

1

Introduction

"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."

– Charles Darwin

1

This thesis proposes constructive and analytical methods to support the engineering of self-adaptive software systems during high-level design. Both types of methods integrate into standard software engineering techniques and tools. As a result, the designer is supported in modeling self-adaptive software systems using concern-specific languages and receives immediate feedback about the quality of his models. This way, Software Engineering for self-adaptive software systems is supported during the high-level design phase leading to less error-prone software specifications, and thus, to better software, overall.

This chapter is organized as follows: In Section 1.1 the setting and background of this thesis is provided, Section 1.2 motivates the problems addressed in this thesis, Section 1.3 gives a brief description of the proposed solution, Section 1.4 presents the used research approach, Section 1.5 defines the scope of this thesis, and finally, Section 1.7 provides an overview of the chapters in this thesis.

1.1 BACKGROUND

SOFTWARE IS EVERYWHERE

Today, in our culture it is no more possible to imagine a life without technology and computational power in particular. It is inherent in our professional environment as well as in goods of every-day use, e.g. washing machines, stove, car, etc. Most computational power is controlled by software. Software provides more and more new convenient functionalities which have long since become

part of our daily lives. Examples include the smart phones, one of which almost everyone owns. To the same degree, the complexity of software increases making its development, deployment, and maintenance difficult. Especially, the high demands on the software's quality (e.g. dependability, safety, fault tolerance, ...) lead to high complexity. These high demands require making trade-offs with current trends of software engineering, including the distribution of systems, genericity of software for the sake of flexibility, and adaptivity to encounter changing environments.

INCREASING COMPLEXITY

Distribution leads to more and more business functionality being outsourced to external service providers. Thus, neither the code is under control, nor is the own service used by only oneself but provided to the outer world. Software is therefore more and more relying on black box components. However, black box components exhibit unexpected behavior and usually cannot be influenced in terms of error correction. Further, due to the increasing size and complexity of software systems, the complexity of maintaining theses systems rises, while errors and failures may have severe financial impacts as evident in the news. While software tends to be more and more generic to be operated in various scenarios (and thus saving money for production; higher ROI), the number of configurations explodes. Finally, without system adaptivity, having a changing context (e.g., changing needs or a changing environment) that demands for a different configuration leads to a system halt, reconfiguration, and system restart. However, the desired operation mode is continuous operation, that is, the system shall be configured at runtime (i.e. adapted), either manually or automatically.

DISTRIBUTION

GENERICITY

ADAPTIVITY

Hence, the ability of coping with distributed components, while being generic and adaptive is key for today's software. Software that does not exhibit these properties suffers the so-called lack of movement [Par94]. Lorge Parnas defines the lack of movement to be the "aging caused by the failure of the products' owners to modify it to meet changing needs".

LACK OF MOVEMENT

To encounter the lack of movement, often the wish of more flexible software entails that adapts itself to a changing context which even might be partially unknown. For instance, mobile applications shall flexibly adapt to different environmental situations at different locations and software applications shall automatically adapt to different customer landscapes or different customer business workloads (e. g. by self-configuration). Software like this that is capable of adapting itself to its changing context is recently known as self-adaptive software [WIMA12].

SELF-ADAPTATION NEEDED

Self-adaptive systems have been studied in several non-computer science dis-

INTERDISCIPLINARY NOTION OF SELF-ADAPTIVITY

ciplines such as biology, economy, and sociology [ST09]. On an abstract level, all disciplines share a common understanding of self-adaptive systems: A system that is able to respond to context changes in order to increase its quality of performance (reproduction, stability, or the like). For several years, computer science has adopted the notion of adaptivity to describe software systems that autonomously adapt themselves to changes in their context. Within computer science several fields have adopted the notion of self-adaptivity, being software engineering, artificial intelligence, control theory, and network and distributed computing [ST09].

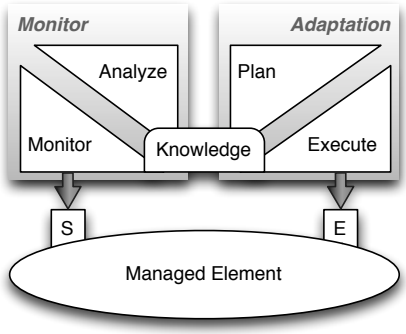


FIGURE 1.1. Annotated IBM MAPE-K Reference Model

CONTROL LOOPS AS FIRST CLASS CITIZENS

MAPE-K FEEDBACK LOOP

The discipline of software engineering identified control loops to be an integral part of self-adaptive software systems [BMSG+09]. A particular reference model proposed by the IBM that describes these control loops in more detail is called the MAPE-K model [KC03]. It divides a control loop into four phases: monitor, analyze, plan, and execute. As shown in Figure 1.1 the MAPE-K model considers a managed element that is adapted eventually. This managed element is monitored through sensors. The resulting data is used for the analysis and if necessary for the plan of adaptation. Finally, the adaptation is executed through effectors. All phases read and store knowledge into a so-called knowledge model. While the MAPE-K model is well-known and often used in the community, several approaches combine the phases *monitor* and *analysis* as well as *plan* and *execute* into single phases named *monitor* and *adaptation* [WIMA12].

In the past decade few software systems were considered to be self-adaptive, mostly in the domain of telecommunication where downtimes were not an option and contemporary human intervention was not always possible [CGS05]. Today, self-adaptation requirements are also explicitly set for software-intensive systems in the mobile embedded system domain and in business information systems.

MOTIVATION 1.2

While self-adaptation capabilities to a certain degree always existed in software systems, their increasing necessity poses new challenges to software engineering. In particular, introducing self-adaptivity capabilities to the software system results in additional complexity during design time making it hard to show correctness of these software systems. This is mainly because a variety of adaptation rules that change the system at runtime make the system behave somewhat unpredictable. High complexity in turn often leads to errors which must be corrected at high expense.

INCREASING COMPLEXITY

As an example, let us consider the scenario of a crisis management system named bCMS.[CCG+12] The main purpose of the system is to support a fire station coordinator and a police station coordinator in exchanging information related to a particular crisis and managing crisis-related tasks such as dispatching vehicles and planning routes. The bCMS has several requirements regarding resilience, e.g., concerning the communication availability. If communication is interrupted, the bCMS system is paused until communication is established again or the system turns into a failsafe mode such that both coordinators may proceed crisis management independent from each other. This functionality can be considered as self-adaptive behavior while the communication availability is part of the uncertain context that has to be monitored.

CRISIS MANAGEMENT SYSTEM SCENARIO

During the first operationalization of the requirements, e.g. using 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 system where the communication availability is considered to be fixed, the specification of the self-adaptive bCMS 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 pauses and continues the process is insufficiently specified, the continuous pausing and continuing of the processes for different reasons might lead into an unstable system state.

SEVERE SPECIFICATION FLAWS

Modern software engineering addresses this problem using model-based design (e. g. MDA [Obj03]) and specific modeling languages, especially concern-specific modeling languages (CSML) that focus on a particular concern (e.g. self-adaptivity) or domain-specific modeling languages (DSML) that focus on one particular application domain such as rack server systems. CSMLs and DSMLs help bridging the semantic gap between the intuition of the problem

MDA, CSML, AND DSML

and programming languages like Java since they allow describing the problem on a high level of abstraction and even allow the designer to use concern-specific terms while still being formal enough to support automated analysis. That is, model-based design is a solution to high complexity and based on formal models it helps ensuring correctness.

SEPARATION OF CONCERNS (SOC)

Model-based design, in addition, eases using the paradigm of separation of concerns (SoC) which can be applied throughout the whole software development process. It allows "focussing one's attention upon some aspect" [Dij82]. Further, SoC allows the designer to apply concern-specific methods and languages to construct and analyze concern-specific specifications.

USE OF SOFTWARE ENGINEERING METHODS

Another—rather basic—approach applied to master complexity during software engineering is the use of software engineering methods (SEM). Software engineering methods prescribe the use of specific concepts and languages (i.e. specification artifacts), processes, as well as tools, and thus, make expert knowledge available for use in a relatively intuitive way by still preserving the genericity to be applied to various different software engineering projects.

NO CONTINUOUS METHOD FOR SELF-ADAPTIVITY

Unfortunately, there is hardly any model-based software engineering method available that continuously separates the concern of self-adaptivity. On requirements level, several approaches suggest the use of goal-modeling to describe self-adaptation requirements [MPP08, CSBW09, GSB+08]. Usually, goal-modeling approaches integrate well into standard software engineering methods that usually make use of general purpose modeling languages such as the UML or BPMN. On low design level (i.e., platform-specific, rather technical design), there are other approaches existing that allow the explicit and separate specification of self-adaptivity (see e.g. [HGB10, GCH+04, AST09]). However, during high-level design (i.e., platform-independent, logical design), self-adaptivity is either neglected or tightly interwoven with core application logic (cf. Figure 1.2).

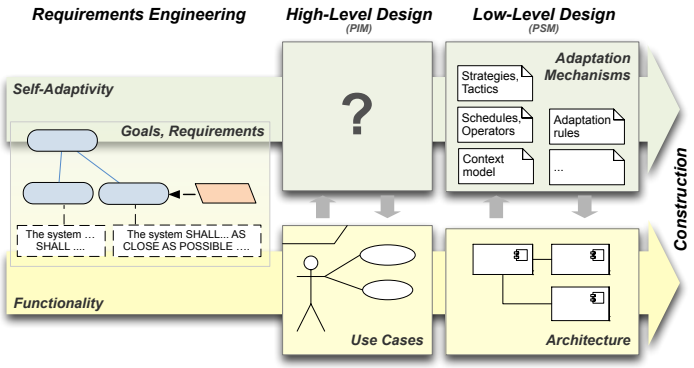


FIGURE 1.2. Software Development Process – Separation of Concerns

Figure 1.2 shows a standard software development process[1]. The lower part of the figure (yellow arrow) sketches the traditional software development process. During the requirements phase of those processes, high-level goals and requirements are identified. These requirements are refined into high-level platform-independent use cases (for PIM cf. MDA [Obj03]). In the subsequent phase the platform-specific model is created—typically the concrete software architecture. High-level use cases are either mapped to components in the platform specific model (PSM) and/or are refined to more low-level (technical) use cases [Coc00]. Finally, the low-level design model is implemented in the construction phase using frameworks, libraries, etc.

Adaptivity in high-level design using use cases

While specifying a software system during the high-level design phase, the core functional concerns are modeled using e.g. use cases. In fact, UML use cases and the attached activity diagrams offer different generic means to also model self-adaptivity, such as decision nodes or exceptions. However, these means fail in maintaining the separation of concerns principle, i.e. self-adaptivity and functional concerns are mixed in the model. See Figure 1.3 for an illustration. The use case combines both application and adaptation logic since the attached activity diagram contains actions that execute application logic (action A) and those that adapt the system (action B) using dedicated adaptation interfaces (green). The structural system model describes com-

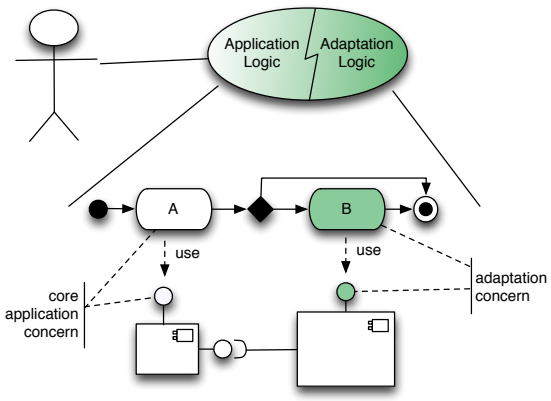


Figure 1.3. Mixed Concerns in SE Models

ponents and interfaces that constitute application logic (white) and interfaces whose main purpose is to support adaptation (green interface). Because of the mixture of different concerns, designers might miss important self-adaptivity patterns, or even induce design flaws due to the increasing complexity. Furthermore, model-based analysis of self-adaptivity is aggravated since extraction of the corresponding concerns from the mixed models is burdensome or even impossible.

Mixture of concerns in UML use cases

[1] Names of phases vary in the different process models.

In this thesis, we present a model-driven engineering approach that supports the high-level design of self-adaptive software systems and thus fills the gap between requirements engineering and low-level design (see Figure 1.2). In the following, we briefly discuss some requirements that a model-driven engineering approach for self-adaptive software systems must fulfill.

REQUIREMENTS FOR MODEL-DRIVEN ENGINEERING OF SELF-ADAPTIVE SOFTWARE SYSTEMS

Separation of Concerns To allow focussing on the concern of self-adaptivity within a software system, this concern should be separated from the core business logic concerns. That is, an approach should provide means to separately specify self-adaptivity and abstracting from the rest of the system's logic, whereas ideally, the separated concern is still related to the rest of the system's logic allowing to always provide an overall model of the system's complete logic.

Analyzability An important benefit of separating concerns is the ability to analyze a concern in depth. For this, however, methods and techniques must actually support the concern-specific analysis. In particular, when building highly complex software systems that change their behavior at run time, it is vitally important to provide *hard guarantees* regarding the system's safety and liveness.

Integration To best leverage a method that supports the specification and analysis of the self-adaptation concern, this method should be integrated into existing software engineering methods. That is, ideally, if a system designer decides to model the concern of self-adaptation explicitly and separately, she may use the method in addition to her existing methods without any considerate alignment efforts. Further, the approach should cover the complete engineering process, for instance in combination with existing other approaches.

Intuitiveness To increase acceptance, the approach should be intuitive to use, especially since early in the engineering process, non-experts might be required to model and/or understand the self-adaptivity specification. To support the intuitiveness, well-known techniques and paradigms should be reused. Further, at this stage in the engineering process, the approach should allow the specification of self-adaptivity close to the intuition of the system's behaviors.

Genericity As known from general purpose languages such as the UML, the approach shall be applicable in a variety of domains. Further, it should support the integration with other specification approaches as known from the UML profiling mechanism. The generic approach should support the specification of self-adaptivity on different levels of abstraction and using different degrees of detail.

In the following, we will give an overview of the solution that is presented in this thesis.

Solution Overview and Research Contribution 1.3

This thesis proposes a model-driven engineering approach to address the requirements posed above for the high-level design phase. The model-driven engineering approach for self-adaptive software systems provides two ways to assure high quality in spite of complexity: constructive and analytical methods.

Constructive methods aim at improving the quality during creation of specification documents and code. As is widely accepted, the earlier high quality is assured the better [Dav93] since the later errors in the system design are recognized the higher the resulting costs. A common way to decrease the complexity during design time is utilizing the principle of separation of concerns, e.g. during modeling. Examples are evident in different disciplines such as performance or security engineering. In the case of self-adaptive systems, the adaptation concern should be separated from the business logic concern. Therefore, separate models are needed, one for the business logic and another for the adaptation logic (see Figure 1.4). Using concern-specific modeling languages

Separation of concerns

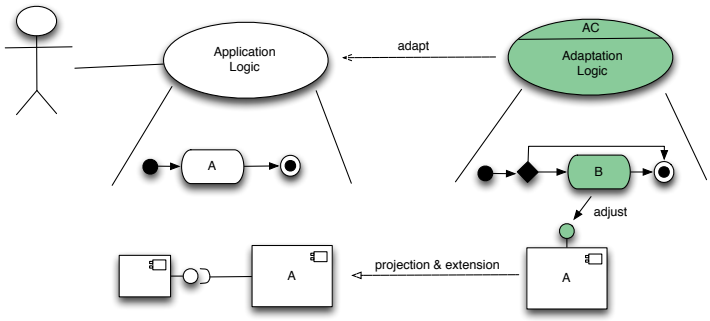


Figure 1.4. Separated Concerns in comparison to Figure 1.3

CONCERN-SPECIFIC MODELING LANGUAGE FOR SELF-ADAPTIVE SOFTWARE SYSTEMS

(CSML) for that purpose eases the creation of models for the respective concern and therefore usually improves the quality of models. Having high-quality models leads to higher quality in the resulting software product since design decisions can be validated early in the development process. Thus, in this thesis, we propose a dedicated method for specifying self-adaptive software systems including a modeling language, the **Adapt Case Modeling Language** (ACML) that allows the designer to specify adaptation in a rule-based manner (cf. event-condition-action rules). An adaptation rule consists of a monitoring definition that observes particular events and checks for certain conditions, and an adaptation definition that performs the corresponding adaptation actions. The proposed language is based on the UML [Obj10b] where UML activities are used for describing the adaptation rules (i.e., the monitoring and adaptation definitions). Based on the UML, the ACML is easy to understand since its visual syntax is widely known. Thereby, the ACML bridges the semantic gap between requirements and low-level design phase, allowing the easy translation of the intuition of a system into a formal language.

EARLY QUALITY ASSURANCE

Analytical methods are the second way of assuring high quality in spite of high complexity during software design. Again, the earlier the analytical methods are applied, the better. That is, we aim at supporting analytical quality assurance techniques already during early high-level design that detect specification flaws, e.g. by the specification of two conflicting adaptation rules (see Figure 1.5). Several static and dynamic analytical methods exist,

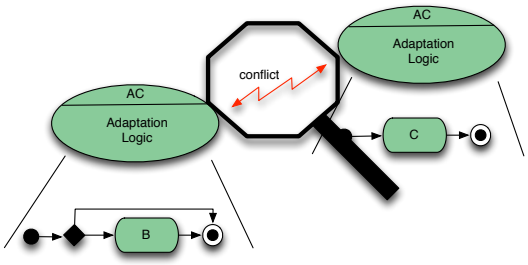


FIGURE 1.5. Quality Assurance during early System Design

including the detection of anti-patterns, testing, model checking, and simulation. Since testing requires an implementation of the system to be present, it cannot be applied in early system design phases. In contrast, model checking allows the early quality assurance of the system to be built since it relies on the design models. Furthermore, while dynamic analysis like testing allows to test single execution paths through the system, model checking enables checking the complete state space of the modeled system. Since model checking covers every possible (and modeled) system state, simulation may be considered as a subclass of model checking. Thus, we aim at supporting early model checking of the modeled self-adaptive software system, i. e. assuring high quality of the

modeled adaptation rules. Using model checking, safety and liveness properties may be checked. Common safety and liveness properties that are important for self-adaptive software system include stability and deadlock-freedom. However, model checking usually requires deep knowledge in formal methods which software designers usually do not have. Therefore, we propose an **integrated quality assurance approach** that is based on model checking but hides its complexity from the user.

model checking for Self-Adaptive Software Systems

All in all, in this thesis, we propose an **integrated rule-based modeling and analysis approach for self-adaptive software systems**. With the ACML, we propose a UML extension that allows the specification of structural and behavioral aspects of self-adaptive software systems. Using this extension, we support the object-oriented modeling of self-adaptive software systems using the de facto standard modeling language UML on the one hand, and concern-specific concepts on the other hand. Utilizing a semantics specification language for meta-model based languages, named DMM [Hau05], we define formal semantics for our modeling language, the ACML. Based on the formal semantics, we enable the quality assurance of the modeled self-adaptive software system early during design time using a modeling workbench that provides immediate feedback to the modeler. That is, we present a formal framework for proving safety and liveness properties on modeled self-adaptive software system while still providing the system designer with concern-specific, easy understandable, and standard aligned modeling languages.

THESIS APPROACH 1.4

The modeling and analysis approach for self-adaptive software systems presented in this thesis has been approached using the following steps.

1. A literature study on self-adaptive software systems from a software engineering perspective with the goal to define the notion of self-adaptivity, identify gaps in the engineering process of self-adaptive software systems, and a first selection of high-level concepts for specifying self-adaptivity.

2. Identification of new high-level concepts necessary to describe the concern of self-adaptivity during high-level design along with a formal definition of these high-level concepts and a high-level integration into stan-

dard software engineering approaches.

3. Definition of artifacts necessary to specify the concern of self-adaptivity, mapping these concepts into a concrete language that fulfills various requirements (UML), and the definition of a UML extension and its formal semantics to cover all new concepts.

4. Definition of a formal quality assurance framework based on the UML extension, first using a naive brute-force approach to gather theoretical results, and second a performance inspection and the definition of a multi-staged model-checking approach with focus on scalability.

5. Integration of the created artifacts and techniques into existing software engineering methods, primarily by the definition of interfaces to other phases.

1.5 Scope and Non-Objectives

The scope of this thesis is the high-level design phase of a software engineering method that addresses the concern of self-adaptivity separately. The approach integrates between the requirements phase and the low-level platform-specific design phase. It is intended to cover the specification of the self-adaptivity concern during the high-level platform-independent design.

The approach aims at supporting the specification in terms of constructive and analytical methods, i.e. the use of concern-specific languages and concern-specific quality assurance methods to assure high-quality specification of self-adaptivity in spite of the increasing complexity of self-adaptive software systems.

This thesis does not extensively cover the process of inferring or identifying self-adaptivity during the requirements and design phase. Further, this thesis does not cover the transformation of the created platform-independent models to the corresponding platform-specific models.

1.6 Publication Overview

The approach presented in this thesis has been influenced by research in various different disciplines. In the following, we will briefly describe the publications that were created in the course of this PhD thesis and how they influenced the approach presented in this thesis.

Most importantly, there have been several publications on the topic of **self-adaptive software systems** themselves, describing the Adapt Case Modeling Language [LNGE11, LM12], the corresponding quality assurance approach [LGSE11, LTGE12], and the holistic engineering for self-adaptive software systems in [LE13]. Further, there is still ongoing work on performance engineering and analysis for self-adaptive systems [BLB12, BLB13] that will be integrated with the presented approach in future.

Our approach heavily bases on **model-driven development** techniques. We extensively gathered experience while providing a model-driven approach towards spreadsheet design as presented in [LEE12]. Especially, the propagation of change on type models to their respective instance models has parallels with adaptation on type level. A model-driven approach towards quality of product and engineering process has been presented in [LBFW10]. Here, especially the relation of quality to requirements has been studied.

A great part of our approach is the development of a new modeling language. In the course of a work about a framework modeling language [CBE+10], we investigated **language engineering** foundations in terms of parameterizable meta models. In ongoing efforts of the *Comparing Modeling Approaches* community, we develop a catalog of comparison criteria for modeling languages with focus on composition techniques [GAA+13, MAA+12].

Another approach that was influencing our *process adaptation techniques* deals with the comparison and merging of **business processes**. Particular parallels to the presented approach are the discussion of conflicting change operations and compound operations (for adaptation) [GL12, GKLE10, GLKE11, GLKE10, GKLE11].

Finally, our approach is built to be integrated into standard **engineering methods**. Creating a good engineering method in general has been studied in [FBLE13] and for requirements engineering in particular in [FBGL+13] (in German). Applying our approach for modeling self-adaptation to engineering methods themselves has been studied in [GLE12] (in German).

1.7 STRUCTURE OF THIS THESIS

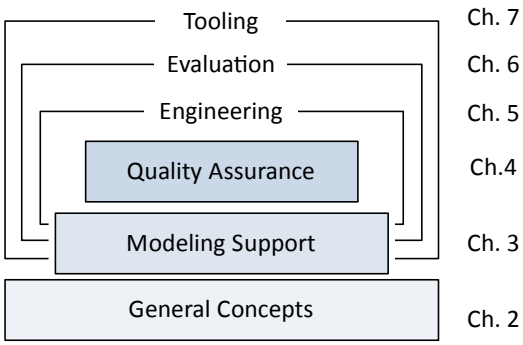


FIGURE 1.6. Structure of this Thesis

This thesis is structured into seven main chapters as illustrated in Figure 1.6.

- In Chapter 2, the foundation for the approach presented in this thesis is laid. Self-adaptive systems will be defined in more detail, modeling approaches will be described and the development of domain-specific languages will be discussed including the syntax and semantics definition. The latter will be used for the quality assurance approach presented in this thesis.

- Chapter 3 will expand on modeling self-adaptive software systems. A section dealing with the requirements analysis for a modeling language will be followed by the concrete description of the language.

- In Chapter 4, the quality assurance approach is presented. In this section, the formal semantics specification of the modeling language is given and quality properties are defined. On the basis of both, the approach is described in detail. Finally, the techniques to optimize the quality assurance approach concerning performance is detailed.

- In Chapter 5, method fragments will be sketched using the SPEM Profile [Obj08] that allow the integration of the presented language and the quality assurance techniques into a UML-based software engineering process.

- In Chapter 6, an evaluation of the overall approach is presented. The evaluation includes an illustrative case study as well as two assessments and two experiments .

- Chapter 7 describes the workbench that has been created to assist a modeler to specify self-adaptive systems. The chapter describes the editors

for the language as well as the functionality provided for the quality assurance approach.

- Chapter 8 concludes the thesis and discusses future work.

Related work for all topics will be discussed in the respective Chapters 3, 4, and 5 with particular focus on the modeling language, the quality assurance approach, and the engineering process, respectively.

- Finally, the appendix starting on Page 257 describes the meta model definitions for the ACML. The complete language specifications can be obtained from the website at [Luc13].

2

Foundations

"Goals and objectives are based on theories and foundations."

– Abdolkarim Soroush

2

FOUNDATIONS and general concepts that are needed to understand the approaches presented in this thesis are elucidated in this chapter. First of all, in Section 2.1, we will introduce the running example that will be used throughout this thesis. In Section 2.2, we will briefly pick up the definition of self-adaptive software systems and describe the different existing types of self-adaptivity, i. e. the modeling dimensions of self-adaptation. In Section 2.3, we will expand on the current state of the art in modeling-driven software engineering in general and discuss its suitability for modeling self-adaptive systems in particular. Further, we describe how existing modeling languages can be extended to define concern-specific modeling languages. Finally, in Section 2.4, we present techniques for quality assurance in software engineering in general.

bCMS – The Running Example 2.1

Throughout this thesis, we will use the running example of a Crisis Management System called bCMS. The example's foundations are taken from the bCMS case study [CCG+12] and extended by self-adaptive features. The resulting system is named bCMS-ADAPT. The initial purpose of the bCMS case

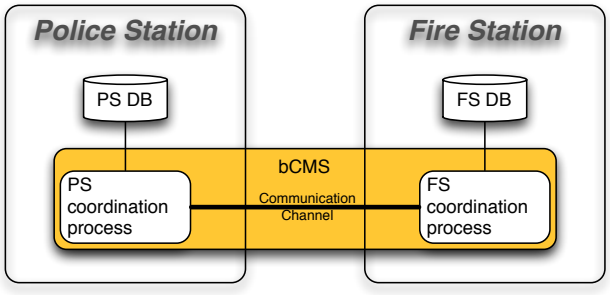


Figure 2.1. bCMS: Crisis Management System

study is to serve as modeling subject for comparing different modeling approaches at the workshop series on Comparing Modeling Approaches (CMA) which is held at the MODELS conference regularly. As shown in Chapter 6, this thesis' results have been submitted to the CMA workshop for evaluation purposes.

Distributed crisis management system

The bCMS system is a distributed crisis management system that is responsible for coordinating the communication between involved fire station and police station coordinators. The bCMS's core are two small business processes—one for each communication party—that systematically structures the different parties' communication (see Figure 2.1). The bCMS global coordination is the result of the parallel composition of the two interactive business processes, one for the police station coordinator and the other for the fire station coordinator. Each party maintains its own local database, a central data storage does not exist.

The bCMS starts operating when a crisis is detected and declared at both parties independently. The basic use case which is focused in this case study involves the establishment of communication, the exchange of crisis details, the coordinated development of a route plan and the dispatch of vehicles on both sides.

Specification using the UML

The specification of the bCMS is given in terms of component diagrams and use case diagrams. Figure 2.2 shows a very basic use case diagram. A coordinator may communicate with another coordinator. Depending on whether the

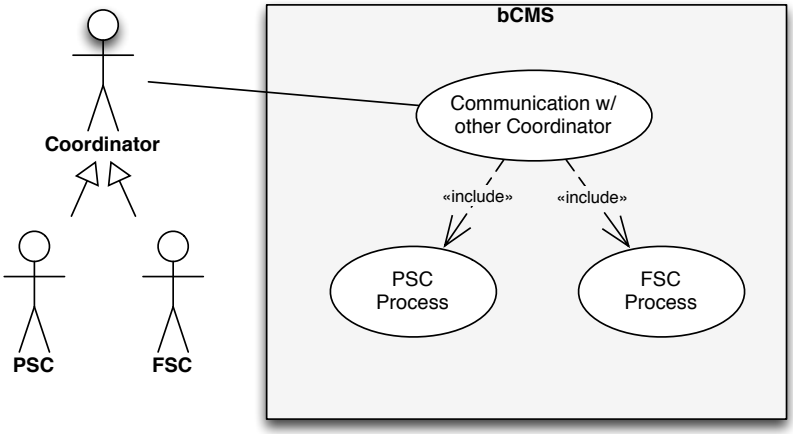


FIGURE 2.2.
bCMS: Use Cases

coordinator is a police station coordinator (PSC) or a fire station coordinator (FSC), the corresponding sub use case is included describing specific actions.

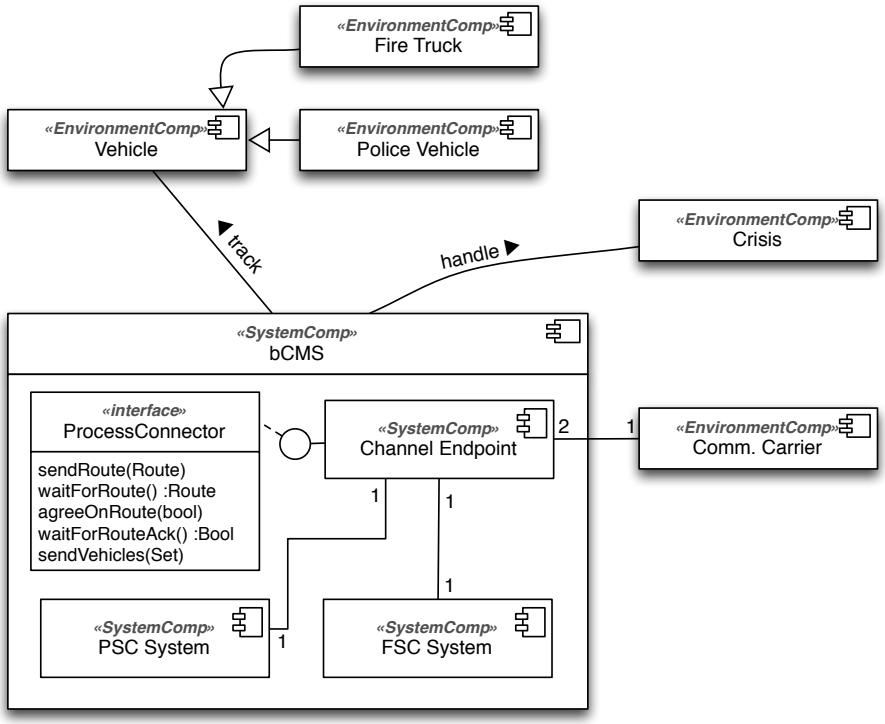


FIGURE 2.3.
bCMS: High-Level Component Diagram

In Figure 2.3, a basic component diagram is shown. The main component is the bCMS component which contains a separate component for each party (police and fire station) as well as two channel endpoints that connect the respective party to the communication carrier (Comm. Carrier). The channel endpoints implement the application specific interface `ProcessConnector` that allows the synchronization of both the PSC and the FSC process. Vehicles and the crisis

itself are modeled as separate components tagged as `EnvironmentComponent` to indicate that they are out of the system scope.

Figure 2.4 gives a very high-level communication overview. Both the PSC system and the FSC system own a behavior and use the communication carrier (channel) for synchronization of these behaviors (processes). The process definition is given in Figure 2.5.

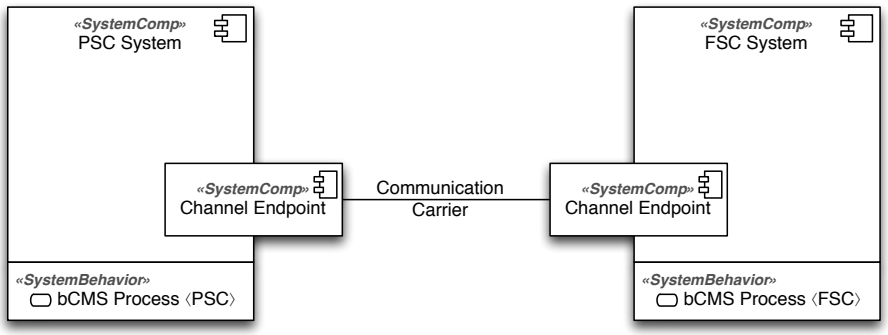


Figure 2.4. bCMS: Communication Overview

The bCMS main process describes the flow of the use case *Communication with other Coordinator*. First, the communication is established followed by the exchange of crisis detail. Next the route plan is developed and the vehicles (fire trucks and police cars) that are send to the crisis location are selected. The activity *Develop Route Plan* is an abstract one since it differs for both parties as shown in Figure 2.6.

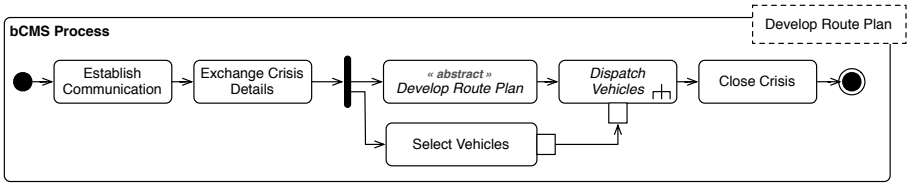


Figure 2.5. bCMS: Main Process

The fire station coordinator creates a route and sends it over the channel (see Figure 2.6 right). The police station coordinator retrieves the route and either agrees on it or rejects it (see Figure 2.6 left). If rejected, the FSC creates a new route. The activities contain UML `CallOperationActions` that call operations which are defined in the component diagram shown in Figure 2.3. In parallel, the parties select the vehicles that will be sent to the crisis.

After the route plan has been developed, the vehicles are dispatched. The main process defines the activitiy *Dispatch Vehicles* that is refined in Figure 2.7.

The activity *Dispatch Vehicles* obtains as a parameter a set of vehicles for each of which the coordinator states whether it has been dispatched, arrived, and completed its objectives. If all vehicles have completed their objectives, the

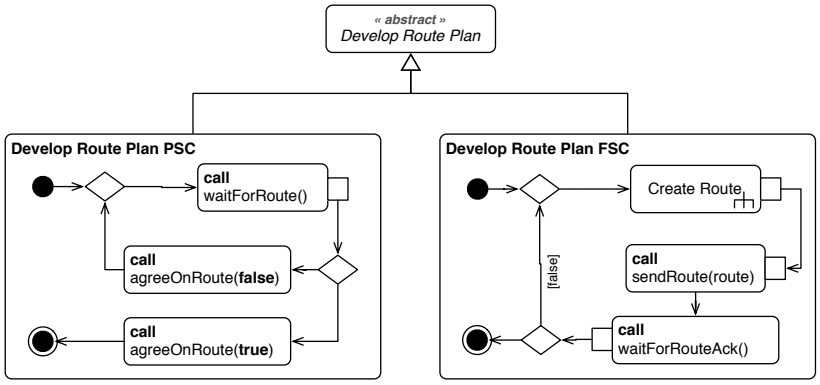


FIGURE 2.6.
bCMS: Activity
Develop Route Plan

crisis is closed. Of course, if the system is completely specified, the actions to set the vehicles' states have to call defined operations as well. However, while using this scenario, we are not interested in these details and thus abstract from them.

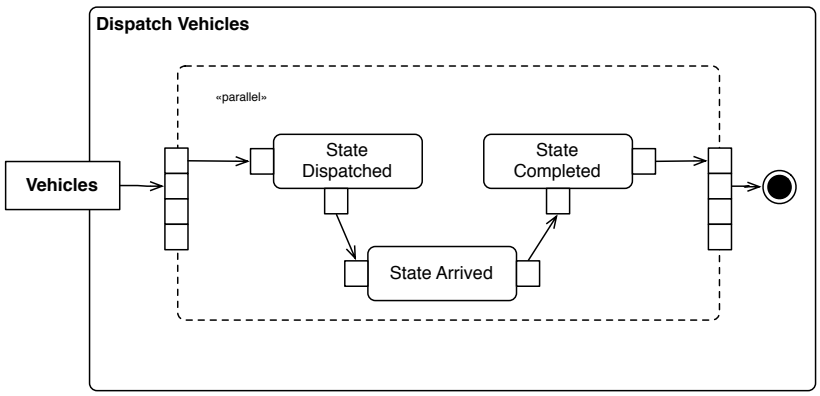


FIGURE 2.7.
bCMS: Activity
Dispatch Vehicles

Figure 2.5 is modeled to be a generic process having a template parameter named *Develop Route Plan*. This parameter is bound separately for the PSC and for the FSC as shown in Figure 2.8. As shown in Figure 2.4, these instantiated processes are defined to be the owned behavior of the respective components: PSC system and FSC system.

FIGURE 2.8.
bCMS: Instantiated
bCMS Main Process

THE SYSTEM CLASS OF SELF-ADAPTIVE SYSTEMS 2.2

Self-adaptive software systems are an emerging class of systems that adjust their behavior at runtime to achieve certain functional or quality of service objectives [WMA10]. Usually, self-adaptive software systems adjust their behavior in response to some stimulus from their context, while context describes any information which can be used to characterize the system's situation, e.g. the operating environment or the system itself. Therefore, self-adaptive software systems usually monitor themselves and their context.

ADAPTATION TO CHANGING CONTEXT

Unfortunately, despite its widely use in computer science and software engineering in particular, there is still no common understanding of self-adaptation, but rather there are several different definitions for self-adaptive software systems. In general, systems that monitor themselves and their environment are variously called self-adaptive, self-healing, or self-managing systems [CGS05]. Further, one distinguishes between adaptable software, adaptive software, and self-adaptive software. "Adaptable systems can be adapted to a particular [context], whereas adaptive systems adapt themselves to a [particular context]." [CE00] Here "adaptive systems" seems to include the self-adaptive software described by Oreizy et al: "Self-adaptive software modifies its own behavior in response to changes in its [context]." [OMT98]

Based on the autonomous computing feedback model called MAPE-K [Mur04] (see Figure 1.1), we extend these definitions by explicitly including the concepts of sensors that are used to gather context information and effectors that are used to actually adjust the self-adaptive software system. Further, we explicitly define the notion of context. The definition is given as follows:

EXTENDED MAPE-K FEEDBACK LOOP

Definition 1 *A self-adaptive software system adjusts its own structure and behavior through effectors in response to its perception of its context using sensors. Context is any information which characterizes the state of the self-adaptive software system.*

Since, self-adaptive software system are explored from various disciplines such as control theory, autonomic computing, and software engineering, Definition 1 deliberately leaves room for interpretation. That is, while on an abstract level the disciplines' definitions for self-adaptive software system nearly equal, they strongly differ in the details, e.g. regarding the different types of self-adaptation. This issue is addressed by creating taxonomies that describe the various different dimensions of self-adaptation [ST09, ALMW09, HMK09, DNGM+08].

TAXONOMIES USED FOR CLARIFICATION

From a *modeling* perspective, a self-adaptive software system has several dis-

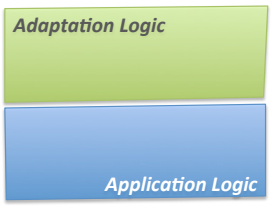


FIGURE 2.9.
Two Different Concerns: Adaptation and Application Logic

tinctive concerns including the system's adaptation logic. As shown in Figure 2.9, the system's adaptation logic is often understood as a distinctive unit, sitting on top of and observing and adapting the application logic. In the sense of multi-view modeling, the adaptation logic concern should be modeled separately. However, for a distinctive modeling of the system's self-adaptivity, Definition 1 is still too vague. Several questions regarding the modeling of self-adaptive systems appear. For instance, what is to be adjusted? How is the system adjusted? What is triggering the adjustment? And what is the state of a self-adaptive system? Hence, to be more precise on the requirements for a modeling language for self-adaptive systems, we are describing a detailed taxonomy for the *modeling* dimensions of the adaptivity concern in the next section.

MODELING SELF-ADAPTIVE SYSTEMS

TAXONOMY FOR MODELING SELF-ADAPTIVE SYSTEMS

2.2.1 MODELING DIMENSIONS OF THE ADAPTATION CONCERN

None of the taxonomies cited above is directed towards the specification (i.e. modeling) of self-adaptive software systems. Thus, we created our own taxonomy integrating the relevant parts of the referenced ones. Our taxonomy describes different dimensions of self-adaptation from the perspective of a self-adaptive software system modeler (i.e. designer). For instance, in our taxonomy, dimensions regarding the implementation types (e.g. open versus closed adaptation) are out of scope since they reflect decisions that are either taken during later software engineering phases or do not have an impact on the creation of specification artifacts during high-level design.

TAXONOMY USED FOR SCOPING

The goal of presenting this taxonomy is to precisely define the scope of self-adaptation that is covered by the techniques proposed in this thesis. More precisely, we will use the taxonomy to define the expressive power of the modeling language proposed in Chapter 3.

As shown in Figure 2.10, we divided our taxonomy into three different root dimensions: *why*, *what*, and *how*. This reflects a common schema taken e.g.

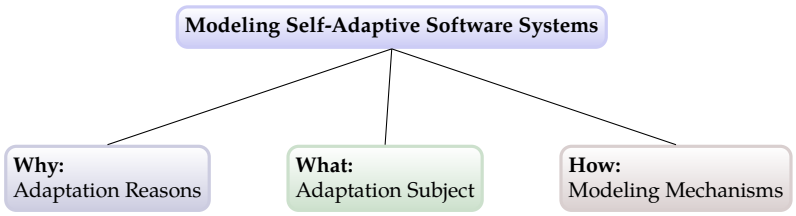


Figure 2.10. Taxonomy for the Modeling of Self-Adaptive Software Systems

in [DNGM+08]. The *why dimension* reflects reasons for adaptation that can be addressed using a particular modeling language. The *what dimension* reflects the subject of adaptation, i.e. the adaptation of what kinds of subjects can be described using a particular modeling language. And finally, the *how dimension* reflects on the various adaptation mechanisms that can be described using a particular modeling language. In the following, each root dimension will be described in more detail.

Adaptation Reasons Self-adaptation capabilities can be introduced into a software system for different reasons. For instance, a self-adaptive software system might react to erroneous system states or optimize its own behavior to perform a specific task more efficiently. These ***reasons*** can be understood as self-* properties. There are different sets of self-* properties focusing on different system classes [ST09], e.g. self-organization for highly distributed, decentralized systems that exhibit emergent functionalities. Since, we do not particularly focus on this system class but rather focus on traditional software systems that may self-adapt as described above, we use the categorization for self-adaptive software systems as proposed in [ST09]. Figure 2.11 reflects the four kinds of ***reasons***, i.e. self-* properties, a designer may model self-adaptation for.

adaptation reasons are self-* properties

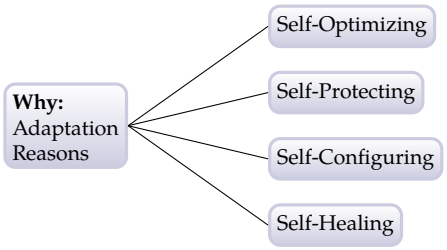


Figure 2.11. Reasons for modeled Adaptation

Self-Optimizing Self-adaptation might be modeled for the reason of self-optimizing a software system, e.g. minimizing the operating costs in a cloud scenario. Self-optimizing adaptation does not require any error or misbehavior of the system to occur.

Self-Protecting Self-protecting adaptation might be modeled for security reasons. The system undertakes some action to protect itself or the user.

Often, self-protecting behavior requires some kind of pro-active self-adaptation (see below).

Self-Configuring If a system is meant to flexibly adapt to different contexts, this is usually a matter of self-configuring adaptation. For instance, a system configures the used network connection type autonomously.

Self-Healing Self-healing adaptation requires an error or some other misbehavior to occur in the system itself or within its context. For instance, if a used external service is not available any more, it could be replaced with another one, being a self-healing adaptation action.

A language that allows the modeling of the self-adaptivity concern may be specialized to a specific self-* property, e.g. by providing specialized modeling elements, or it may support a subset or even all self-* properties by providing more generic modeling elements.

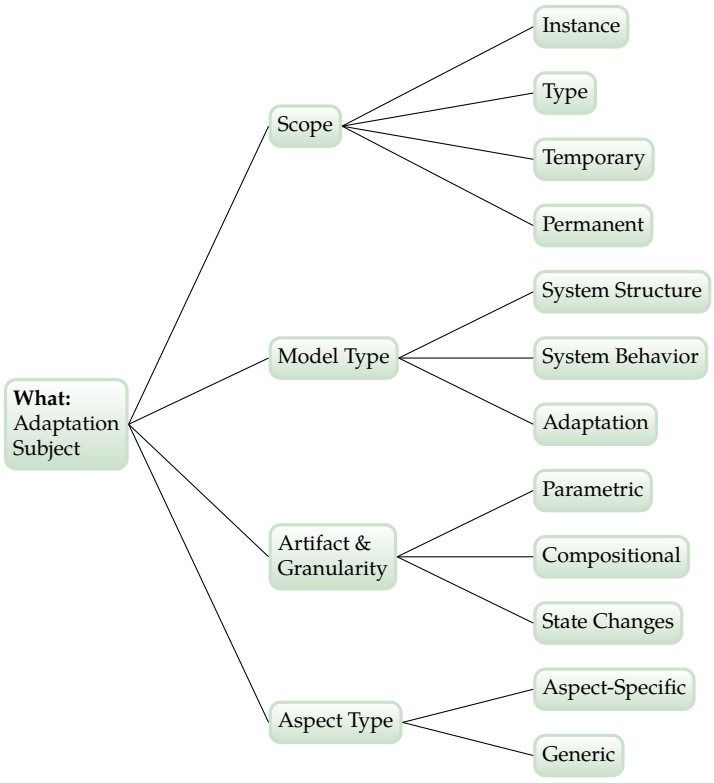


FIGURE 2.12. The Subject of modeled Adaptation

WHAT IS TO BE ADAPTED?

Adaptation Subject Figure 2.12 lists different dimensions to characterize the ***subject of self-adaptation*** that is to be specified. The **scope** distinguishes whether the adaptation action affects the type of an encapsulation unit or a particular instance of a type. Further, the effect of temporary adaptation actions holds only for specific instances or contexts while permanent adaptation actions modify the software system permanently, and thus, have an effect on

other instances and contexts. Usually, permanent adaptation is implemented using type adaptation while temporary adaptation are applied to single instances such that the adaptation is put out of order when the instance gets removed. Thus, especially during modeling, temporary and instance adaptation as well as permanent and type adaptation might coincide.

When modeling a software system, the designer usually distinguishes between structural and behavioral models (i.e. **model type**). This distinction can also be made when modeling adaptation. Some modeling languages support the definition of adaptation of software processes (i.e. system behavior), while others support the definition of adaptation of the system's structure, e.g. replacing a particular component or service by several others. As third model type, we consider the adaptation specification itself. That is, a modeling language might support the specification of meta-adaptation or *higher-order self-adaptation*, i.e. the adaptation of adaptation specifications.

The dimension of **artifact & granularity** distinguishes whether the modeling language supports the adaptation specification of only parameters (e.g. variable values) or the software composition (e.g. recomposition or restructuring of software processes), too. Further, the modeling language might support the specification of adapting the current system state, e.g. represented by state charts.

Finally, the **aspect type** dimension distinguishes between languages that focus on a specific aspect (e.g. performance or usability) or exhibit generic modeling means that support the adaptation specification of any aspect.

Modeling Mechanisms Figures 2.13 and 2.14 show the third root dimension which contains various sub dimensions distinguishing the different ***mechanisms of self-adaptation*** that a language might support to model.

HOW TO ADAPT?

First, there are different **locations** where adaptation may take place. Horizontally, adaptation may either be modeled to locally modify the self-adaptive software system usually with the use of only local information, or adaptation may be modeled to globally modify the system with global information available. Usually, global adaptation implies a separation of adaptation logic from application logic. Vertically, the adaptation may take place on the various different layers in a software system, where process, service, and component layer are just example layers known from process-driven service-oriented architectures.

Considering the **timing of adaptation**, the modeling language might support the specification of *reactive* and/or *proactive* adaptation. Opposed to reactive

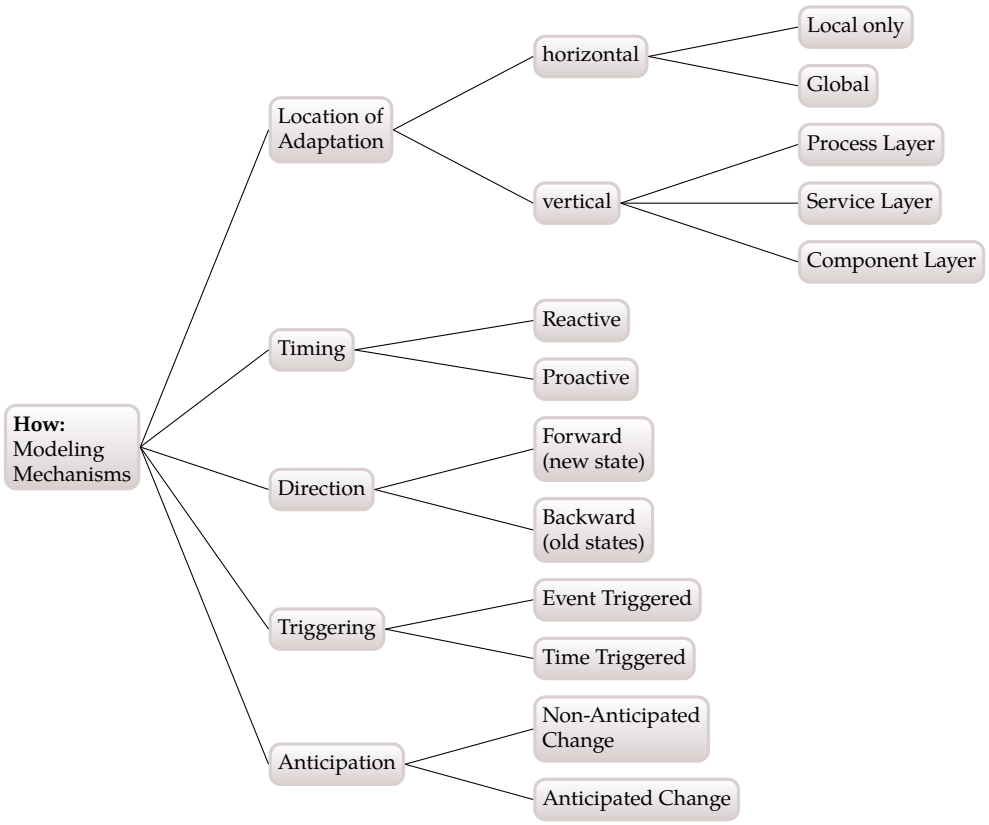


FIGURE 2.13. Modeling Mechanisms for Self-Adaptation

adaptation, proactive adaptation aims at predicting the future and avoiding undesired system states. Therefore, the modeling language must support the use of prediction models and usually the use of history data.

Further, adaptation modeling may support two different **directions** of adaptation. *Forward* adaptation is the standard case, where by adaptation a new system state is generated. *Backward* adaptation describes the case, where adaptation resets the system to an old (functioning) state. Therefore, the modeling language needs to support accessing old states.

Adaptation may be **triggered** in various different ways. That is, a modeling language may support the specification of timed triggers or event triggers (e.g. a change of an observed variable or the occurrence of a signal).

Finally, there might be modeling support for **anticipated** and/or non-anticipated changes and adaptation actions.

As shown in Figure 2.14, another dimension is the degree of **automation**. That is, a modeling language may support the specification of autonomous, human, and/or interactive adaptation.

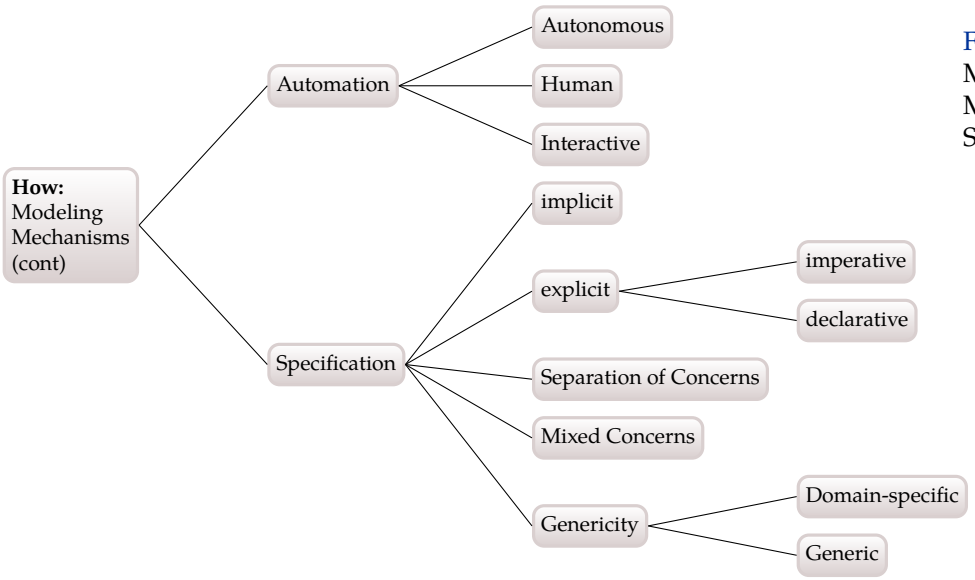


Figure 2.14. Modeling Mechanisms for Self-Adaptation (cont)

Finally, there are different dimensions concerning the **specification** itself. The specification can be *implicit*, that is tightly interwoven with the rest of the system's specification. If adaptation is specified *explicitly*, it may either be given imperatively, i.e. the concrete actions to be performed are specified, or it may be given declaratively, i.e. the adaptation goals are specified without describing the approach to meet the goals. Further, the adaptation concern can be specified *separately* from other concerns or *mixed* with other concerns. Finally, the modeling language may focus on a specific domain, e.g. by providing domain-specific modeling elements, or provide rather generic modeling elements that suit every domain.

TAXONOMY USED FOR COMPREHENSION AND COMPARISON

This taxonomy helps in understanding the adaptation concern in greater detail. Especially, using this taxonomy, modeling approaches for self-adaptation may be characterized and compared precisely. In Chapter 3, we will use this taxonomy to exactly state the expressive power of the proposed modeling language. Although this taxonomy helps understanding in which cases self-adaptation *can be* modeled in which way, in some cases it is still difficult to distinguish between adaptation and application logic, that is to draw a sharp line between the two concerns from Figure 2.9. We will discuss this problem in the next section.

2.2.2 ADAPTATION OR APPLICATION CONCERN: A DESIGNER'S CHOICE

LINE BETWEEN ADAPTATION AND APPLICATION LOGIC IS BLURRY

Although there are numerous efforts to come up with an even more precise definition of self-adaptivity, this aspect of software seems to be hardly tangible. The problem of defining a self-adaptive systems often arises while giving an example of such a system. Often, self-adaptation features are seen as application features rather than adaptation features. For example, consider a mobile device that adapts its connection type to the respective availability of wireless LAN or GPRS. One might argue that this feature is a simple if-then-else condition and therefore no self-adaptivity. Further, one might argue that adapting the connection type is a core feature of a mobile device which is essential for a mobile device and therefore considered as core functionality but not as self-adaptivity. Finally, diving deeper into a possible implementation of such a system, one might argue that there is a component which is responsible to always maintain a network connection if possible. This component is existing for this very reason only, and therefore, the adjustment of the connection is not self-adaptivity but application logic—the logic, the application, i. e. the component, is designed to provide. In this manner, it is easy and hard at the same time to argue for and against self-adaptivity.

A similar problem arises in the area of non-functional requirements. While the compatibility of a piece of software with Windows 7 is a non-functional requirement for one person, while another person considers this as a central feature which is a core functionality of the software. The answer to this situation is "*it depends*". Primarily, requirements classifications are meant to provide a better overview, provide particular views onto the requirements (e. g. all performance requirements), or enable semi-automatic requirements analyses. Usually, it depends on the final purpose of the classification how to classify a particular requirement.

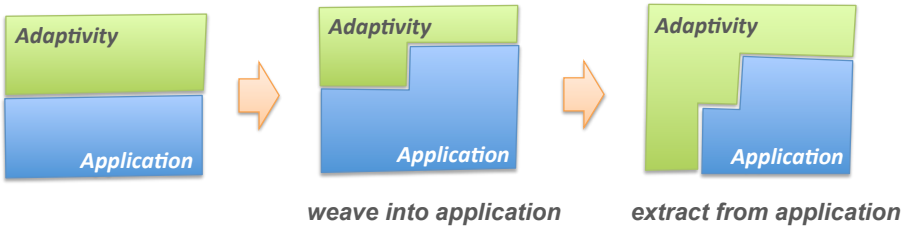


FIGURE 2.15. Changing the Border between Adaptation and Application Logic

We adopt this point of view for self-adaptation. That is, we consider the decision of a feature to be adaptation or application logic to be a designer's choice. In many cases, the choice is rather natural, in all other cases, the designer is free to model the feature as adaptation or application logic. In our view, the sepa-

rate treatment of adaptation aspects during software engineering is due to the wish to separate concerns. This in turn is meant to help the designer to manage complex software constructions. That is, the designer decides which feature is adaptive and which not and may of course swap a feature from adaptation logic to application logic and vice versa as indicated in Figure 2.15. Needless to say that a language for modeling adaptation logic should be close to languages to model application logic in order to ease this kind of swapping. Therefore, it is important to understand how software systems are modeled nowadays. In the next section, we will describe the techniques of model-driven software engineering and discuss the use of existing techniques to model the adaptation logic concern.

A designer's choice

Model-driven Software Engineering (MDSE) 2.3

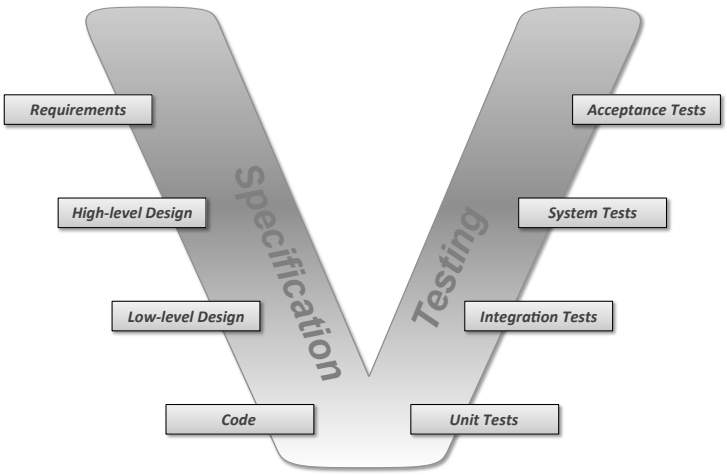


Figure 2.16.
The V-Model

Usually, software engineering methods are used to successfully accomplishing large software projects. By software engineering methods we denote the full set of elements needed to describe a software development project, including the development process and its activities, the artifacts produced and the tools and techniques that are employed as well as relationships between these concepts [ES10]. Applying a software engineering method creates a common understanding and coordinates the activities to perform and the creation of artifacts. There exist several widespread software engineering methods based on different process models (e.g. RUP [Kru03], V-Model XT [KNR05] or Scrum [Sch97]). The V-Model is a process model for the software development which relates early phases (left branch) to later phases (right branch)

Software engineering processes coordinate engineering activities

using tests of the corresponding level (see Figure 2.16). It is particularly suitable to demonstrate traditional software engineering methods since the basic structure is reused by the majority of used methods: A requirements engineering phase is followed by the creation of a high-level, platform-independent design. This design is detailed into a more technical low-level representation that may be platform-specific, followed by the implementation. At implementation level, unit tests are used for quality assurance. On low design level, integration tests are leveraged and on high design level, system tests are performed. Finally, on requirements level again, the system is tested for acceptance by the customer (validation). Of course, plenty of variations exist that, e.g., refine or put emphasis on single phases.

2.3.1 Multi-View Modeling & Concern-Specific Modeling Languages (CSML)

Model-driven software engineering (MDSE) is based on the usage of models to specify artifacts, generate code and document the system. Let us briefly investigate different ways, models can be used during MDSE. This allows us to provide a better understanding of the characteristics our approach exhibits. According to [EG00], models consist of three different kinds of constituents, different in nature. Model constituents are "model parts that reflect the way one wants to separate one's concerns" [EG00]. The first kind reflects the elements, that originally describe the system to build. These constituents reflect exactly how the model has been constructed. The second kind of constituents form different representations of the original model. This representation is a different form the model takes when represented depending on a particular use. These representation are often called *views*. They consist of the same type of constituents as the original model does, but differently represented. The third kind of constituents are *aspects*. Aspects describe general characteristics or concerns for the model as a whole. An example for aspects is security that might implement a login feature that is woven into the model at different places.

MODEL CONSTITUENTS REFLECT CONCERN SEPARATION

These constituents allow to model a system differently, e.g. using the corresponding paradigm. Views are used in view-based modeling approaches. The challenge of view-based modeling is to ensure consistency of the different view models, especially, since these views may overlap and same elements may be represented differently. Aspect-oriented modeling usually requires an

VIEWS

ASPECTS

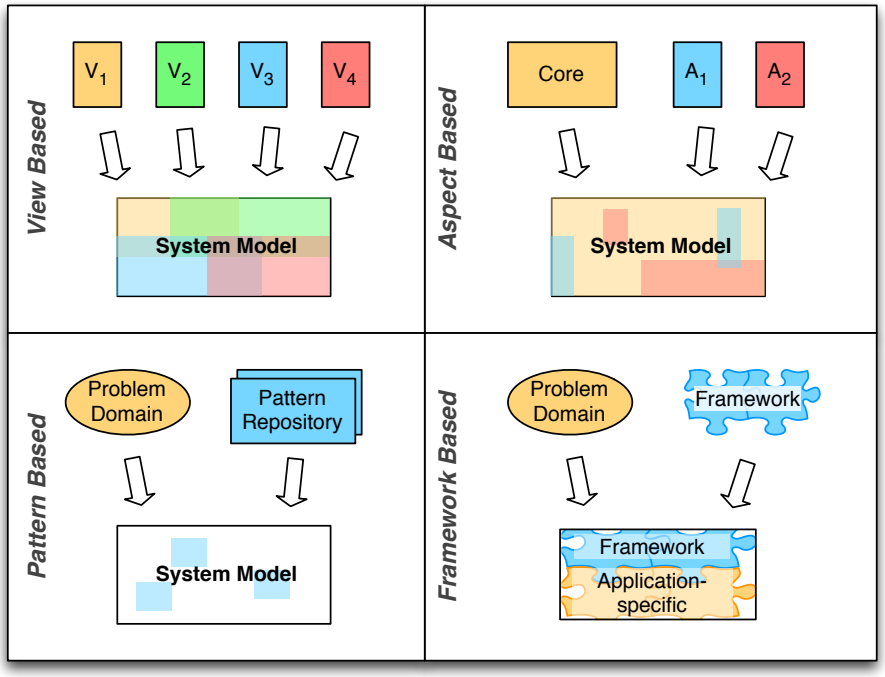


FIGURE 2.17. Modeling Approaches for Describing Concerns

aspect weaver to be in place. Different from views, aspects do not focus on a specific representation of the system model, but usually address single cross-functional concerns that are defined separately and loosely coupled with the rest of the models. The two different modeling approaches are depicted in the top of Figure 2.17. Of course, modelers and language designers may mix the two approaches to their needs. Two other approaches to modeling are *pattern*-based modeling and *framework*-based modeling. Similar to views, pattern may be composed with each other to describe the system. Frameworks already described a large portion of the system for a particular domain or concern and may be used by specialization or implementation of hooks.

PATTERN

FRAMEWORKS

All four types of modeling approaches can be used for describing particular concerns. Usually, the language designer decides for a particular type of modeling approach that his concern-specific modeling language shall support or use. A concern-specific modeling language (CSML) allows the modeling and thus focus on a particular concern (such as security or self-adaptivity) opposed to domain specific modeling languages that focus on a particular domain such as gaming, embedded systems, process-driven service orchestrations, etc. For the concern of self-adaptivity, we will show in the next chapters how to create a particular system *view* for self-adaptivity to describe sensors, effectors, etc. and how adaptation rules will be specified in an *aspect*-like manner.

VIEWS AND ASPECTS ARE USED IN OUR APPROACH

2.3.2 Unified Modeling Language (UML)

UML, A DE-FACTO STANDARD

Software engineering methods often propose the use of specific modeling languages. A language that is compatible with most methods (including the V-Model XT) is the Unified Modeling Language (UML) [Obj11], the de-facto standard for object-oriented modeling. The UML was created with the goal to unite a variety of different object-oriented modeling languages in the early 90s and to create a common meta model [Obj00, BRJ05, Obj11].

STRUCTURAL AND BEHAVIORAL DIAGRAMS

The UML distinguishes two different diagram types: structural and behavioral diagrams. Structural diagrams, e.g. the class diagram, allow the specification of a system's static structure usually in terms of entities and their attributes and relationships. Behavioral diagrams, e.g. activity diagram, allow the specification of the system's or an entity's dynamic behavior. Activity diagrams allow the description of actions that are performed in a specific order. Another behavioral diagram, the state chart allows the description of a system's or entity's states and their transitions.

For each kind of diagrams, the UML consists of several different notations some of which will be detailed below. Every notation serves a specific modeling purpose, e.g. use cases are often used during requirements engineering and the early high-level design. Table 2.1 gives an overview of UML notations assigned to the software engineering phase they are most often used in. Of course, variations exist.

Structural Diagrams

The next paragraphs briefly introduce class, component, and object diagrams as these will be used in the following chapters.

Object Diagrams Objects are the foundation of object-orientation. Objects are abstractions of reality that highlight single characteristics that allow their distinction [Sha80, Boo94]. In the UML, an object is a rectangle with an underlined name (cf. Figure 2.18). Objects may have a state that is characterized by the concrete value of its attributes. Attributes are given in an additional compartment of the object rectangle. An attribute has a name and a value divided by an equal sign.

Objects may have relationships. They are denoted using lines, so-called links, between two objects. These links may have an underlined name and roles.

Phase	Language
Requirements Engineering	**Use Cases** are often used to describe requirements using a scenario-based approach. **Class Diagrams** are often used to model the domain model / problem domain. **Activity Diagrams** are often used to model business processes. Sometimes, they are already used during requirements engineering to describe scenario flows that are attached to use cases.
High-level Design	**Use Cases** are used to refine requirements and provide detailed scenarios. **Class Diagrams** are used for the creation of logical system models, i.e. entity types, their relationships, attributes, and sometimes even operations. **Component Diagrams** are used especially in component-based design to describe logical components and their interfaces. At this level of abstraction, components are often modeled as black-box components. **Activity Diagrams** are used to describe the detailed flows of scenarios attached to use cases or behavioral components. **Sequence Diagrams** are used to describe the interaction of (human) actors with the system. **State Charts** are often used to describe the state model of complex entity types (in class or component diagrams).
Low-level Design	In the technical design phase, basically the same notations are used as in the logical design. However, several design decisions are made that apply to a specific platform. Thus, the diagrams created in the logical design phase are detailed and attached with more technical information (i.e. technical identifiers). Often the logical structure is slightly modified to fit the specific needs of the concrete platform.
Implementation	On implementation level code is either generated from the models (model-driven engineering) or the code is created manually based on the models (model-based development). A third option on this level is to build or configure model interpreters that run the created models. Therefore, the models might need further refinements.

Table 2.1.
SE Phases and UML Notations

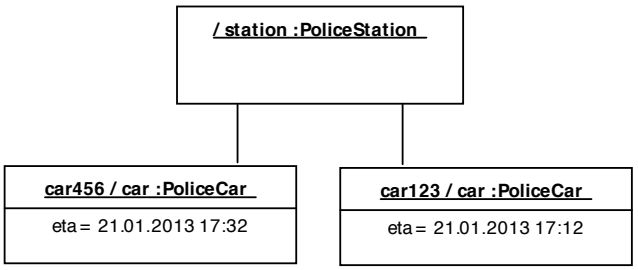


Figure 2.18.
UML Object Diagram

Roles are denoted at one or both ends of a link and describe the role an object in the relationship with the other object plays. Alternatively, roles may be denoted right after the object identifier divided by a slash (cf. with roles *station* and *car* in Figure 2.18). Links may further be unidirectional or bidirectional which is denoted by arrow heads at either one or both ends. Each object denoted has a unique identity, i.e. even if two objects have the same attributes with same values, they are different. However, these two objects might be of the same type. The type of objects is described using class diagrams (see the following section).

The relation between an object of some type and the type itself is called instantiation. In the UML meta model, objects are called instance specifications. Instance specifications are a basic concept that could be applied to every UML model element. That is, a component, an activity, a state, or an association could be instantiated using instance specifications. Instance specifications have slots which describe the value assignments (e.g. class attributes).

Class Diagrams As mentioned in the previous section, class diagrams describe types of objects. That is, a class subsumes the attributes, operations, and relationships of objects that are of the same type. An object with the type of a specific class is called an instance of that class.

As shown in Figure 2.19, a class is denoted with a named rectangle. In contrast to objects, class names are not underlined. Classes may have definition of attributes, operations, and relationships (so-called associations) with other classes. Just like objects, classes denote their attributes within a specific compartment. Attributes defined in classes have a name, too, but do not carry a specific value. Instead class attributes define a specific attribute type which can either be primitive (e.g. integer, float) or structured (e.g. another class from any class diagram). Classes define operations to modify of select object attributes. Operations are denoted within another compartment and have a name and define the type of the return value (e.g. integer, void, a class).

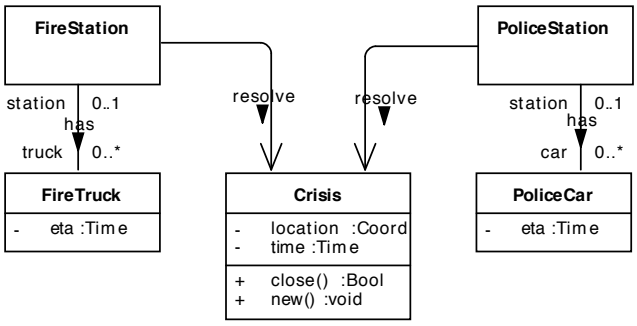


FIGURE 2.19.
UML Class Diagram

Associations may have a name (not underlined) and/or roles. Further, associations may be unidirectional or bidirectional. If an association is unidirectional, the association can be navigated into that particular direction, i.e. the associated class has access to the other class. An association may also be non-navigable which is denoted using a small x at the non-navigable end (or at both ends). Finally, associations may have cardinalities. Cardinalities define the lower and upper bound of the number of instances that may be associated with one another.

Aggregations are specialized associations that define a hierarchy, while the aggregating class is responsible for its aggregated classes. Aggregations are denoted using a diamond at the association end at the aggregating class. A specific aggregation is the composition which sometimes is described as strong aggregation. The additional properties of compositions are that an object of the composed class cannot exist without the composing object, the composed object is deleted cascadedly upon deletion of the composing object, and the composed object must only be composed by one composing object. Compositions are denoted with a filled diamond at an association end.

Classes may inherit all attributes, operations, and associations from one another if inheritance is specified. The inheriting class is the more specialized class while the inherited class is called the general class. Inheritance is denoted with an arrow between the specialized and the general class with a triangular arrow head at the general class's end.

As mentioned in Section 2.3.2, an object may be of the type of a particular class. The object is said to be the instance of the particular class. An instance of a class is denoted just as an object with a specific type (i.e. the class name) after the object's name divided by a colon.

Component Diagrams UML component diagrams allow the modeling of components with their interfaces and relationships. Components may be structured hierarchically. The concept of UML components allows to supply different implementations for a specific component. That is, components are usually implementation independent and may be used to logically structure the system to be. Figure 2.20 shows a component diagram that contains the most important notational elements. Component *FireStation* realizes the interface *Vehicle* and communicates with Component *Channel*. The concrete form of communication, e.g. the used interface, is left underspecified. The `Channel` in turn communicates with `PoliceStation` using a provided interface. Further, it requires another component that provides the interface `Vehicle`.

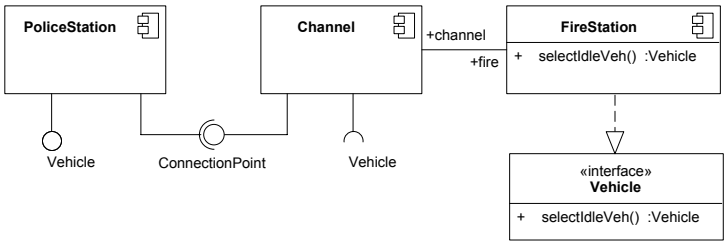


FIGURE 2.20.
Example Component Diagram

More precisely, components encapsulate elements and define interfaces to access these elements. They may contain other classes and interfaces. Interfaces may be defined to be required or provided. Interfaces may be described using classes with an attached stereotype «*interface*». Components may be connected with each other using these interfaces; a provided interfaces may be connected to one or more required interfaces.

It is important to distinguish elements that specify a component's interface from those that realize it. Specifying elements, i.e. interfaces, are realized by realizing elements, i.e. classes that are contained in the component.

Components are denoted like classes with an additional stereotype or the corresponding symbol in the upper right corner. Required and provided interfaces are denoted with a lollipop notation that may be further specified using the interface in class-like notation.

Behavioral Diagrams

In the following, we will briefly introduce the basic notation of UML use cases and activity diagrams.

UML Use Cases Usually, each use case is specified individually using a tabular form. Use case diagrams visualize the relationships between use cases as well as the actors. Both use cases and their diagrams are described in the following.

Use Cases Use Case are widely accepted to be the language for requirements engineering and the early high-level design [Coc00, Rup07]. Use Cases provide a view on the system under discussion by describing the interactions of (human) actors and the system. In particular, use cases specify *how* a system is used.

Use cases can be used on different levels of abstraction, e.g. business use cases are often used in the very early requirements phase while system use cases are often used during the logical design phase. The specification of the different use case types is slightly different. For instance, business use cases have a different scope than system use cases, while system use cases are way more detailed.

A use case is often given in tabular form, while the table lists all use case's attributes and values. The most often used attributes include the following:

Attribute	Description
Name / ID	A short but unique identifier for the use case. It is particularly used to refer to the use case.
Description	The main description of the use case given in natural language.
Scope	Describes the system or, if more detailed, the component the use case is located in.
Actor	Describes all participating actors.
Pre Condition	Describes the state of the system and its environment prior the use case's execution.
Post Condition	Describes the respective state after the use case's execution.
Trigger	Describes the event that makes the use case execute.
Standard Scenario	A description of the basic flow of execution, the use case describes.
Alternative Scenarios	A list of alternative flows of execution that are taken at particular condition.
Exceptional Scenarios	A list of flows of execution that describe the handling of exception.

The scenarios may either be given in tabular enumerative form, or be modeled using activity diagrams. Usually, all three different types of scenarios (standard, alternative, and exceptional) are modeled within a single activity diagram.

Use Case Diagrams A UML use case diagram provides an overview of all use cases. It shows the actors, the use cases, and the system boundary of the system under discussion. In addition, a use case diagram specifies the relationships between actors and use cases as well as relationships between use cases themselves. The most often used relationships between use cases are «*include*» and «*extend*» which are briefly explained next. A higher-level use case «*include(s)*» a lower-level use case if a scenario of the lower-level use case is carried out in a scenario of the higher-level use case. The relationship is modeled by calling the activity of the lower-level use case in at least one of the action steps of a higher-level use case's scenario. One use case is «*extend(ed)*»

by a second use case if the second use case extends a scenario of the first use case. The scenario of the second use case integrates into the scenario of the first at a defined extension point.

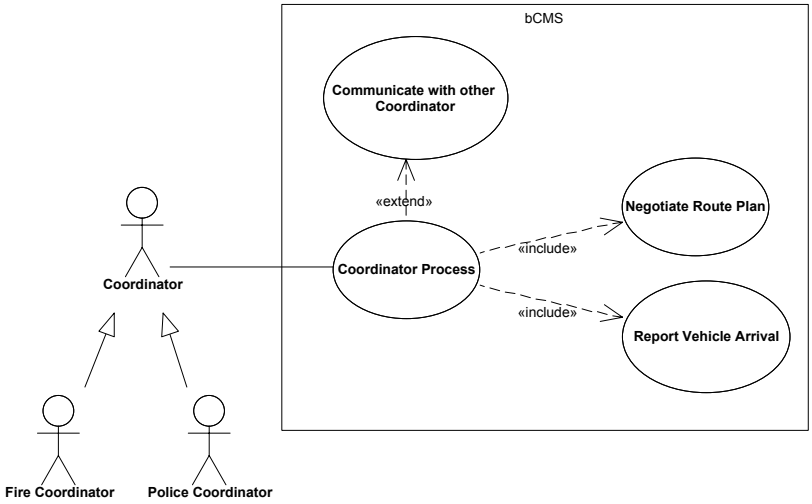


FIGURE 2.21. Example Use Case Diagram

Figure 2.21 shows the most important notational elements of use case diagrams. Use case `CoordinatorProcess` «*includes*» the use cases `Negotiate Route Plan` and `Report Vehicle Arrival` and «*extends*» use case `Communicate with other Coordinator`. The actor uses use case `Coordinator Process`. All use cases will be provided by the system (i.e. scope).

Activity Diagrams UML activity diagrams can be used to describe flows of actions (e.g. in a business process) using an easy visual language. As such, they fit best to describe the scenarios of use cases. Especially, if a use case contains a variety of alternative scenarios, an activity diagram helps to grasp a fast understanding since alternative scenarios are integrated with the standard scenarios using specific language elements such as decision nodes. Thus, use cases with high complexity, especially concerning exceptions and alternatives, are usually modeled with activity diagrams. Activities may also be used to describe the execution logic of operations e.g. defined in classes.

Activity diagrams are very similar to other process model languages such as the BPMN or basic workflow graphs. Activity diagrams have a token-offer semantics, which is an extended token-flow semantics [Obj11]. Further, activity diagrams propose a variety of specialized actions to structure the model (e.g. hierarchically) or perform specific model manipulation actions, such as calling an operation from another UML diagram (e.g. component diagram). The basic notational elements are shown in Figure 2.22. An activity may contain various

actions which may call other activities again, thus allowing hierarchical structuring.

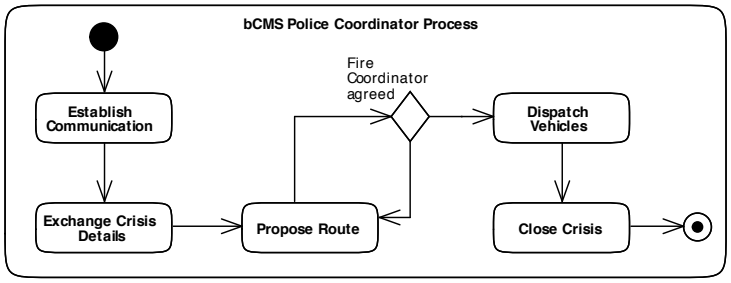


Figure 2.22. Example Activity Diagram

An activity may contain actions, control nodes, and object nodes. Actions may simply have a name with no explicit semantics. Additionally, the UML offers a set of predefined actions that exhibit a particular semantics, e.g. calling another activity or operation or setting the value of an object's attribute. Control nodes include decision nodes, fork nodes, final nodes, and initial nodes. Object nodes include pins that allow to pass objects from one action to another, an parameter nodes that allow passing parameters to an activity.

2.3.3 Adaptivity Concerns in the UML

Using the described techniques that are provided by the UML it is rather convenient to describe the application logic of a software system (cf. Figure 2.23). The different diagrams provide several different views onto the system, e.g. the class diagram provides a structural view onto the system whereas the activity diagram provides a behavioral process view onto the system. But how about using the UML to model the adaptation logic concern?

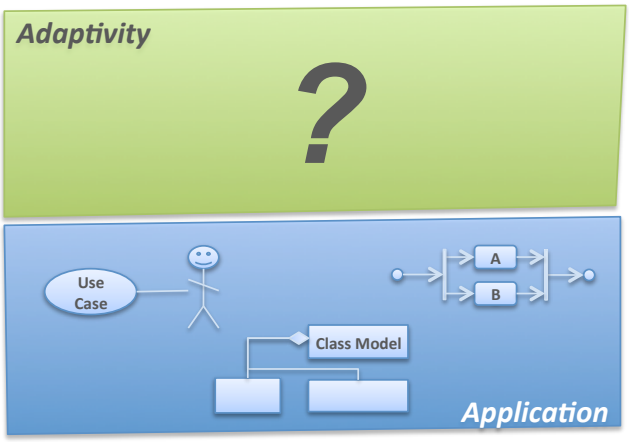


Figure 2.23. The use of UML for the Application Logic

NO EXPLICIT SUPPORT FOR SELF-ADAPTIVITY WITHIN THE UML

By now, there is no explicit support for modeling self-adaptivity in the UML. That is, adaptivity is usually modeled implicitly within the specification of the core application logic. Of course, adaptivity can be separated in UML models, e.g. by creating separated activity diagrams that describe flow of adaptation actions. However, since there is no concern-specific notation for explicit specification, the resulting models usually are hard to understand or at least often highly bloated. Moreover, particular self-adaptation (e.g. the adaptation of the system's type) cannot be expressed directly using the UML, but only by leveraging rather complicated workarounds. Further, it is very difficult to apply automatic quality assurance techniques specifically to modeled self-adaptation features, since these features are hardly distinguishable from core application logic.

NEED FOR CONCERN-SPECIFIC EXTENSION

To efficiently, and even more important, effectively specify self-adaptivity, the UML needs to be extended by concern-specific modeling means. Specifically, it must be possible to describe self-adaptivity in behavioral models or views as well as in structural models or views. Since self-adaptivity must be considered in high-level design at the latest and down to implementation, the mentioned extension must support the high-level specification as well as the low-level specification of self-adaptivity. Thus in the following section, we describe how to define concern-specific (UML-based) modeling languages.

2.3.4 Concern-Specific Modeling Language Definition

CSML DEFINITION BY META MODELING

Concern-specific and domain-specific modeling languages are usually specified using the technique of meta modeling. In meta modeling a modeling language is used to describe the abstract syntax of another. Today, usually MOF [Obj06] is used for that purpose but there are a variety of different languages, e.g. KM3 [JB06], Ecore [BBM03], MetaGME [LMB+01]. In the following, we will describe the basics behind meta modeling using the MOF by exemplary describing the specification of the UML. In the subsequent Section 2.4, we describe how to specify the semantics of meta model based modeling languages.

Meta Modeling

A meta model is a model of a set of models, i.e. meta models are specifications. Models are valid if they contain no false statements according to

the meta model (i.e. they are well-formed). Typically, meta models represent domain-specific models (real-time systems, safety critical systems, e-business) or concern-specific models (performance, user interfaces).

The domain of meta modeling is the definition of languages. A meta model is a model of (i.e. specifies) some part of a language. Which part depends on how the meta model is to be used. Parts include the syntax of a language (usually the abstract syntax), the semantics of a language (e.g. using DMM [Hau05]), views/diagrams, etc. A meta meta model is a model of meta models. Reflexive meta models are expressed by using themselves (see Figure 2.24).

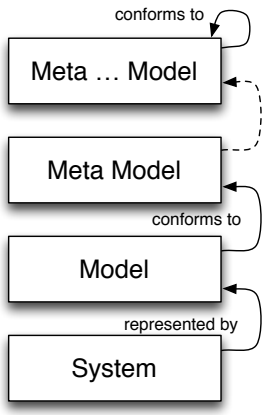


Figure 2.24.
Meta Model Levels

In the domain of language definition, meta models are usually used to define a language's abstract syntax, i.e. the language's concepts and their relations. Further, a language definition consists of one or more concrete syntaxes as well as static and dynamic semantics. Static semantics are often given by constraint languages such as the OCL or in natural language. Dynamic semantics are either given in natural language or using formal methods such as set theory, process algebras, graph transformations, or the like.

static and dynamic semantics

The Unified Modeling Language (UML) defines a meta model and the notation. The latter has been introduced in Section 2.3.2. The UML meta model defines the concepts of the UML and their relations. See Figure 2.25 for an example meta model excerpt for class diagrams. It defines classes to own operations and properties (attributes). An association connects two properties. Additional information is captured in the attributes. For instance, if `isComposite` is set to true, the association turns into a composition. An instance of this meta model (e.g. using object notation) represents a UML class model in abstract syntax. In concrete syntax, e.g. an association object is represented using a single line.

The UML meta model itself is an instance of the MOF (Meta-Object Facil-

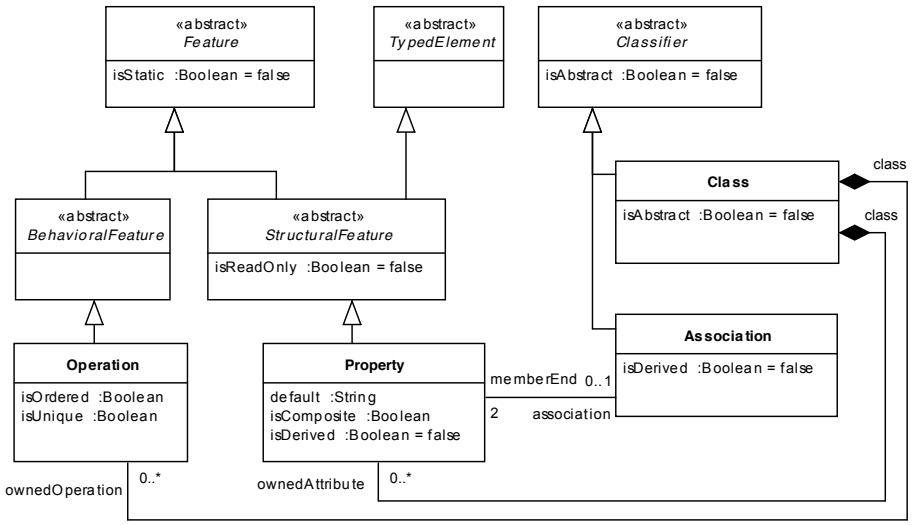


FIGURE 2.25. UML Classes Meta Model

ity) [Obj06], a closed meta model language. That is, the MOF itself is an instance of the MOF, again. The MOF itself is a very slim language that basically uses MOF::Classes for language definition. See Figure 2.26 for the UML meta model levels.

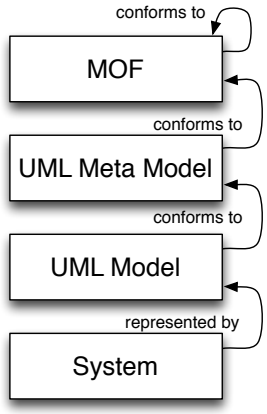


FIGURE 2.26. UML Meta Model Levels

Extension of Meta Model based Languages

HEAVYWEIGHT VS. LIGHTWEIGHT EXTENSION

Basically, there are two different mechanisms to extend a language: heavyweight and lightweight. Heavyweight extensions arbitrarily change a given meta model (i.e. add, delete, modify elements). Thereby, given semantic constraints may be violated or semantics may be changed, possibly leading to undesired side-effects. Lightweight extensions extend the language by refining single elements in order to apply concern-specific or domain-specific names and properties to these elements. However, lightweight extensions do not change the meta model and do not contradict with existing constraints and se-

mantics. The UML defines heavyweight extensions to be first-class extension mechanisms.

UML profiles for lightweight UML extensions

For UML, it is important to notice that heavyweight extensions require to change and thus know the meta model while lightweight extensions may be defined on M1 level without any knowledge of the meta model. This is because the UML offers the concept of UML Profiles that allow the definition of stereotypes (specializations) and tagged-values (additional properties) using a particular UML diagram.

> **UML Superstructure Specification, v2.3:** The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile. The only other restrictions are those inherent in the profiles mechanism; there is nothing else that is intended to limit the way in which a metamodel is customized.
>
> First-class extensibility is handled through MOF, where there are no restrictions on what you are allowed to do with a metamodel: you can add and remove metaclasses and relationships as you find necessary. Of course, it is then possible to impose methodology restrictions that you are not allowed to modify existing metamodels, but only extend them. In this case, the mechanisms for first-class extensibility and profiles start coalescing. [Obj11]

Common examples for UML profiles are the UMLSec [Jö2] for modeling security concerns and the SoaML [Obj12] for modeling services within a service-oriented architecture.

2.4 SEMANTICS & STATIC QUALITY ASSURANCE IN MODEL-DRIVEN SOFTWARE ENGINEERING

In the last section, we showed how model-driven software engineering is supported by the UML and how the UML can be extended to address specific concerns. The most important reasons to create models of the software system to be build is to enable a better understanding and allow for (automated) analysis of the modeled system. In this section, we describe several techniques how models can be formalized and analyzed automatically. Therefore, we briefly describe the model checking approach in Section 2.4.1. In Section 2.4.2, we describe graph transformations and the model checker Groove [DDF+06] which act as basis for Dynamic Meta Modeling (DMM), a visual language semantic specification approach which will be described in Section 2.4.3.

2.4.1 Model Checking & CTL, LTL

Model checking is a fully automatic verification technique where a system model is checked against a specification. That is, given a system model M and specification h determine whether the behavior of M meets the specification h [Eme08]. The model M may be given in several representations, e.g., finite automata or Labeled Transition Systems (LTS). The specification is usually given in some logic, e.g., the Computation Tree Logic (CTL) or the Linear Time Logic (LTL). In the following, we will describe labeled transition systems as well as CTL and LTL in detail.

Labeled Transition Systems (LTS)

A Labeled Transition System is similar to finite automata but do not have end states. A Labeled Transition System $\Gamma = (A, S, \delta, I)$ defines a set of action labels A, a set of states S, a set of start states $I \subseteq S$, and a state transition relation $\delta \subseteq S \times A \times S$. In software engineering, the states often represent concrete system states that are described by instances of the created models, e.g., class model, activities, and state charts. The transitions describe all state changes the system may perform. Thus, the LTS describes all reachable system states. See Figure 2.27 for a sketch of a Labeled Transition System. The states $S0$ and $S1$ show the current system state. The transitions reflect system progress or,

e.g., system adaptation. In real labeled transition systems, the transition labels (here *progress*) differ to reflect the actual action that takes one state to another. The figure sketches UML models within states *S0* and *S1*, that is, a state reflects a particular system state whose change is reflected by outgoing transitions.

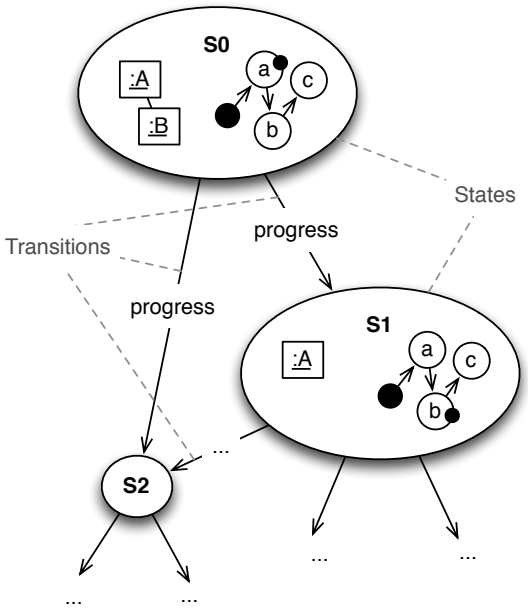


Figure 2.27. Sketched Labeled Transition System

An infinite trace $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ of Γ is defined as $(s_1, a_i, s_{i+1}) \in \delta$ for all $i \geq 0$ and $s_0 \in I$. A finite trace is a prefix of an infinite trace.

Computation Tree Logic (CTL)

The Computation Tree Logic (CTL) is a temporal logic that allows the specification of properties for specific system states as well as the change of these properties in traces. The CTL is typically used with model checkers that check whether a specific model, e.g. an LTS, meets the specific safety or liveness properties which are specified using the CTL. *Safety properties* assure that if at a particular state a starting condition is met, then on all possible future infinite traces some undesired condition never meets. *Liveness properties* assure that in all traces with an arbitrary start state a particular condition meets at some point in time.

safety and liveness properties

The Computation Tree Logic (CTL) consists of proposition logical formulas and temporal operators and is syntactically defined as follows:

$p \in P$	atomic logical formulas
$\neg\phi, \phi \wedge \psi, \ldots$	proposition logic operators
EX ϕ	on at least one **next** state holds ϕ
AX ϕ	on all **next** states holds ϕ
EF ϕ	on at least one outgoing trace holds ϕ at least once (**finally**)
AF ϕ	on all outgoing traces holds ϕ at least once (**finally**)
EG ϕ	on at least one outgoing trace holds ϕ always (**globally**)
AG ϕ	on all outgoing traces holds ϕ always (**globally**)
ϕ **EU** ψ	on at least one outgoing trace holds ϕ **until** ψ holds finally
ϕ **AU** ψ	on all outgoing traces holds ϕ **until** ψ holds finally

CTL formulas are usually applied to Kripke Structures, basically finite automata with a labeling function that labels states with atomic propositions. It is possible to apply CTL formulas to Labeled Transition Systems while the atomic logical formulas are matched with the transition labels instead of state labels. Therefore, in a preprocessing step, the transition labels are move to the preceding states as shown in Figure 2.28. The resulting meaning is that in state $S0$, transitions $a1$ and $a2$ can be taken.

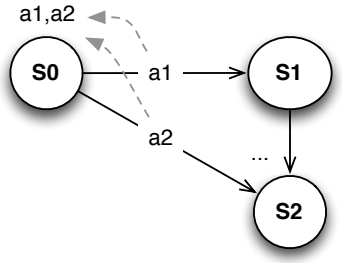


FIGURE 2.28.
Translate Labeled Transition System into Kripke Structure

Linear Time Logic (LTL)

In contrast to the CTL that makes statements about trees of system states, the Linear Time Logic (LTL) makes time related statements about single traces within a system model. Formulas of the LTL are defined as follows:

$p \in P$	atomic logical formulas
$\neg\phi, \phi \wedge \psi, \ldots$	proposition logic operators
X ϕ	ϕ holds in the **next** state
F ϕ	ϕ holds at some state in the future (**finally**)
G ϕ	ϕ holds at all state in the future (**globally**)
ϕ **U** ψ	ϕ holds **until** ψ holds finally
ϕ **W** ψ	ϕ holds globally **unless** ψ holds finally

The LTL does not have quantifiers over paths. All formulas have an implicit preceding "**A**" quantifier.

Both, CTL and LTL formulas can be used to describe common liveness and safety properties that must be fulfilled by a software system. The sets of properties that can be described by the LTL and the CTL overlap, but are not equal. To be able to describe the complete range of properties in our approach, we use both the LTL and the CTL. Temporal logic formulas can be checked using a model checker that gets as input a labeled transition system (or Kripke Structure) and a concrete formula and delivers a counter example if the LTS does not fulfill the formula.

To build up a labeled transition system, graph transformations can be used. Since this technique will be used within this thesis, it is described in the next section.

2.4.2 Graph Transformations & Groove

Graphs are often used for verification in software engineering. This is because graphs have the advantage of a visual representation while still having a formal and powerful mathematical foundation. UML models, which are heavily used in software engineering, can be understood as highly structured graphs. Thus it is almost obvious that the transformation of UML modeled systems, e.g. their progress or their adaptation, can be expressed using graph transformation systems.

Graph transformation systems or graph rewriting systems are used to transform (visual) graphs with (visual) transformation rules [Roz97]. Graph transformation rules r have a left hand side (LHS) and a right hand side (RHS) graph pattern: $r : LHS \rightarrow RHS$. The LHS pattern is matched in the graph G being transformed and replaced by the RHS pattern in place: $G \overset{r}{\rightarrow} H$. The resulting new graph is H. Usually, more powerful graph transformation systems operate on typed, attributed, labeled graphs. That is, an additional logic is needed to compute target attribute values such as element names or attribute values.

Groove [Ren03] is a graph transformation tool that includes a model checker that allows the use of LTL and CTL formulas. Groove takes as input an initial start graph and a set of graph transformation rules to explore the complete state space. The state space (given as labeled transition system) represents all possible variations of the start graph which are reachable by applying the

graph transformation rules repeatedly to the start graph. To transform a UML model and its semantics to graphs and transformation rules that can be read by Groove, the Dynamic Meta Modeling (DMM) approach can be used [Sol13]. In fact, most importantly, DMM allows to formally define the UML's semantics using the meta modeling approach that has been used to define the UML itself. This will be described in the next section.

2.4.3 Dynamic Meta Modeling: Language Semantics Specification

There are several approaches to formally specify the semantics for a modeling language [CKTW08, BCGR09]. In the following, we will focus on Dynamic Meta Modeling (DMM) [Hau05, Sol13] that uses graph transformations to formally specify the execution semantics for meta model based languages. DMM is a semantics specification technique which is not only formal, but also (relatively) easily learnable and understandable due to its visual, object-oriented syntax: DMM rules are basically annotated object diagrams. DMM builds directly on the language's meta model or an extension of it.

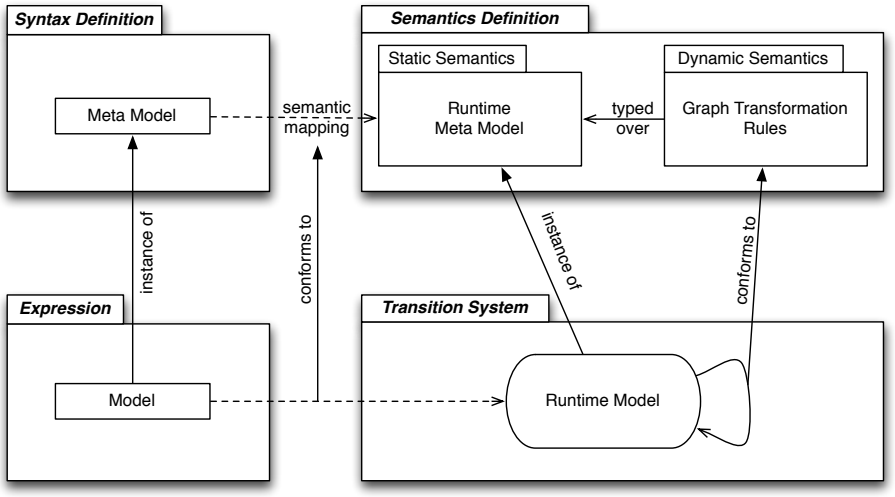


FIGURE 2.29. Overview of DMM [Hau05]

DMM tries to achieve maximum understandability partly by reusing object-oriented concepts, which are expected to be well-known by the target language users. The DMM approach shown in Figure 2.29 consists of three major parts: the runtime meta model, graph transformation rules, and the transition system. The runtime meta model is an extension of a language's meta model with the purpose to explicitly include runtime elements as well as helper constructs.

An example for a runtime element is a *Token* that marks an active action within an activity diagram. The elements added for the sake of describing runtime information are described on the MOF class level and then are instantiated (i.e., within state graphs and rules).

The runtime meta model types the set of graph transformation rules that describe how an instance of the meta model changes through time. For instance, for activity diagrams, a particular graph transformation rule describes under which conditions a token moves from a source to a target action. For this, the instances are mapped to *typed graphs*, i. e. graphs whose nodes and edges are typed over classes and associations of the meta model. The operational rules are then defined as *typed graph transformation rules*, working on the derived typed graphs.

Given the complete set of DMM rules and an instance model, we are able to compute a labeled transition system (LTS). The resulting labeled transition system can be used with an appropriate model checker (DMM suggests the use of Groove [Ren03]) to prove properties that are expressed using temporal logic formulas. The LTS has concrete model instances as states exposing concrete model objects and assigned attributes. The transitions correspond to the applied graph transformation rules (i.e. DMM rules). The LTS is computed by using the state corresponding to the model as the initial state. On that state, all matching rules are applied, leading to new states. This is done until no new states are found and the complete LTS is computed. The transition system can then be analyzed using model checking techniques [ESW07].

Thus, the DMM approach does not only allow the formal specification of a meta model based language, but also provides the infrastructure to use the formal semantics specification for proving certain properties. In [SE09], the authors show how to use the DMM approach and its model checking capabilities to test semantics specifications for correctness.

3

Modeling of Self-Adaptive Systems

"Intelligence is the ability to adapt to change."

– *Stephen Hawking*

3

This chapter describes our approach to the modeling of self-adaptive systems. That is, in Section 3.1, we describe the overall language engineering approach that we have taken. Next, in Section 3.2 we define formally how we perceive adaptivity and what we need to express adaptivity. The resulting requirements are discussed and compared to existing languages. In Section 3.3 the core concepts of our modeling language approach named Adapt Case Modeling Language (ACML) are introduced. Section 3.4 concludes this chapter.

3.1 LANGUAGE ENGINEERING APPROACH

The language that is presented in this chapter was systematically derived from several information sources. Besides the formative evaluations that are described in Chapter 6 (i.e., evaluations that were performed throughout the language engineering process and that highly influenced the language's design), the main information sources where the high-level language requirements that have been introduced in Section 1.2 and characteristics of the target concern *self-adaptation* as presented in Section 2.2. The approach used for language engineering is depicted in Figure 3.1.

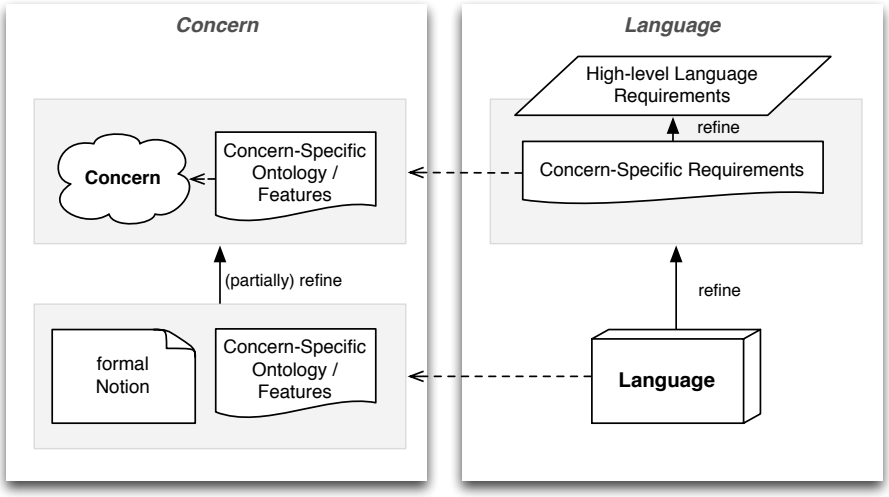


Figure 3.1. Language Engineering Approach

The figure shows the two main pillars of the approach, the *concern* and the *language* itself.

The concern pillar describes the characteristics of a particular concern, e.g., by using definitions, ontologies, or feature trees. In this thesis, we presented an ontology in Section 2.2.1 that shows the concern's different facets. Contents of the ontology include the distinction of reactive and proactive adaptation, or the definition of type vs. instance adaptation. As shown in the figure's bottom left, we refined the concern definitions by formally defining the concern's core elements using a formal notion (see Section 3.2.2). This formal notion helps to develop a clear understanding of the concern on a rather abstract level. The concern-specific ontology keeps compatible to the formal notion.

The language pillar shows the high-level requirements on its top. The high-level requirements defined in this thesis are *Separation of Concerns*, *Analyzability*, *Integration*, *Intuitiveness*, and *Genericity* as described in Section 1.2. Based on the ontology that is defined in the concern pillar, the high-level requirements were refined to concern-specific requirements as shown in Section 3.2.3. Finally, a language was defined that fulfills these requirements, refines the formal notion, and implements most of the features described in the ontology. The language is described in Section 3.3 together with a description of which features have been implemented and which not.

3.2 ANALYSIS

In the following, we analyze the modeling of self-adaptive software systems. That is, we describe where it is needed in the development cycle (Section 3.2.1), what exactly is needed here (Section 3.2.2), i.e. we formally define a notion of self-adaptivity to precisely state what we understand under the term, and finally, we infer requirements for the modeling language that can be addressed e.g. by a meta-model based language (Section 3.2.3).

3.2.1 ADAPTIVITY IN THE DEVELOPMENT CYCLE

While, current research puts effort into supporting single phases such as requirements engineering and implementation approaches for self-adaptive systems, the design phase, especially the early design, is rather neglected concerning modeling language support and techniques for early quality assurance. Put into the context of the OMG's MDA, currently, the Computation Independent Model (CIM) and the source code are covered sufficiently (see Figure 3.2). In an MDA sense, the Platform Independent Model (PIM) and the Platform Specific Model (PSM) are not yet covered sufficiently with engineering methods (i.e., supporting languages, engineering processes, assurance techniques, etc.) as motivated in the following.

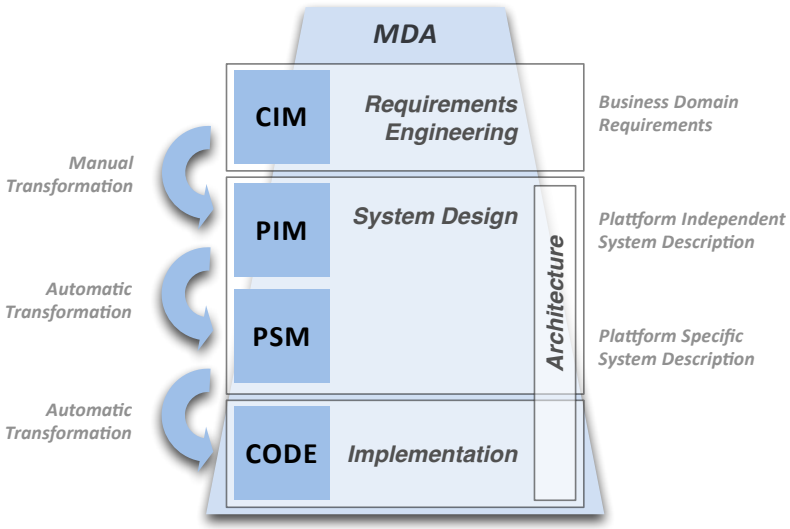


FIGURE 3.2. Model-driven Architecture [Obj03]

The *computation independent model* (CIM) describes the business setting of a system, i. e. its environment and its requirements. Structure and processing are undefined or hidden at this level of abstraction [Obj03]. On this level, self-adaptive systems are usually described using goal-oriented modeling [GSB+08]. Another approach is using controlled natural language [WSB+09] which has been integrated with goal-based modeling in [CSBW09]. A textual requirements approach that explicitly supports the translation of requirements into feedback loops is called the *Awareness Requirements* approach [SSLRM11]. Considering these approaches, the computation independent model (i.e., the requirements level) is sufficiently covered with adaptation-specific approaches.

COMPUTATION INDEPENDENT MODEL

The *platform independent model* (PIM) describes the system's structure and operations abstracting from a concrete technical platform. That is, the model contains the information that is independent from a specific platform. At this level, the OMG suggests to use general purpose modeling languages (e. g. the UML) or domain-specific languages [Obj03]. The *platform specific model* (PSM) enriches the platform independent model with information needed to use the system on a specific platform. This can be operating system specific information, information for a specific database, or the like.

PLATFORM INDEPENDENT AND PLATFORM SPECIFIC MODEL

Regarding modeling language support, the platform independent model and the platform specific model are hard to distinguish since usually the same modeling language can be used for both levels. Therefore, we refer to both as *system design*. For self-adaptive software systems, there are some approaches that are targeted at system design modeling of self-adaptive systems [HGB10, FS09]. Some of these even can be translated automatically into code. However, as shown in Section 3.2.3, they do not fulfill all of our requirements, e.g., they are not integrated, for instance, with the UML, there is no quality assurance approach, or even no engineering process support.

The last step of the MDA process is the transformation of a PSM into code. This is usually a semi-automatic process. To ease transformation and maintenance of the generated code, the coding language may explicitly support self-adaptation, as well. Approaches that explicitly consider self-adaptation at code level include context-oriented programming [SGP12] and frameworks such as StarMX [AST09] and Rainbow [GCH+04]. From our perspective, in an engineering process for self-adaptive software systems, the code level is sufficiently covered with concern-specific approaches, as well.

CODE LEVEL

The MDA approach suggests to create a system architecture as early as possible and maintain it throughout the MDA process. There are several well-

ARCHITECTURES

accepted architectures available for self-adaptive systems. *Generic architectures* for self-adaptive systems such as MAPE-K [KC03] or the three layer architecture [KM07] are first considered on platform-independent level. They can be used as blueprints for creating a good system architecture, however, either they come with no modeling language support, or they do not provide corresponding assurance techniques. Modeling languages for self-adaptive software systems used in the MDA process should support representing these generic architectures, e.g., by including architecture specific concerns as language elements (e.g., sensors, effectors, ...).

All in all, to the best of our knowledge, for the system design in particular, and for the whole engineering cycle in general, there is no continuous engineering method for self-adaptive software systems. Especially the specification of self-adaptivity during early system design combined with a corresponding quality assurance approach is hardly addressed in current research approaches. Therefore, we aim at handling self-adaptivity as a separated concern throughout the complete development cycle, paired with quality assurance approaches allowing to find and remedy errors as early as possible.

SELF-ADAPTIVITY AS SEPARATED CONCERN

But why defining *self-adaptivity* as a separated concern? Just like security, usability, etc., adaptivity is a software aspect that increases in importance, especially because the requirements for software systems increase in complexity. Software systems shall be flexible, dependable, etc. and must cope with various different contexts. Thus, to cope with this increasing complexity during software engineering, like security, usability, etc. there must be a separated adaptivity engineering. The benefits of defining self-adaptivity as a separated concern include a better understanding of modeled adaptivity since the model's focus emphasizes this particular aspect. Further, due to separation of this concern, the models can be quality checked on their own, allowing a distinguished quality analysis of the adaptivity concern of a software system. This is especially important, since self-adaptive software systems come with *new* requirements such as stability which should be assured as early as possible in the engineering process.

WHY ANOTHER LANGUAGE?

However, why defining yet *another* modeling language for self-adaptivity? Current standard modeling languages such as the UML do not explicitly support the concern of adaptivity. Again compared to other disciplines, this leads to shortcomings, e.g., regarding the expressiveness. That is why for instance security engineering invented security cases (UMLSec [Jö02]), usability engineering introduced user stories, and systems engineering proposed the SysML [Obj10a]. In all cases, the standard languages such as the UML did not suffice and were extended to support the concern-specific, or aspect-

specific, modeling. Therefore, our language, the Adapt Case Modeling Language (ACML), is an extension to the standard language UML to introduce the concern of adaptivity.

In the following section, we will precisely describe our understanding of adaptivity that will be used in the remainder of the thesis. The formal representation does not only precisely define our understanding of self-adaptive software systems, but also helps in defining a corresponding language meta-model that is presented in Section 3.3.2. Finally, the formal representation will be used in Chapter 4 to define the properties of self-adaptive systems that may be checked given a suitable quality assurance approach.

3.2.2 Notion of Adaptivity

> Self-adaptation is "the capability of the system to adjust its behavior in response to its perception of the environment and the system itself" [CLG+09].

This definition of adaptivity that conforms to Definition 1 on Page 25 mentions four important aspects of adaptivity: the **system**, the **environment**, **adjusting**, and **response to its perception**. More precisely, to model system adaptivity, we need to model how the system manipulates its behavior—which again is described by models—in the light of some event that occurs in the system itself or its environment. To actually be able to perceive its own behavior and its environment, the system must exhibit the capability of self-awareness and context-awareness. That is, the system must be able to observe itself and its environment which is known as monitoring capabilities. Note that in the following, we use the terms context and environment synonymously.

OUR PRECISE DEFINITION OF ADAPTIVITY

In the following, we will provide a formal description of our notion of self-adaptivity in terms of an algebra. An algebra defines a set of valid terms using a signature that in turn consists of a set of sorts, operations, and reducing or rewriting rules. Operations operate on terms and members of the sorts. The reducing and rewriting rules describe equality and change of terms. The algebra that we will present in the following allows the description of a self-adaptive system in an object-based manner on instance level. *Classes* as known from object-oriented modeling are not expressible in our algebra. Moreover, the algebra simplifies and abstracts from a complete formal model for self-adaptive systems to maintain comprehension. The reason for providing this algebra

is to precisely describe the structure (sorts and operations) and depict the semantics (rewriting rules) of self-adaptive systems. In later sections within this and subsequent chapters, the given algebra will be implemented and detailed using meta modeling and graph transformation rules.

For the description of the algebra, we use *maude syntax*. Maude [CDE+03] is a reflective programming language that supports both equational and rewriting logic specification. Maude uses *modules* to define a collection of sets, operations, and their interactions, i.e., an *algebra*. Further, a module provides the information necessary for reducing and rewriting terms specified by the algebra. The module that we will describe in the following is constructed as follows.

DEFINITION 3.1
Maude Module for Self-Adaptive Systems

```
mod SAS is
  protecting STRING .
  protecting INT .
  protecting QID .
  protecting CONFIGURATION .
  ...
endm
```

The module (**mod**) is named SAS. Using the keyword **protecting**, we are importing four auxiliary algebras from standard maude being STRING, INT, QID, and CONFIGURATION. Strings and ints are used for names and values, QID provides quoted identifiers starting with a tick mark (e.g., 'ABC), and CONFIGURATION is a module that provides object-based specification allowing for an object-specific type of rewriting (see [CDE+03] for more information). The following algebra can be understood without knowing the CONFIGURATION algebra.

To describe which parts of the environment and the system have to be monitored, we first need to describe the system *S* and the environment *ENV* themselves.

In practice, not every piece of information about the system and the environment is of interest for the concern of self-adaptation. Especially, many information which is given in common system models (component models, class diagrams, etc) is not necessary for the description of adaptation. On the other hand, some information which is important for the description of adaptation is usually not captured in common system models. Examples include the description of sensors and aggregated sensor values. Thus, to describe adaptation we need an adaptation-specific view onto the system and the environment. This view is coined *adaptation view model*. In our algebra, we define the adaptation view model *AVM* as follows:

```
sorts System Environment AVM .
op AVM : System Environment → AVM [ctor] .
```

Definition 3.2
Adaptation View Model *AVM*

Using the keyword **sorts**, we define several new sorts divided by blanks. The keyword **op** is used to define operations over these sorts with the given signature. The information given in the brackets further characterizes the operation, e.g., ctor means *constructor*, comm means *commutative*, assoc means associative, and the like. The constructor *AVM* takes as input a system description and an environment description and produces a result of sort **AVM**.

As shown in Definition 3.3, we understand a system *S* to consist of a structural description *ST* and a behavioral description *B* that operates on the structural description.

Definition 3.3
System *S* (*ST*, *B*)

```
sorts Structure Behavior .
op S : Structure Behavior → System [ctor] .

sorts SysComp . *** System Component
subsorts SysComp < Object .
op <_||_,_> : Oid NameAttr ValueAttr → SysComp [ctor object] .
op ST : SysComp → Structure [ctor] .
op B : Action → Behavior [ctor] .
```

The structural description *ST* consists of a set of objects of sort **SysComp** (i.e., system component). Such an object has a unique identifier `Oid`, a name attribute and a value attribute. The object's constructor syntax `<_||_,_>` contains three holes indicated by the underscore `_`. The holes may carry values of the respective kind that is given after the colon in the same order. The behavioral description *B* consists of a set of actions that operate on the system components (see below). See the following listing (Figure 3.3) for an example.

```
S (
  ST ( < 's-001 || Name : "Comp-A", Value : 10 > ),
  B ( ACT ( 'a, 's-001, 1 ) ACT ( 'b, 's-001, 2 ) ACT ( 'c, 's-001, 3 ) )
)
```

Figure 3.3.
Example System Definition

The system's structure description contains a single system component with id `'s-001`, name `"Comp-A"`, and value 10. The behavior description contains three actions each of which changes the system component's value to 1, 2, and 3, respectively (details about actions are given below). Of course, this is a very simplified system description. For instance, in a more complete formal model,

a system component could have an arbitrary amount of attributes with arbitrary names. However, for our purpose this simplification suffices.

Name and value attributes are defined as follows.

DEFINITION 3.4
Name and Value Attributes

```
sorts NameAttr ValueAttr .
subsorts NameAttr ValueAttr < Attribute .
op Name :_ : String → NameAttr [ctor] .
op Value :_ : Int → ValueAttr [ctor] .
```

These two attributes are now *built-in* attributes in our algebra. Especially, the attributes' names are predefined. In a more complete algebra, an attribute constructor would have two arguments, one of which being the attribute's name, the other being the attribute's value of arbitrary kind.

The next definition shows system behavior. The system's behavioral description *B* may be described by any available auxiliary algebra that orchestrates a set of actions *ACT*. Actions are defined as follows.

DEFINITION 3.5
Actions *ACT* used as System Behavior *B*

```
sorts Action Aid .
subsorts GID < Aid < Oid .
op ACT : Aid Oid Int → Action [ctor] .
op none : → Action [ctor] .
op __ : Action Action → Action [ctor config assoc comm id: none] .
```

The actions defined here simply may change a value of system components. An action has an id `Aid`, the id of the target system component `Oid` and the new value that will be written. The last operation allows to specify an arbitrary amount of actions with `none` being the identity element.

Similarly to the system, the environment *ENV* consists of a structure description as shown in Definition 3.6. In addition, the environment may send events *EV*.

DEFINITION 3.6
Environment *ENV*

```
sorts EnvStructure Events .
op ENV : EnvStructure Events → Environment [ctor] .

sorts EnvComp .
subsorts EnvComp < Object .
op <_ | | _,_> : Oid NameAttr OpenAttr → EnvComp [ctor object] .
op ENVST : EnvComp → EnvStructure [ctor] .
```

The environment's structure is given by a set of environment components. They are very similar to system components but have an open attribute instead

of a value (Definition 3.7).

Definition 3.7
Open Attributes

```
sorts OpenAttr .
subsorts OpenAttr < Attribute .
op Open :_ _ : Int Range → OpenAttr .
```

The open attribute has a value, and additionally, a range for its value. Open attributes may change in the given range. This change may be simulated as shown with the following two rewriting rules.

Definition 3.8
Rewriting Rules for Open Attributes

```
crl [openAttrIncr] : Open : v [ l ; u ] ⇒ Open : v + 1 [ l ; u ] if v < u .
crl [openAttrDecr] : Open : v [ l ; u ] ⇒ Open : v − 1 [ l ; u ] if v > l .
```

A rewriting rule is defined using the keyword **rl** for rule or **crl** for conditioned rule. The term on the left-hand side (before ⇒) is matched and replaced by the term on the right-hand side. The first rule increases an open value by one of the value does not exceed the upper boundary. The second rule decreases an open value, accordingly. Defining a range for open attributes might abstract from reality but is necessary for avoiding infinite system state spaces.

The following listing shows the definition of events. Events may be of different type `Signal`, `Change`, or `Time`. A signal event may be thrown by any environment component. A change event occurs whenever a particular variable in the environment changes, and a time event occurs after a predefined amount of time. The specific creation of events is not defined in our algebra. Instead, an event of any type has a unique id, the type, a name, and a boolean that indicates whether the event is active or not.

Definition 3.9
Events *EV*

```
sorts Event EtId .
subsorts Event < Object .
subsorts EtId < Cid .
ops Signal Change Time : → EtId [ctor] .
op <_:_ | _ | _> : Oid EtId NameAttr Bool → Event [ctor object] .
op EV : Event → Events [ctor] .
```

If an event is active, the self-adaptive system might react to the event and deactivate it (described by rewriting rules below). The rewriting rule given in Definition 3.10 arbitrarily activates events. Note that events are objects as well. In maude, objects may form arbitrary large sets. That is, two or more objects together again are of sort **Object**. Therefore, the operation *EV* given in Definition 3.9 may be passed an arbitrary amount of events.

DEFINITION 3.10
Rewriting Rule for Creating Events

```
rl [createEvent] :
    < evid : et | Name : t | false > => < evid : et | Name : t | true > .
```

Now, that we have described the adaptation view model *AVM*, we will proceed with the adaptation rules that are described using a model named *adapt case model* (*ACM*). As shown in the following definition, an *ACM* consists of a set of adaptation rules *AR*.

DEFINITION 3.11
Adaptation Rule *AR*

```
sorts ACM AR .
op ACM : AR -> ACM [ctor] .

sorts MonitoringActivity AdaptationActivity .
subsorts Action < AdaptationActivity .
op AR : MonitoringActivity AdaptationActivity -> AR [ctor] .
op none : -> AR [ctor] .
op __ : AR AR -> AR [ctor config assoc comm id: none] .
```

An adaptation rule in turn consists of a monitoring activity and an adaptation activity. An adaptation activity is a super sort of **Action**. Hence, an adaptation activity is an arbitrary large set of actions.

DEFINITION 3.12
Monitoring Activity *MON*

```
op MON : Corridor Aid -> MonitoringActivity [ctor] .
op MON : EtId String Aid -> MonitoringActivity [ctor] . *** String = event name

sorts Corridor .
op CORR : Oid Range -> Corridor [ctor] . *** monitors attribute "Value"
op none : -> Corridor [ctor] .
op __ : Corridor Corridor -> Corridor [ctor config assoc comm id: none] .

sorts Range .
op [_;_] : Int Int -> Range [ctor] .

msg trigger_ : Aid -> Msg .
op noMsg : -> Msg .
```

As shown in Definition 3.12, a monitoring activity may have two different forms. First, it may consist of a corridor definition *CORR* to monitor a specific value of an environment component or a system component. The corridor specifies the unique id of the system or environment component that is monitored and the range in which its value is allowed to reside. A range is given by a lower and an upper bound. Second, the monitoring activity may consists of

a three tuple describing an event that is monitored by defining the event's type and name. In both cases, the monitor contains the action's id that is triggered if the monitor observed negatively, i..e., the specified event is active, or the observed component's value is outside the specified range. The triggering of the adaptation activity is done by the `trigger` message (**msg**) that carries the id of the triggered action.

Finally, adaptation view model *AVM* and adapt case model *ACM* together with a set of message form the *ACML* model.

Definition 3.13
Adapt Case Modeling Language Model *ACML*

```
sorts ACML .
op ACML : AVM ACM Msg → ACML [ctor] .
```

After having defined the complete *ACML* model, we can define the rewriting rules, i.e., the dynamic semantics for adaptation in our algebra. The following definition shows the rewriting rule `monitorSystem`. Most of the rule is used for pattern matching, i.e., there must be at least one system component and a monitoring activity that monitors this system component defined with a corridor. If the system component's value is lower than the corridor's lower bound or higher than its upper bound and if the corresponding adaptation activity has not yet been triggered, a triggering message (`trigger aid`) is appended to the end of the term.

Definition 3.14
Rewriting Rule for Monitoring the System

```
crl [monitorSystem] :
      ACML (
          AVM ( S ( ST ( c1 < sid || n , Value : v > ), b ), env ),
          ACM ( ar AR ( MON ( cor CORR ( sid, [ l ; u ] ), aid ), a1 ) ), m )
   ⇒ ACML (
          AVM ( S ( ST ( c1 < sid || n , Value : v > ), b ), env ),
          ACM ( ar AR ( MON ( cor CORR ( sid, [ l ; u ] ), aid ), a1 ) ),
                m trigger aid )
    if v < l or v > u and not trigger aid in m .
```

Maude uses this and similar rules to construct a transition system. Applying this rule to a term in ACML yields in another term in ACML. These terms are considered as states of the transition system, the applications of rewriting rules are considered as transitions labeled with the rules' name.

The following two rules are very similar. The `monitorEnvironment` rule (Definition 3.15) monitors an environment component instead of a system component. Other than that, the rule is identical to the `monitorSystem` rule.

The `monitorEvents` rule (Definition 3.16) is applied if an event is activated

DEFINITION 3.15
Rewriting Rule for Monitoring the Environment

```
crl [monitorEnvironment] :
      ACML (
          AVM ( sys, ENV ( ENVST ( c1 < envid | | n , Open : v r > ), es ) ),
          ACM ( ar AR ( MON ( cor CORR ( envid, [ l ; u ] ), aid ), a1 )), m )
    ⇒ ACML (
          AVM ( sys, ENV ( ENVST ( c1 < envid | | n , Open : v r > ), es) ),
          ACM ( ar AR ( MON ( cor CORR ( envid, [ l ; u ] ), aid ), a1 ) ),
                m trigger aid )
     if v < l or v > u and not trigger aid in m .
```

that is currently monitored as well. In this case, the event is deactivated (set to `false`) and a triggering message is added to the message heap.

DEFINITION 3.16
Rewriting Rule for Monitoring Events

```
crl [monitorEvents] :
      ACML (
          AVM ( sys, ENV (envst, EV ( < evid : et | Name : t | true >) ) ),
          ACM ( ar AR ( MON ( et, t, aid ), a1 ) ), m )
    ⇒ ACML (
          AVM ( sys, ENV (envst, EV ( < evid : et | Name : t | false > ) ) ),
          ACM ( ar AR ( MON ( et, t, aid ), a1 ) ),
                m trigger aid )
     if not trigger aid in m .
```

Finally, the last rewriting rule executes the adaptation activity by applying the contained action. If a triggering message for the action exists, the rule applies and sets the system component's value to the action's value.

DEFINITION 3.17
Rewriting Rule for Adapting a System s

```
rl [adaptSystem] :
      ACML (
          AVM ( S ( ST ( c1 < sid | | Name : t, Value : v > ), b), env ),
          ACM ( ar AR ( mon, ACT ( aid, sid, c ) ) ), m trigger aid )
   ⇒ ACML (
          AVM ( S ( ST ( c1 < sid | | Name : t, Value : c > ), b ), env ),
          ACM ( ar AR ( mon, ACT ( aid, sid, c ) ) ), m ) .
```

A similar rule exists for adapting environment components.

The listing in Figure 3.4 shows an example definition of a self-adaptive system using our algebra. The adaptation view model *AVM* defines a system with a single component and an environment with a single environment component. The environment component has an open value that ranges from 1 to 5. Further, the environment defines a signal event named "`CompNA`" that is cur-

rently activated. The adapt case model *ACM* defines three adaptation rules. The first monitors the system component, the second monitors the environment component and the third monitors a signal event named "`CompNA`". The correspondingly triggered adaptation actions adapt the value of system component `'s-001`.

```
ACML (
  AVM (
    S (
      ST ( < 's-001 | | Name : "Comp-A", Value : 10 > ),
      B ( ACT ( 'a, 's-001, 1 ) ACT ( 'b, 's-001, 2 ) ACT ( 'c, 's-001, 3 ) )
    ) ,
    ENV (
      ENVST ( < 'e-001 | | Name : "Extern", Open : 4 [ 1 ; 5 ] > ),
      EV ( < 'ev-001 : Signal | Name : "CompNA" | true > )
    )
  ),
  ACM (
    AR (
      MON ( CORR ( 's-001, [ 1 ; 8 ] ), 'a-001 ),
      ACT ( 'a-001, 's-001, 5 )
    )
    AR (
      MON ( CORR ( 'e-001, [ 1 ; 4 ] ), 'a-002 ),
      ACT ( 'a-002, 's-001, 6 )
    )
    AR (
      MON ( Signal, "CompNA", 'a-003 ),
      ACT ( 'a-003, 's-001, 12 )
    )
  ),
  noMsg
) .
```

Figure 3.4. Complete Example of a Self-Adaptive System defined in our Algebra

It already has been noted that this algebra by far does not express all the necessary concepts that we need for modeling self-adaptive systems. However, most missing concepts are specializations of the concepts included in the algebra. Thus, the presented algebra is a sufficient abstraction to precisely describe our notion of self-adaptivity, i.e., the constituents of the adaptation view model, the rule descriptions in the adapt case model, and their interrelation in terms of rewriting rules.

Type Adaptation In Section 2.2, we motivated the need for adaptation on type level. Since the provided algebra is meant to only precisely describe our notion of self-adaptive systems, we do not go into detail of formally defining type adaptation here. However, this could easily be included into our algebra. For each object, a class has to be added. Further, we need to add constructors for classes, and finally, we need to add *adaptation* actions that may change class

definitions and propagate changes to objects as well. In the course of this thesis, we will add type adaptation in the algebra's more detailed version using meta models and graph transformations.

Relation to MAPE-K Based on our notion of self-adaptivity, we slightly customize and detail the MAPE-K reference model that has been introduced in Section 1. See Figure 3.5 for details. We divide the adaptation rule into two partitions, the monitor and adaptation parts. Further, we introduce the Adaptation View Model that is a view onto the system and its environment. The Adaptation View Model describes both the structure and behavior of system and environment. The second major component of the Adaptation View Model is the Knowledge that may define corridors, aggregations, constraints, invariants, histories, etc. The Adaptation View Model also defines sensors and effectors (dashed arrows labeled with S and E) that allow the systematic definition of access points for the system and the environment.

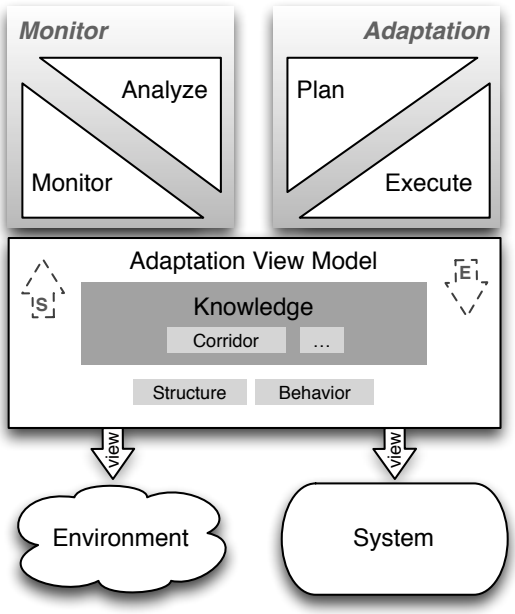


Figure 3.5. Customized MAPE-K Model

Based on our notion of self-adaptivity, we will identify requirements and analyze existing modeling languages for self-adaptive systems.

3.2.3 Requirements & Related Work

We divide the requirements for a modeling language into two sets. The first set of requirements describes the required language principles and the language structure that can be inferred from the high-level requirements from Section 1.2 and the formal notion of self-adaptivity as described in Section 3.2.2. These requirements constrain the basic language design, e.g. used principles and first-class elements, but do not constrain specific modeling elements or the language's expressiveness. We use this set of requirements to compare different approaches to each other and our ACML. On the other hand, the second set of requirements describes the required language features, i.e. requirements on the language's scope and expressiveness. The language features describe the language's scope and expressiveness. For instance, a feature describes whether or not the language supports the description of adaptation on type level. Further, language features may describe which kinds of adaptation actions are allowed. The language features have been described in an taxonomy in Section 2.2.1. After we have described our language's core elements in Section 3.3.2, we will discuss the language features our language supports in Section 3.3.3.

In the following, we describe the requirements concerning language principles & structure.

Requirements concerning Language Principles & Structure

Based on the high-level requirements (Separation of Concerns, Analyzability, Integrated, Intuitiveness, and Genericity) and the concern description (cf. Section 2.2), we derive requirements for a modeling language for self-adaptive software systems:

Separation of Concerns Dealing with concerns (self-adaptivity, application logic) separately, allows to focus on one particular aspect more closely. For our modeling language, we infer the following requirements:

- MR01: The language must support the separation of self-adaptation logic concerns from core application logic concerns. That is, language elements should include concern-specific means to describe self-adaptivity (e.g. monitoring, adaptation activity specification).
- MR02: The language must contain an *adaptation view model* with structure and behavior description of the system and its environ-

ment that enables *selection* of relevant information and *projection* e. g. to aggregated values.

- MR03: The language must contain an *adaptation rule model* that allows to model the *monitoring* and *adaptation* activities separately from each other while enabling the adaptation on *instance* and *type* level.

The opposite of separation of concerns is composition. That is, each approach that supports the separation of concerns should allow the composition of the created model artifacts with the remaining system specification. That does not necessarily include the merging of these models but at least a the definition of proper relationships between the two models. Therefore, we infer an additional requirement as follows:

- MR04: The language must support the composition of the self-adaptation logic concern models with models of the core application logic concerns.

Analyzability The easiest way to define a language that is semantically unambiguous, is to base the language on a formal model. Besides being most precise, formal models usually allow formal analysis and thus quality assurance. For our modeling language, we infer the following requirements:

- MR05: The language must be based on a formal model that allows for unambiguous semantic specification and, eventually, verification.
- MR06: The language must support the definition of constraints, properties, and other means to allow for formal analysis.

Integrated A method for the specification of self-adaptive software systems should be integrated into existing software engineering methods. State of the art in software engineering is the usage of model-driven approaches. That is, models are first-class entities in the engineering cycle and often transformed and enriched automatically. Therefore, a language should be prepared to be part of a model-driven environment, e.g., by the introduction of a new concern-specific view. For our modeling language, we infer the following requirements:

- MR07: The language must be prepared for model-driven approaches, that is, use a technology that is capable of being integrated into model-driven approaches (e.g. EMF/Ecore).

Additionally, since an overall goal for a new language or language extension is acceptance and usability, the language should follow the state of the art. Considering self-adaptive systems, the state of the art is using control loops as first-class entities. Thus the language should support this paradigm. For our modeling language, we infer the following requirement:

- MR08: The language must have control loops as first-class entities.

Further, relying on standards, increases the acceptance and usability of languages since standard languages, such as the UML, are usually well-known and well tool supported. For our modeling language, we infer the following requirement:

- MR09: The language must be based on standard modeling languages.

Intuitiveness Since the language is used very early in the design phase, the users may be technically untalented. Therefore, a language must target a layman audience. For our modeling language, we infer the following requirement:

- MR10: The language must not be too technical, or technical details can be hidden. The language should be aligned to other laymen-friendly approaches, such as use case specification or business process languages.

Genericity While on the one hand the language should be specific to the particular concern of self-adaptivity, on the other hand it should be generic considering the domain of application. That is, it should be usable for embedded systems as well as business information systems. For our modeling language, we infer the following requirement:

- MR11: The language must not focus on one particular domain (embedded, BIS) but be as generic as possible.

In Table 3.1 the requirements are listed in the columns and used to compare the different approaches listed in the table's rows. For each requirement, in the following we will briefly discuss some possible implementation alternatives as well as the technique that will be used in our approach.

Since the adaptation concern's root phase is the early design, the related work analysis considers approaches that allow for platform-independent and platform-specific design. The phases requirements engineering and imple-

mentation are deliberately left out here. The relation of the ACML to preceding and succeeding phases is discussed in Chapter 5.

TABLE 3.1. Modeling Approaches compared to Requirements

Design Approaches	MR01	MR02	MR03	MR04	MR05	MR06	MR07	MR08	MR09	MR10	MR11
Use Cases [Obj10b]	○	×	×	✓	○	✓	✓	×	✓	✓	✓
Cheng, 2006 [ZC06]	✓	×	○	×	✓	✓	×	×	×	×	✓
Giese, 2007 [Gie07]	✓	×	✓	✓	✓	✓	✓	×	✓	○	×
Fleurey, 2009 [FS09]	✓	○	○	×	✓	✓	✓	○	×	×	○
Hebig, 2010 [HGB10]	✓	✓	×	✓	×	×	✓	✓	✓	✓	✓
Garlan, 2012 [CG12]	✓	×	✓	×	✓	×	○	✓	×	×	✓
Vogel, 2012 [VG12]	✓	×	✓	×	✓	×	✓	✓	×	○	✓
ACML, 2013 [LE13]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

MR01: Separation of Concerns The separation of concerns can be achieved by several different means. For instance, Fleurey et al. [FS09] define a separate independent meta-model that allows the specification of adaptivity related concerns using concern-specific concepts such as variants or context variables. Another approach towards separation of concerns is the extension of an existing language, e.g. by introducing a concern-specific view. As shown in the subsequent section, the ACML uses this approach by introducing two new adaptation-specific diagram types to the UML.

MR02: Adaptation View Model The requirement for a separated adaptation view model refines requirement MR01. An adaptation view model may be achieved, e.g., by decorating an existing system model. The UML [Obj10b] supports this kind of view by the concept of UML Profiles. A UML Profile allows the definition of particular tags for classes and associations, called *stereotypes*. Stereotyped elements may be equipped with further semantics. Hebig et al. [HGB10] make use of the UML Profiling concept to define so-called *strands* which indicate the information and dependency flow between UML interfaces that are used as sensors and effectors. While UML Profiling is known to be a light-weight extension mechanism, the ACML uses a heavy-weight approach to extend the UML directly on meta-model level. Thereby, the ACML provides additional adaptivity-specific concepts such as sensors, effectors, and knowledge that can be used to decorate existing UML diagrams.

MR03: Adaptation Rule Model The requirement for a separated adaptation rule model refines requirement MR01, too. The explicit separation of the adaptation rules not only allows to further focus on a subset of the overall specification, but also enables easy reuse of adaptation rule pattern. Further, the distinction of the monitoring and the adaptation part within a single rule takes into account the current state of the art and under-

standing of self-adaptivity [CLG+09]. It fosters the clear separation of the WHEN (monitor) from the HOW (adaptation). Fleurey et al. [FS09] do not explicitly separate the adaptation rules from the remaining system specification on meta model level, but rather provide a specific tabular adaptation rule view onto the model. The UML does not provide a specific rule view at all. The two approaches by Garlan [CG12] and Vogel [VG12] define separate (textual) languages for the specification of adaptation rules following the MAPE-K pattern. The ACML defines a specific adaptation rule model, named Adapt Case Model (ACM), that is separated and decoupled on meta model as well as on concrete model level using the notion of UML interfaces (sensor and effector interfaces).

MR04: Composition Composition is the opposite of separation of concerns. To be easily usable in an engineering process, a language that separates a specific concern should allow to compose this separated models with other system models. While the UML provides specific relationships, inheritance and other mechanisms to compose models with each other, the language provided by Fleurey et al. [FS09] is rather isolated and cannot be composed with other existing system models out of the box. Same applies to the approach proposed by Cheng et al. [ZC06] unless the other models are given as petri nets. Since the ACML is an extension of the UML, all composition mechanisms of the UML are inherited as well.

MR05: Formal Semantics Definition Approaches such as the one of Fleurey et al. [FS09] and of Cheng et al. [ZC06] rely on formal models such as Alloy [Jac02] or petri nets. Thereby, the semantics of the language are precisely defined. Other approaches rely on graph transformations or even provide an interpreter for their language [VG12] where the formal semantics are given by, e.g., the Java semantics. The UML does not define its semantics formally, but rather rigorous using natural text. The ACML uses the semantics specification language *Dynamic Meta Modeling (DMM)* [Hau05] to formally define its semantics and thus allow for formal analysis.

MR06: Constraints, Properties, and other Means for Analysis The UML uses the Object Constraint Language [RG02] to define constraints that may validate a model as well as properties. Fleurey et al. [FS09] include a small constraints language within their approach that may validate their respective models. Cheng et al. [ZC06] use temporal logic expressions to constraints. The ACML uses a specific language that is provided by the DMM approach and is translated into temporal logic expressions as well. Further, the ACML allows to define invariants and

other constraints that validate a model. Finally, the ACML provides the mechanism of history knowledge that allows to not only reason about the current state but also about past states and predict future states.

MR07: Ready for Model-Driven Engineering The UML is fully implemented by the Eclipse Modeling Framework (EMF) [BBM03] that is used by Fleureys [FS09] and Vogels [VG12] approaches as well. This provides the necessary basis to be used within in common MDE approaches. The ACML is implemented using this framework as well and further allows the translation of models into labeled transition systems (LTS) to allow for further formal analysis using the DMM framework. Of course, even the petri nets used by Cheng et al. [ZC06] may be used for model-driven engineering if suitable transformations are used or defined.

MR08: Control Loops as First-Class Entity While Fleurey et al. [FS09] hide the control loops within their models, Hebig et al. [HGB10] explicitly specify control loops by the use of *Strands* that relate sensor and effector interfaces to each other. The ACML takes this idea even further by implementing the architectural structure that has been proposed with the MAPE-K Feedback Control Loop [Mur04]. The MAPE-K loop explicitly defines a control loop to consist of the four phases *monitor*, *analyze*, *plan*, and *execute* that use *sensors*, *effectors*, as well as a shared *knowledge*. The approaches by Garlan [CG12] and Vogel [VG12] are heavily based on the MAPE-K loop, too.

MR09: Based on Standard Modeling Languages While the approaches presented by Hebig et al. [HGB10] and Giese [Gie07] are based on the UML using the UML Profiling mechanism, the ACML extends the UML in a heavy-weight manner. Both approaches allow the corresponding language to be based on the standard modeling language UML. In contrast, the approach by Fleurey et al. [FS09] does not use any standard modeling notation.

MR10: Laymen-Friendly Whereas the UML is known to be laymen-friendly, and thus are the approach from Hebig et al. and the ACML, the use of petri nets is not suitable for laymen. Even the approach presented by Fleurey et al. [FS09] may be more difficult to be used by laymen since it provides completely new concepts and syntactical representation.

MR11: General-Purpose Language The approach presented by Fleurey et al. [FS09] has been development and presented for the use with robotic systems, i.e. embedded systems. Thus, the language might be less suitable to be used for business information systems that tend to be more

complex since less local. Giese proposes an approach directed towards mechatronic systems [Gie07]. However, basically all approaches may be used for any domain. Languages such as the UML and the ACML explicitly have been defined to be general purpose languages. Thus, no domain-specific elements are included within the language.

In the next section, we will present our concern-specific modeling language that aims at fulfilling all requirements listed above.

3.3 ACML: A Concern-Specific Modeling Language for Self-Adaptive Systems

In this section, we present the Adapt Case Modeling Language (ACML) that allows modeling self-adaptive software systems. In Section 3.3.1, we will describe where in the engineering process for self-adaptive software systems, the ACML can be used. In Section 3.3 we will describe the language's core principles followed by the language's features in Section 3.3.3 and a placement of the ACML within the meta model layers in Section 3.3.4.

3.3.1 ACML in the Engineering Process

As motivated in Chapter 1, to develop self-adaptive systems of high quality, we use constructive as well as analytical methods during the software engineering process.

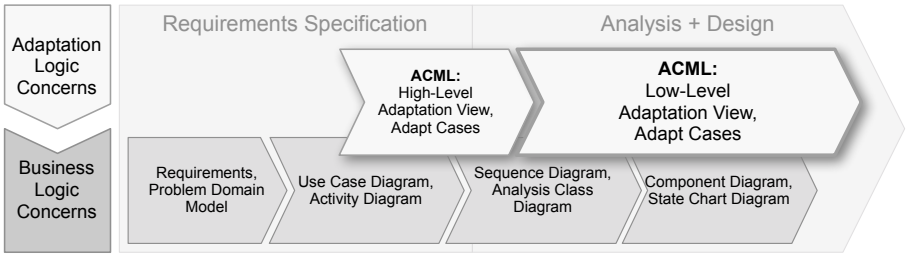


Figure 3.6. Development Process of Self-Adaptive Systems

The constructive methods include modeling the system using concern-specific modeling languages within an MDA [Obj03] process with strong emphasis on

the principle *separation of concerns*. Based on separated models, we then use analytical methods to check the models for high quality.

Figure 3.6 shows 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 life cycle 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, e.g. in the light of component-based development.

The concern of self-adaptation needs to be considered in every engineering phase. That is, beginning with the requirements elicitation, self-adaptation needs to be considered. Currently, self-adaptation is rather neglected in practical software engineering. That is, although some adaptation features are included in today's software, they are not explicitly modeled. To make self-adaptation more explicit, practical approaches need to be developed that allow expressing or preparing adaptivity during requirements engineering, the logical and technical design, and implementation phase.

CONTINUOUS SUPPORT FOR SELF-ADAPTATION NECESSARY

While for requirements, goal-driven modeling approaches are adopted, the models for logical design are hardly user-friendly, i. e. the user is meant to model with computer science techniques such as petri nets, model checking, etc. For the phase of technical design, languages have been developed. Examples include strands [HGB10] and the rainbow component model [GCH+04]. Even later in the implementation phase, dedicated frameworks (e. g. spring) and programming languages (e. g. Context Erlang) are proposed to support the development of self-adaptive systems. In order to support a continuous development process for self-adaptive systems, the aspect of self-adaptive systems needs to be *a*) supported within each engineering phase, and *b*) seamlessly supported throughout the different phases.

For the separate specification of the system's self-adaptation logic (see the top

of Figure 3.6), we propose to use a concern-specific modeling language, called Adapt Case Modeling Language (ACML) that uses the architectural style of control loops (cf. Requirement MR08) and integrates with the UML [Obj10b] (cf. Requirement MR09). The ACML consists of an Adapt Case Model that describes adaptation rules with an extended Activity Diagram, as well as an Adaptation View Model (cf. requirements MR03 and MR02). The Adaptation View Model is a view on the designed software system particularly focusing on aspects that are important for adaptation (cf. Figure 3.7). Examples include the specification of sensors, effectors, and adaptation knowledge (e.g., aggregated values).

ADAPT CASE MODELING LANGUAGE (ACML)

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.

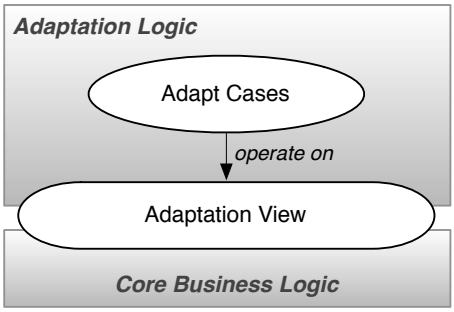


FIGURE 3.7. Adaptation Logic operating on the Adaptation View

The two process actions at the top of Figure 3.6 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.

HIGH-LEVEL ADAPT CASES

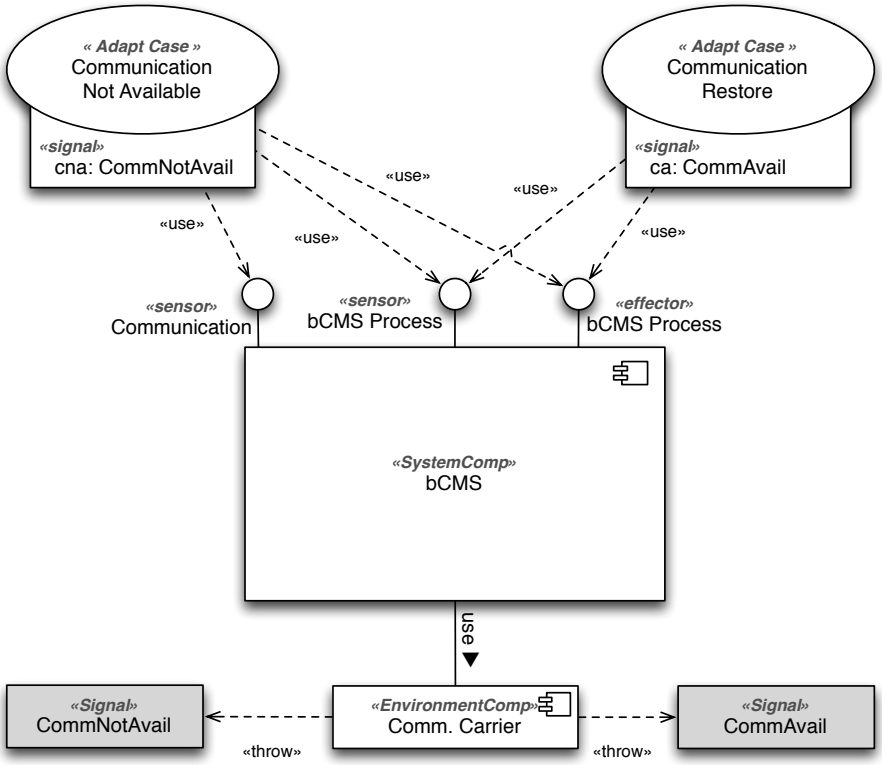


FIGURE 3.8. High-Level ACML Model

Let's consider Figure 3.8 as an example of a high-level ACML model that reflects the scenario from Section 2.1. 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 the bCMS system and a communication carrier which is used for the communication of the two parties FSC and PSC. Further, the model defines three adaptation interfaces, namely two sensors for sensing the system's communication and the bCMS process (state, number of instances, etc.) and one effector for manipulating the bCMS process (pause processes, continue processes, etc.). Further, the model shows two signals that are sent from the environment component, the communication carrier. In turn, the two Adapt Cases are defined to observe and receive these signals. 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.

LOW-LEVEL ADAPT 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 3.6 top right). The monitoring and adaptation rou-

tines are specified using specific UML activities as described in Section 3.3.2. The Adaptation View Model is refined with detailed interface descriptions and knowledge.

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. [Coc00] for use case techniques).

Let us look at the single concepts of the ACML in detail in the next section.

3.3.2 ACML Core Principles and Modeling Concepts

In this section, the core concepts of the Adapt Case Modeling Language are presented, mainly in concrete syntax. All concepts are precisely defined in the appendix with meta modeling techniques where the concrete syntax is picked up again (see Section A.1 and Section A.2). Some of the findings presented in this section are based on master theses [Bec11, Mut12].

STRUCTURAL DESCRIPTION BY ADAPTATION VIEW MODEL

With the Adapt Case Modeling Language, an adaptation specification is divided into two parts, a structural description and a behavioral description. The structural description is a view onto the system and its environment. It describes all adaptation-relevant elements, such as components, interfaces (sensors and effectors), or adaptation operations that actually change the system under consideration. The behavioral description defines the adaptation rules that consists of a monitoring activity and an adaptation activity. Basically, the behavioral description is an orchestration of sensing and effecting adaptation operation calls.

Technically, the structural description, named Adaptation View Model (AVM), is an annotated excerpt of the system model. That is, the AVM selects adaptation-relevant parts (e.g. components) from the system model and adds additional information that is needed for adaptation:

- Adaptation interfaces (e.g. sensors and effectors)
- Aggregated informations (e.g. value computations)
- Constraints, invariants, histories, etc.

As illustrated in Figure 3.9, the system model remains unchanged and is only decorated by the adaptation view model (please note the mapping to the formal notion in the right column of the figure). Of course single information can be moved from the AVM to the system model and vice versa.

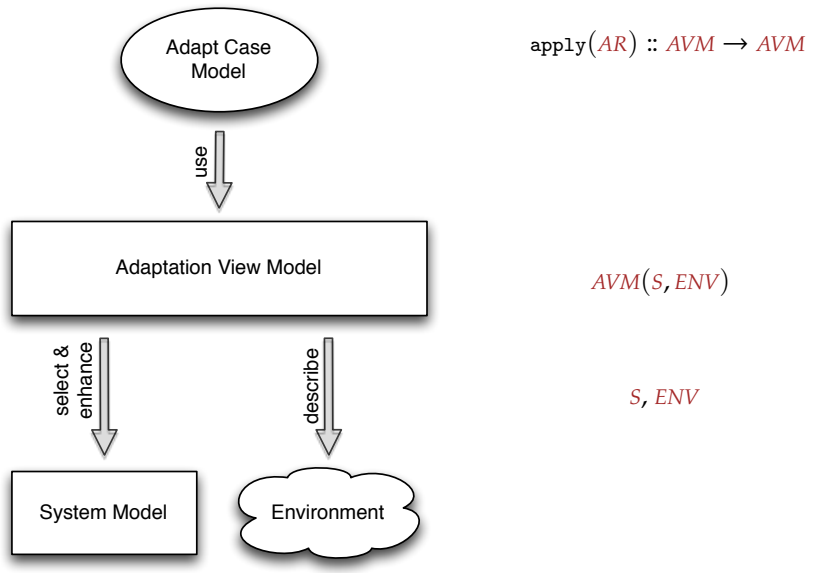


FIGURE 3.9. Adaptation View Model, System Model, and Adapt Case Model

BEHAVIORAL DESCRIPTION BY ADAPT CASE MODEL

The behavioral description, named Adapt Case Model (ACM), is a rule-based description of adaptation behavior. It consists of a definition of what and how to monitor using sensors and knowledge information from the AVM (e.g. histories for pro-active adaptation), a definition of the analysis of gathered data, and a definition of the planing and actual execution of appropriate adaptation actions. The range of adaptation actions that are supported is almost arbitrary since all UML actions can be used, while additionally, specific helper actions further support adaptation-specific behavior (especially for type adaptation).

COMPONENT-BASED MODELING

The AVM relies on component-based modeling, i.e. major functional units are encapsulated in components that expose interfaces for accessing them. By the use of components, application logic that is not relevant for adaptation is hidden or not even specified allowing a clear separation of concerns (cf. Requirement MR01). Further, components and their exposed interfaces naturally provide mechanisms for model composition which allows to precisely relate the adaptation logic concerns to application logic concerns (cf. Requirement MR04). Components may have behavior that orchestrate the components' functionality (see `SystemBehavior` in Figure 3.10). As such, a component is an independent functional unit that exposes a particular structure and behavior. If a component-based system has an orchestrating root behavior, this

behavior may be encapsulated by a specific component with its root task to orchestrate its subcomponents. If no components are used at all, a single component named "system" might be appropriate. Thereby, every system might be described using component-orientation allowing to utilize the paradigm's benefits. In general, every `SystemBehavior` that is specified for a specific component is *adaptable*. `SystemBehavior` is representing the system's core business logic and does not perform any adaptations. The set of system components corresponds to the system description S that consists of a behavioral description B (the `SystemBehavior`) and a structural description ST (the `SystemComponents` and their relations).

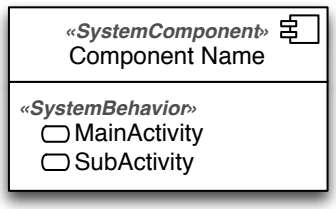


Figure 3.10. SystemComponent with SystemBehaviors

The AVM decorates components by adding additional adaptation behavior (see `AdaptationBehavior` in Figure 3.11). This adaptation behavior defines concrete changes of the component's internals (e.g. properties or actions and flows in `SystemBehavior`, etc...). Changes may be applied to the component's structure or behavior. Adaptation behavior may also define sensing behavior which is needed for adaptation. Usually, the adaptation behavior is very specific to the concrete domain and can (indirectly) be reused by different adaptation rules. `AdaptationBehavior` represents a system's adaptation logic with its main purpose to adapt the systems structure and adaptable behavior. `AdaptationBehavior` contains adaptation actions which are specialized UML actions allowing the manipulation of type and instance models.

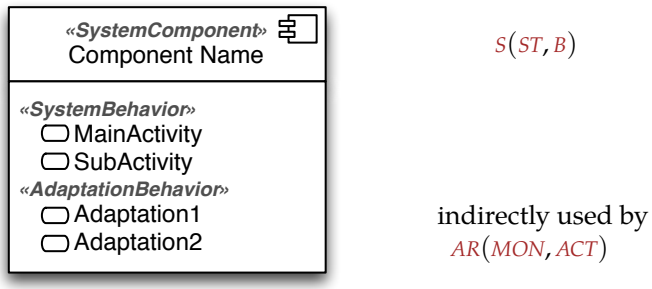


Figure 3.11. SystemComponent with AdaptationBehavior

As shown in Figure 3.12, the AVM further decorates components with adaptation interfaces, i.e. sensor and effector interfaces, which provide external ac-

cess to the adaptation behavior by means of operation definitions. That is, the effector operation *deactivateServer()* may be implemented by one of the two behaviors *Adaptation1* or *Adaptation2*. The sensor and effector definitions are used by the Monitoring *MON* and the Adaptation *ACT* that are part of the Adaptation Rule *AR* (cf. Page 66). While the monitor uses only sensor interfaces to observe and analyze the system and the environment, the adaptation activity may use both the sensors and the effectors to plan and execute an adaptation. The distinction of interface and implementation separates the interface definition from its implementation (cf. Requirement MR01) and further allows us to define underspecified sensors and effectors, i.e. adaptation interfaces that are not yet implemented. Sensors and effectors may further define properties which may have a computation specification. The defined sensors and effectors are used as an interface for the Adapt Case Model allowing to clearly separate the adaptation rule specification from the adaptation view model (cf. Requirement MR01). Finally, the usage of sensors and effectors fulfill Requirement MR08 as they are an integral part of the control loop architecture.

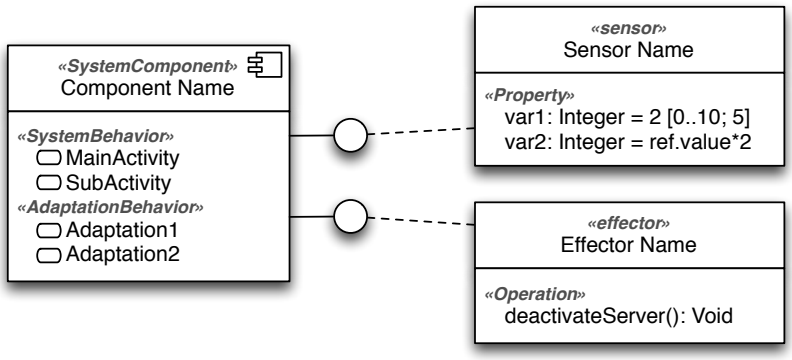


FIGURE 3.12. SystemComponent with AdaptationInterfaces (Sensors, Effectors) and Properties

The AVM allows to add additional knowledge to the model. Knowledge can be

- constraints (invariants, pre and post conditions, etc.) as shown in Figure 3.13
- histories of property changes
- histories of instance and type adaptations
- computations ranging over sensors and history, i.e. inferred knowledge
- reusable policies, i.e. activities that may be reused everywhere in the model

The support for constraints and histories fulfill Requirement MR06 that de-

mands for means to validate the model using formal analysis. The history icon shown in Figure 3.13 defines that for property *var2* every former value is preserved and can be accessed for adaptation or analysis purpose. The invariant shown in the figure is checked during analysis and provides application specific design-time guarantees. As such, invariants correspond to mathematical formulas which range over the Adaptation View Model *AVM* or the adaptation rules *AR*.

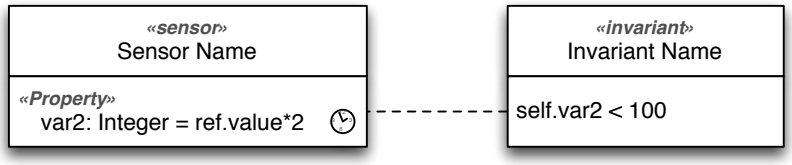


Figure 3.13. Sensor with attached Invariant and History

The AVM distinguishes between environment and system components. System components describe the system to be built (and may be decorated as described above). Environment components describe the environment *ENV* (cf. Page 64). They are black box components that cannot be changed by adaptation, though they may provide sensor and effector interfaces and might send signals, which correspond to events *EV*. Other events, such as *change events* may be specified by special UML *AcceptEventActions*. Figure 3.14 shows an `EnvironmentComponent` that exposes a sensor and defines that it may throw a signal. Again, the definition of signals allows to further decouple the AVM from the ACM and thus allows to separate different concerns (cf. Requirement MR01).

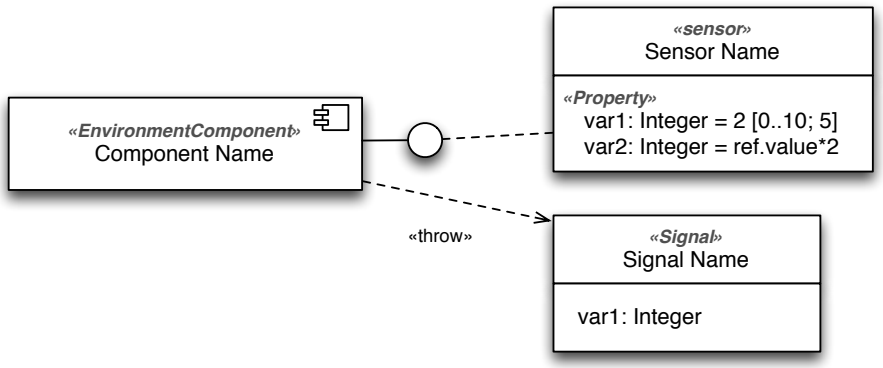


Figure 3.14. EnvironmentComponent with Sensor and Signal

The Adapt Case Model (ACM) defines adaptation rules (Adapt Cases) that make use of sensors and effectors. Adapt Cases define a monitoring activity (or more) which makes use of sensors to observe the system and the environment as shown in Figure 3.15 (cf. Requirement MR03 and MR08). By the use of specialized UML activities for the monitoring definition, we achieve a very generic specification mechanism. Since arbitrary UML actions may be used

in connection with specialized adaptation actions, almost arbitrary adaptation routines are imaginable. Thereby we fulfill the requirement for a general purpose language (MR11). Further, since for activity diagrams, precise formal semantics already exist (cf. [ESW07]) which have been extended for the specialized adaptation actions, the use of activities (for any behavior that is specified with the ACML) meets the requirement to enable formal analysis (cf. Requirement MR05).

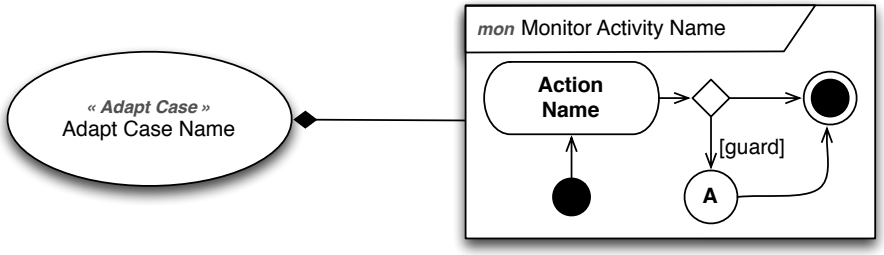


FIGURE 3.15. Adapt Case with Monitoring Activity

$MON\,(event, id_{ACT})$ or
$MON\,(CORR, id_{ACT})$
$CORR$ (corridors) are guards that access sensors properties, aggregations, computations, etc.

The Adapt Case may also retrieve signals which are broadcasted to both Monitoring and Adaptation Activity. These signals are defined as shown in Figure 3.16. The figure also shows how to define arbitrary conditions that must hold when the Adapt Case applies. Using the features of activity modeling, the monitor may analyze the data gathered from sensors or signals and decide to start an adaptation activity.

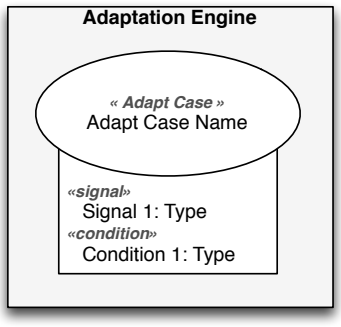


FIGURE 3.16. Adapt Case with Signal and Condition

As shown in Figure 3.17, Adapt Cases further define an adaptation activity (or more) which makes use of sensors and effectors to plan and execute adaptation. The adaptation activity makes use of activity modeling to orchestrate the use of different adaptation operations from effectors of different com-

ponents. Thereby, applying the adaptation activity manipulates the system: $\texttt{apply}(ACT) :: S \rightarrow S$ where ACT can be hierarchically nested with sub activities. Adaptation and monitoring activities both are adaptable behavior, as well. Hence, Adapt Cases may define the adaptation of other Adapt Cases, i.e. the definition of meta-adaptation. The use of UML activities for adaptation activities has the same benefits as described above for the use for the monitoring activity and thus several requirements are fulfilled (MR03, MR05, MR08, and MR11).

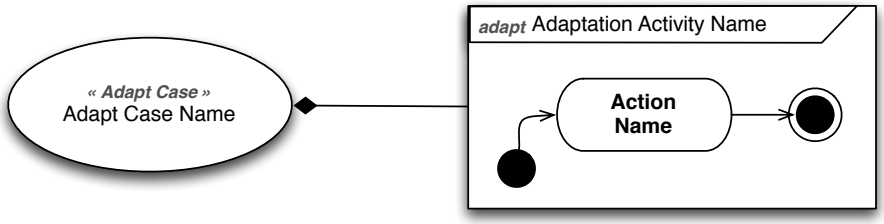


FIGURE 3.17. Adapt Case with Adaptation Activity

$\texttt{apply}(ACT) :: S \rightarrow S$

The ACML defines additional actions for the use in monitoring and adaptation activities. These actions allow to select instances or types within the AVM. Further, they allow to handle the history that can be built. More details about different types of actions are given below.

Figure 3.18 shows a complete example Adaptation View Model that reflects the example scenario from Section 2.1. The model defines a few system components and environment that expose different sensors and effectors. For instance, the bCMS system component exposes a sensor and an effector named *bCMS Process* for sensing and effecting process related information and the channel endpoint component exposes a sensor *Communication* for sensing the state of the communication channel. The model describes the structural part of the self-adaptive system. The behavioral part is described using Adapt Cases.

Figure 3.19 shows an Adapt Case diagram that defines an Adapt Case that handles the breakdown of communication. Therefore, it adapts the main use case *Communication with other Coordinator* (i.e. including all included use cases).

The basic idea is depicted in Figure 3.20. If the communication carrier (i.e. the channel) notifies that communication is not available, the corresponding signal is recognized by the adaptation engine (containing the Adapt Cases). Here, the signal is analyzed and eventually, an adaptation is planned and finally executed to cope with the non-available communication.

Figure 3.21 shows the Adapt Case that monitors the occurrence of the signal

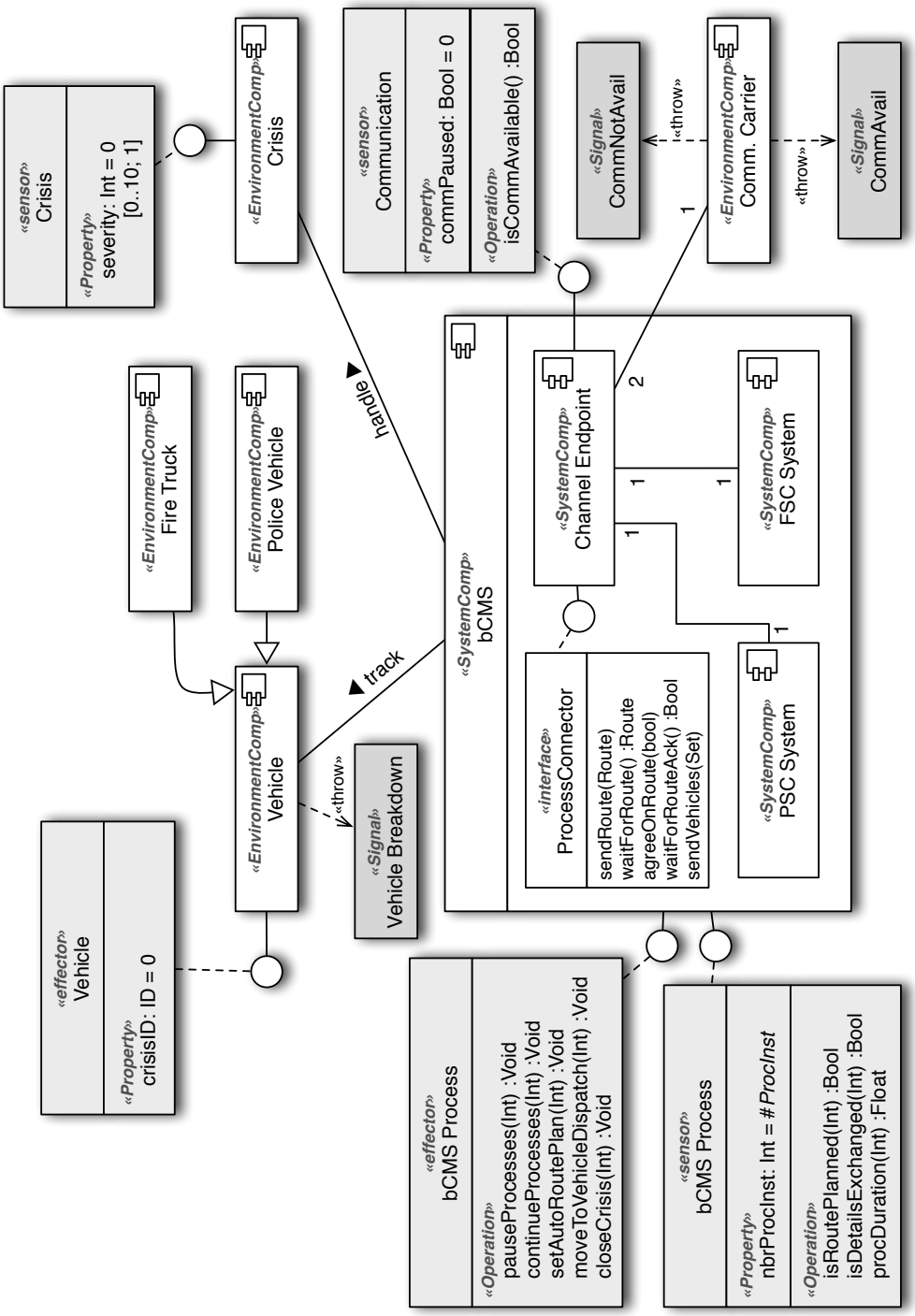


FIGURE 3.18.
AVM for bCMS Case Study

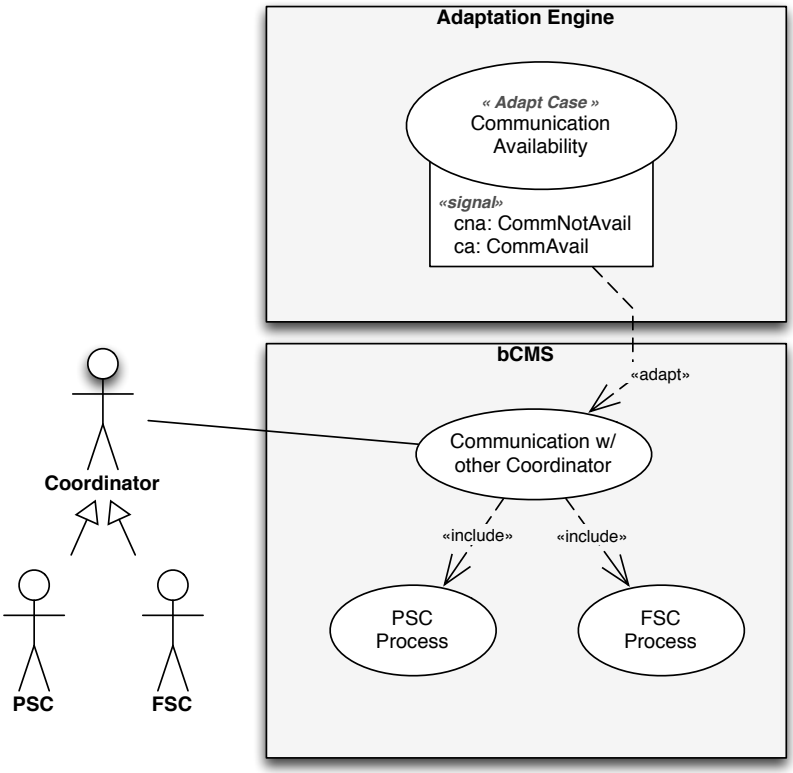


Figure 3.19.
bCMS: Adapt Case Diagram

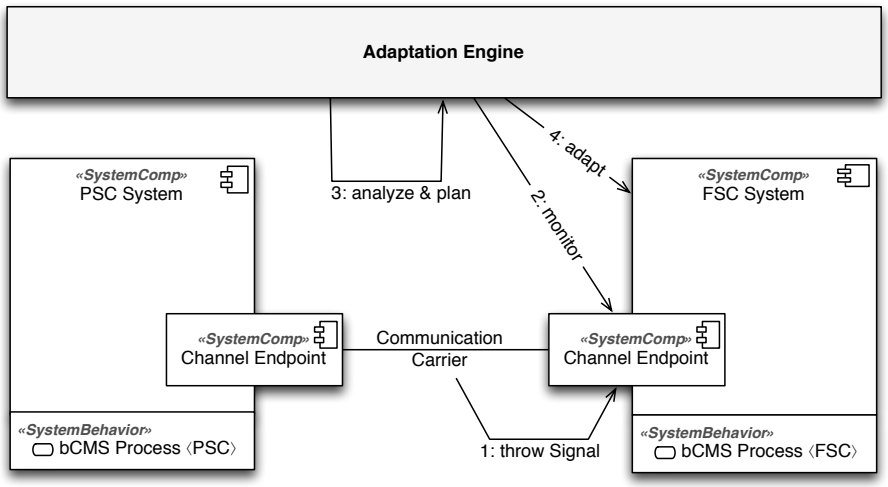


Figure 3.20.
bCMS: Overview with Adaptivity

CommNotAvail and triggers an adaptation if the route has not been planned yet. If the route has been planned, the signal is ignored assuming that both parties can proceed independently. Of course, here a notification could/should be send to both parties. The adaptation activity pauses the process if crisis details have already been exchanged (thus being in the route negotiation phase). After a defined timeout of 5 minutes, it checks if the communication is available again and continues the process again. If the communication is still not available, the processes are set to automatically plan the route. The corresponding process adaptation is hidden in the definition of the *setAutoRoutePlan(Int)* adaptation operation which us exposed by the effector `bCMS Process`. More details on this and other adaptation operations are given in Chapter 6 that discusses a complete model for the self-adaptive bCMS. All operations that are used are defined in the corresponding Adaptation View Model (cf. Figure 3.18).

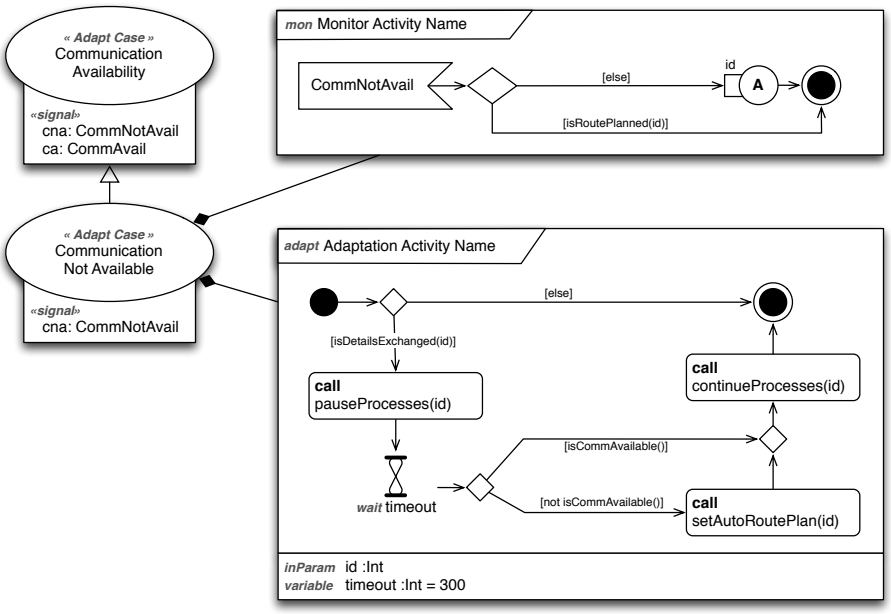


FIGURE 3.21.
Adapt Case for bCMS: Communication Not Available

Figure 3.22 shows the Adapt Case that continues the processes after the communication is available again. This Adapt Case is similar to the one shown before and in fact may interfere with the other Adapt Case possibly leading to deadlocks, etc. We will show how to analyze the Adapt Cases' interdependencies in Chapter 4.

Sections A.1 and A.2 in the appendix precisely show how Adapt Cases relate to the Adaptation View Model on the meta model level.

Adaptation Actions. The UML proposes a set of UML actions that allow the

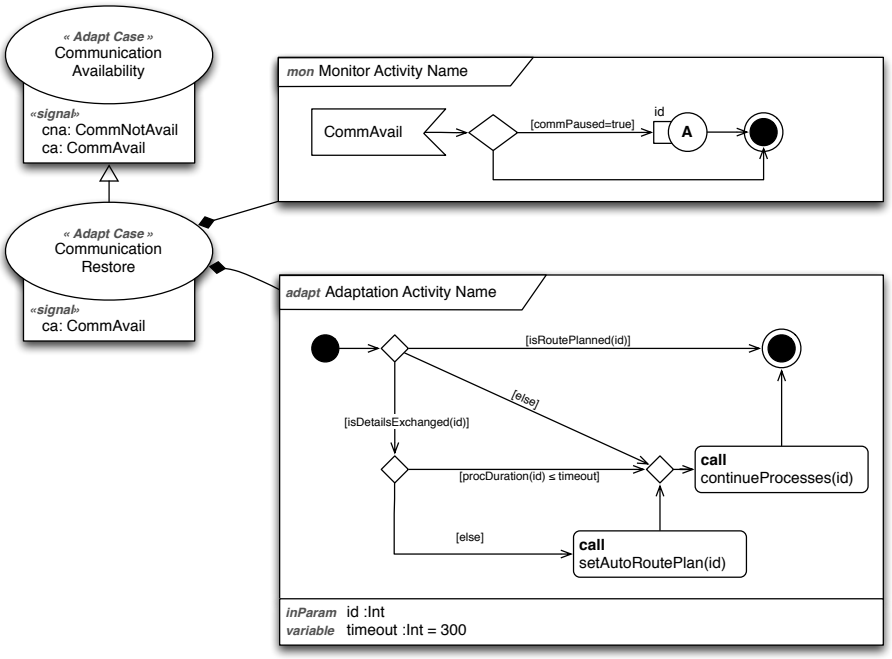


FIGURE 3.22.
Adapt Case for bCMS: Communication Restore

manipulation of instance models. For example, the UML defines an action that allows the creation of a link between two InstanceSpecifications. However, the UML does not provide any action that allows the manipulation of type models, nor does it provide actions to manipulate behavior such as activities. Hence, using plain UML, most of the language features described in Section 2.2 cannot be supported. That is why the ACML defines a set of specialized adaptation actions that inherit from the basic UML action and allow the manipulation of structure (component models) and behavior (activity models) on type and instance level.

SPECIALIZED ADAPTATION ACTIONS

Referring to the formal notion of adaptivity, an adaptation activity can be an arbitrary large set of actions: $ACT\ ACT\ ACT \ldots :: \textbf{AdaptationActivity}$. Each action ACT is an atomic or composed adaptation action that manipulates a system: $\texttt{apply}(ACT) :: S \rightarrow S$.

Figure 3.23 shows the action groups that allow the adaptation of structural information on type level. This includes the creation of ACMLComponents, ACMLAssociations, ACMLProperties, ACMLOperations, ACMLInterfaces, and ACMLReceptions. All of these elements may be changed as well. That is, e.g., an action might change the multiplicity, visibility, parameters, etc. of operations. A special sub group of actions is the reflective change group. This groups contains actions that, e.g., set attribute values by name instead of by reference. These actions are used whenever an attribute of an element shall be set which itself has been created by an adaptation action and thus cannot

be referenced, yet. The last special sub group of the structure type change actions is the adaptation history group. These actions allow to use the adaptation history, e.g. by reverting a earlier applied adaptation.

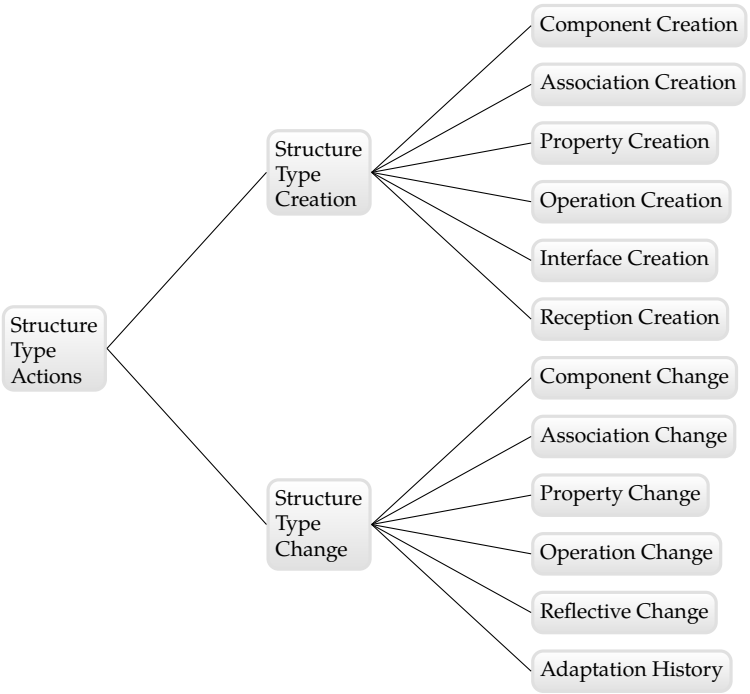


FIGURE 3.23. Adaptation Actions for Structure on **Type** Level

Analogously, Figure 3.24 shows the action groups that allow the adaptation of structure on instance level. This includes the creation of objects, links, and slots (property instances). Furthermore, the change sub groups contain actions that allow to change objets, links, and slots, as well as to delete single instances or all instances of a particular type. The instance upgrade group allows the upgrade of instances to new versions of their types if possible, e.g. by casting. Finally, the last sub group contains actions that support the handling of adaptation histories, e.g. to revert an earlier performed instance adaptation.

Besides actions for adapting the system's structure on instance and type level, there are actions for adapting the system's behavior. In the ACML approach, system behavior is described using activity diagrams, hence the action groups presented in the following allow the manipulation of UML activities. Figure 3.25 shows the action groups for adapting activities on type level. This includes the creation and the change of elements. Creation is further divided into the creation of activities, adaptive regions, flows, adaptive regions' children, value elements, and event elements. Adaptive regions are structured regions within activities which are adaptable by adaptation actions. Adaptive regions contain arbitrary UML activity modeling elements such as forks, joins, etc. The creation of these elements is covered by the action group *AdaptiveRe-*

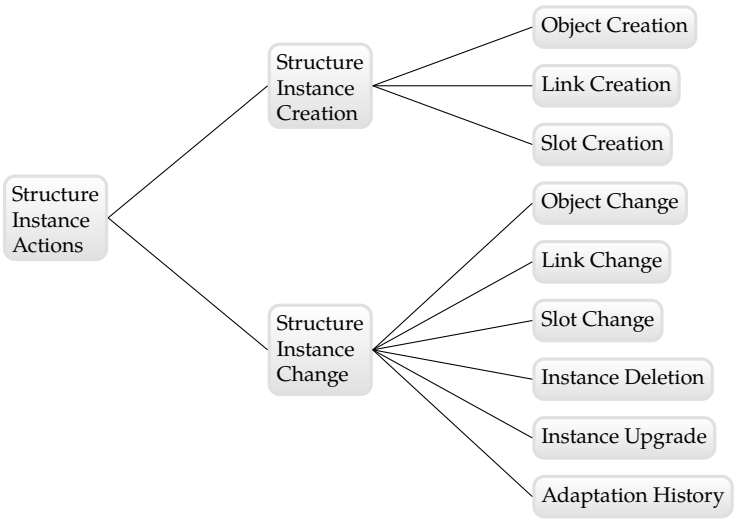


Figure 3.24. Adaptation Actions for Structure on **Instance** Level

gion Child Creation. Value elements subsume parameters, parameter sets and nodes, and variables. Event elements subsume actions to create events and triggers.

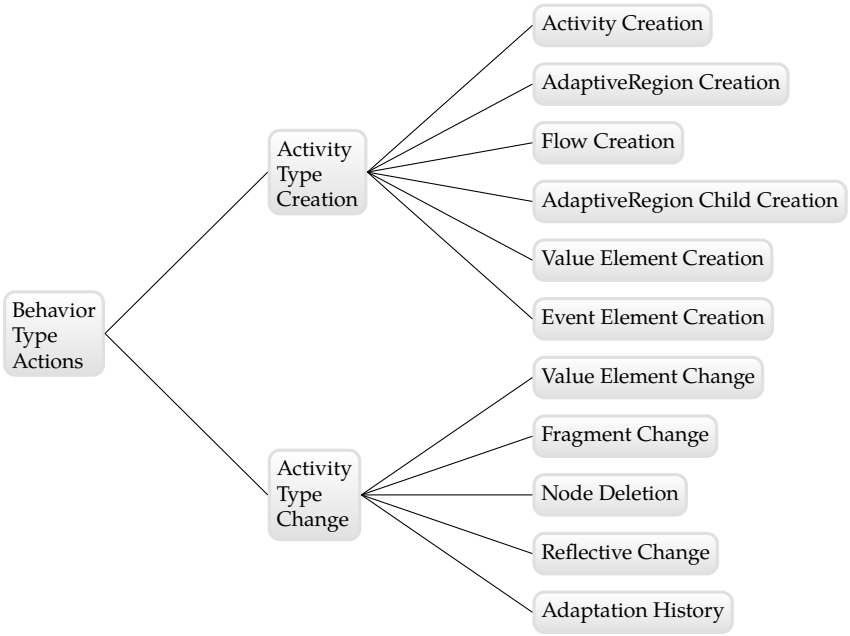


Figure 3.25. Adaptation Actions for Behavior on **Type** Level

Among others, the change group contains actions to change value elements, i.e. change their type, multiplicity, default value, etc. Fragment change actions are special actions that operate on single-entry-single-exit fragments, i.e. fragments of actions and flows with a single entry edge and a single exit edge. Example actions include copy, move, and parallelize. The next action group enables the deletion of arbitrary nodes within an adaptive region. Since elements within an adaptive region are not versioned, the nearest versioned con-

tainer (i.e. the adaptive region) is duplicated to form a new version. This way, no unintended side effects are generated if deleting, e.g., an action within an activity that has an existing instance. Finally, there is another group for reflective changes and for adaptation history actions, just like for *structural* type changes.

Analogously, there are actions for adapting activities on instance level, again divided into those that create and those that change activities. The creation of an activity instance is simply the *starting* of an activity, i.e. tokens are placed on the initial places according to the activity semantics. Activity change is further divided into groups to change values and states, to pause, stop, and continue an activity, as well as to upgrade an instance to a new activity type version and to revert an instance to a state before a previous adaptation has been performed.

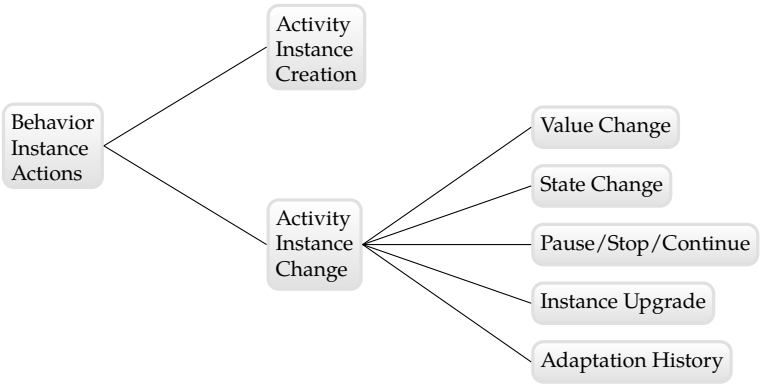


FIGURE 3.26. Adaptation Actions for Behavior on **Instance** Level

Type Adaptation with existing Instances. As mentioned above, the ACML allows the adaptation on type level. For instance, an Adapt Case might for some reason change the type of an attribute as shown in Figure 3.27. The attribute Location has a complex type GPS that within the instance an object is assigned to. If this type information is changed, the instance is not a valid instance of the class any more. In this case, the value can neither be casted to the new type nor is there any transformation rule given for propagating the type change to its instances.

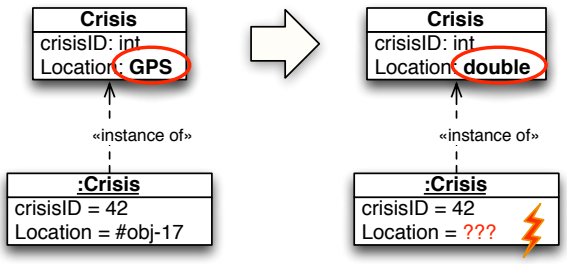


FIGURE 3.27. Adaptation of Type Information with existing Instance

A less complex solution for this problem is the introduction of versions. Whenever a type is changed that has existing instances, a new version of the type is created. Thus, the old version remains in the system and types the existing instances which in turn never lose validity. The versioning is depicted in Figure 3.28.

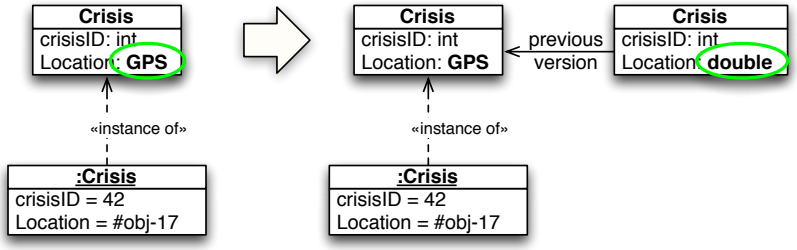


Figure 3.28. Versioning of adapted Types with Instances

The new version has a pointer to the previous version. Thereby, a chain of versions is built up, allowing for reconstructing the adaptation history and even reverting one or more adaptation actions. New instances are always created from the most current available version. Only the most current version may be adapted in order to not build up version trees which would be unnecessarily complicated. The concept of versions is described in terms of meta models in Section A.1 in the appendix.

3.3.3 ACML Language Features (from Taxonomy)

In this section we classify the ACML using the taxonomy from Section 2.2. Thereby, we precisely state what kind of adaptivity, the ACML can be used to model and how single adaptation features are supported by the ACML. That is, we describe how the ACML meets the special feature requirements that have been requested in Section 3.2.3.

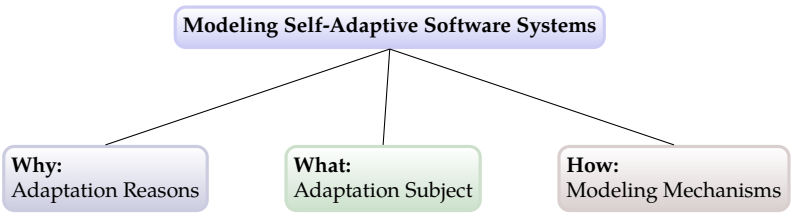


Figure 3.29. Taxonomy for the Modeling of Self-Adaptive Software Systems

Figure 3.29 again shows the three main categories that have been described in Section 2.2. The first category (Why) distinguishes four self-* properties: self-optimizing, self-protecting, self-configuring, and self-healing. The ACML

supports the adaptation specification for the sake of all four self-* properties. This is because of the generic nature borrowed from UML activity modeling. Basically, the monitoring and adaptation activity may use arbitrary actions to either optimize a running system (e.g. minimizing the cost of a server farm by de/activating servers in dependence of the load), protect a system (e.g. disconnecting a particular user in the light of malicious actions), configure a system (e.g. changing the system's behavior in response to a changed environment), and heal a system (e.g. exchanging a used web service if it fails).

The second category (What) describes the subject of modeled adaptation. Table 3.2 describes the ACML's classification.

Table 3.2. What: Adaptation Subject

Feature		Supported
Scope		
	Instance	yes
	Type	yes
	Temporary	yes
	Permanent	yes
Model Type		
	System Structure	yes
	System Behavior	yes
	Adaptation	yes
Artifact & Granularity		
	Parametric	yes
	Compositional	yes
	State Changes	yes
Aspect Type		
	Aspect-Specific	no
	Generic	yes

Both, instance and type adaptation are supported using specific actions within Adaptation Activities or Adaptation Behaviors. During modeling, the scopes instance and temporary as well as the scopes type and permanent coincide since a type change has an effect on every future instance and an instance change drops away when new instances are created.

The ACML supports modeling the system's structure (using classes and components) as well as behavior (using activities). The adaptation of both kinds may be specified using Adapt Cases. Further, Adapt Cases may be specified to adapt other Adapt Cases. Thus, the ACML supports the adaptation specification concerning all three model types.

Using specific actions within the monitoring and adaptation activities, Adapt Cases may not only describe the adaptation of parameter changes, but also the adaptation of the system's composition and state.

Finally, the ACML is not specific to any particular aspect such as performance, but is designed as a generic, general purpose language to describe adaptation in any domain for any particular aspect.

Considering the third category in the taxonomy (How), the ACML supports the features described in the following (Table 3.3).

Table 3.3. How: Modeling Mechanism

Feature		Supported
Location of Adaptation		
	horizontal	
	Local only	no
	Global	yes
	vertical	
	Process Layer	yes
	Service Layer	yes
	Component Layer	yes
Timing		
	Reactive	yes
	Proactive	(with histories)
Direction		
	Forward (new state)	yes
	Backward (old state)	yes
Triggering		
	Event Triggering	yes
	Time Triggering	yes
Anticipation		
	Non-Anticipated Change	no
	Anticipated Change	yes
Automation		
	Autonomous	yes
	Human	no
	Interactive	no
Specification		
	implicit	no
	explicit	
	imperative	yes
	declarative	no
	Separation of Concerns	yes
	Mixed Concerns	no
	Genericity	
	Domain-specific	no
	Generic	yes

The adaptation specifications created with the ACML have access to global information via standard UML qualified access. Of course, models can be specified to only use locally available information to ease an implementation.

Software systems can be divided into three different layers: a component layer that contains functionality encapsulating components, a service layer that contains services that group several components to provide a logical service via interfaces, and a process layer on top that orchestrates several services using some kind of process notation. The ACML is designed to allow the specification of adaptation on all three different layers. On process layer, process models can be adjusted regarding their control flow and state. On service layer, service bindings can be monitored and adjusted. On component layer, component internals such as classes, associations, and properties can be changed arbitrarily.

Regarding the timing of adaptation, the ACML allows both the reactive and

proactive adaptation. Reactive adaptation can easily be achieved by polling the necessary information (e.g., environment properties) or by reacting to signals or other events. Proactive adaptation can be achieved by analyzing histories of system and environment changes, however, the ACML does not support dedicated analysis techniques such as prediction models but only provides a generic construction kit in terms of extended activity diagrams.

The ACML supports forward as well as backward adaptation, that is, by adapting the system, either new system states can be generated or old system states can be restored from history using specific adaptation actions. Adaptation can be time and event triggered, both implemented using standard UML features. Time triggering can be of form *each x minutes* or *at time x*. Events include signals, changed variables, exceptions, and more. The ACML is designed to allow the specification of anticipated adaptation. Non-anticipated adaptation is not supported. ACML specifications allow for autonomous adaptation. Human interaction is not yet supported exceeding the standard UML features. The adaptivity specifications created with the ACML are explicit and imperative. The ACML allows for easy declarative change descriptions, however, this is not yet supported for every diagram type. Finally, the ACML uses the principle *separation of concerns*, that is mixing of adaptation and business logic concerns is not intended. Further, the ACML is not domain-specific but a general purpose language for any kind of self-adaptive system. A small exception are business processes which are supported with a dedicated set of adaptation actions.

As shown within this section, the ACML is a domain-independent adaptation specification language which is designed to be as generic as possible while still providing sufficient details for early and expressive quality assurance.

3.3.4 ACML on Meta-Model Layers

The Adapt Case Modeling Language (ACML) has been defined in terms of meta models (cf. appendix). The basic concept of the definition is described in this section.

Figure 3.30 shows the languages on the meta-model layer M2 and concrete models on level M1. The behavioral model (*Adapt Case Model*) allows the specification of adaptation rules, a self-adaptive system's behavioral part. The rule description, called Adapt Case, is based on use case modeling and uses UML

activity modeling to describe the manipulation of the system. For that purpose, Adapt Cases refer to adaptation interfaces (to influence system and environment) and react to signals defined in the Adaptation View Model.

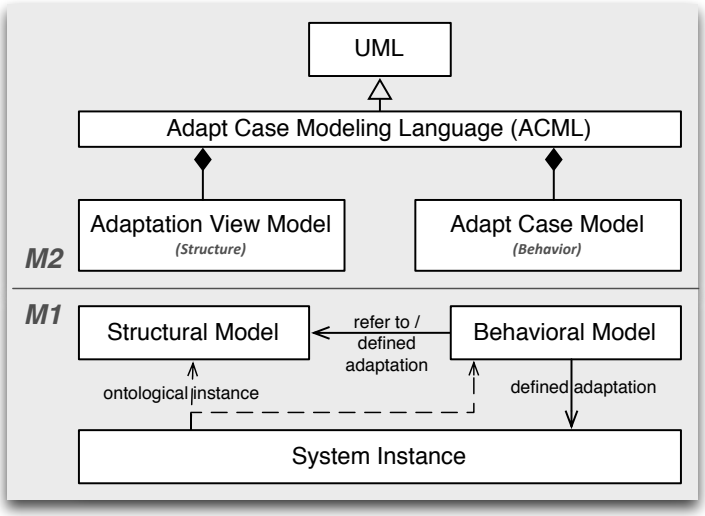


Figure 3.30. Our Models to describe Self-Adaptive Software Systems

A concrete system instance may be represented using an object diagram taken from the UML. If Adapt Cases are interpreted on model level, they adapt (i. e. manipulate) the system instance model, or in the case of type adaptation, the structural model, respectively.

The structural model (*Adaptation View Model*) supports the description of the system and its adaptation interfaces. The diagram allows specifying an adaptivity-specific view onto the system. That is, the model supports the definition of adaptation interfaces that are used to sense and effect the system as well as to sense and influence the environment. Further features include the specification of adaptation knowledge which is used for adaptation.

The M1 layer is detailed in Figure 3.31. Adaptation-relevant parts of system and environment are modeled using extended UML diagrams (e.g., state machines, activities, classes, and components) using object-oriented design with discrete states.Adaptation rules are described by Adapt Cases in event-condition-action rule style using extended use cases and activities (incl. actions). Instances (still within M1) are described by UML objects (i.e., instance specifications). An executed Adapt Case instance may adapt the system on instance level or on type level. Additionally, Adapt Cases may adapt other Adapt Cases, known as *meta adaptation*.

Adaptation View Model and Adapt Case Model are ontologically instantiated

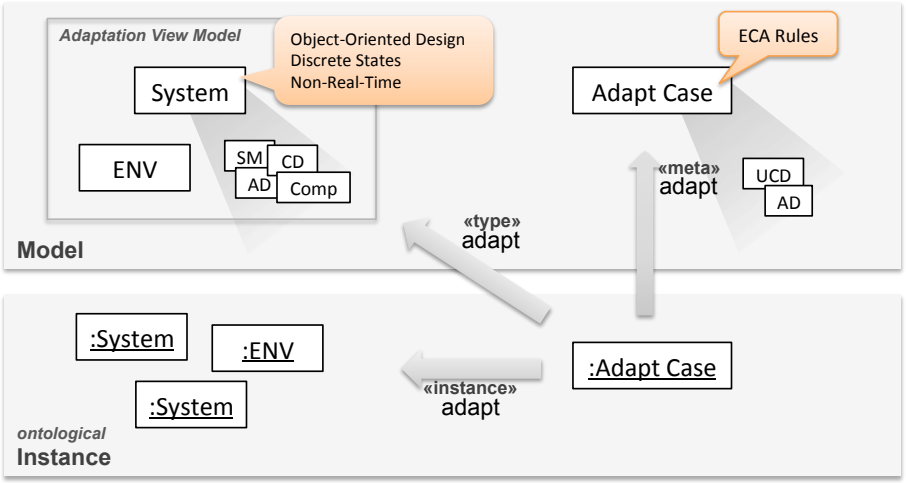


FIGURE 3.31.
Models on M1 layer (Machine Model)

Type Adapt Cases vs. Instance Adapt Cases Adapt Cases can be used to describe a wide range of different adaptation scenarios. In the following, we want to distinguish between two different categories of adaptation scenarios since they may be represented differently in use case diagrams: Instance Adapt Cases and Type Adapt Cases.

Instance Adapt Cases describe the adaptation of some system functionality concerning one particular instance, e. g. a user or a product.

Type Adapt Cases describe the adaptation of some system functionality for all future usage, i. e. for all possible instances.

ADAPTATION ON TYPE LEVEL KEEPS DESIGN CLEAN

Instance Adapt Cases may be denoted by prefixing their abbreviation with the small letter "i". An Instance Adapt Case resembles the *extends* relationship between two use cases since in the case of an Instance Adapt Case, the targeted use case decides per instance whether or not the Adapt Case is used to adapt the original behavior. Thus, Instance Adapt Cases can be included in the same system as use cases. Type Adapt Cases change the internals of the targeted original use case, i. e. they are changing the type of that use case. The execution of a Type Adapt Case resembles a transformation of a use case. Because of that, Type Adapt Cases should not be included in the same system, the use cases are. They are contained in a separate system (e.g. adaptation engine) and have a *type-adapt* relation to the use case they are adapting. See Figure 3.32 for an example. The *Handle Vehicle Breakdown* Adapt Case reacts to some signal and performs some action to assign a new vehicle to the crisis. Theoretically, this Adapt Case could be completely integrated into the adapted use cases, although of course this would violate the principle of separating concerns and the designer would loose the ability to perform dedicated analyses.

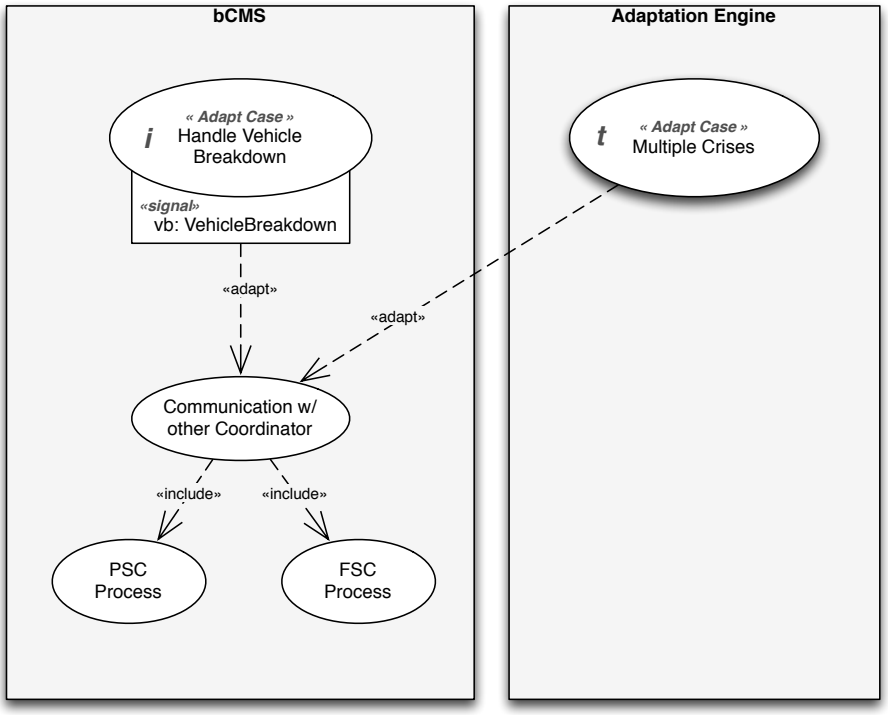


FIGURE 3.32.
Instance vs. Type Adapt Cases

Type Adapt Cases completely change the use case they are targeted at. After a type adaptation, the old use case cannot be found in the system any more at all. Type Adapt Cases may be denoted by prepending a small letter "t" to their abbreviation.

The Type Adapt Case *Multiple Crises* in Figure 3.32 monitors the number of crises that are handled by the system and if a given threshold is exceeded, the main processes is adapted such that less manual and more automatic activities are included. Modeling this adaptation on the instance level would heavily bloat the complete design, while its definition on type level is clean, separated and comparatively small.

Note that in our case the distinction of Type and Instance Adapt Cases is only important for the diagrammatic visualization of Adapt Cases. The meta model does not reflect this distinction since in the end both Type and Instance Adapt Cases adapt some element of the adaption context. The concrete adaptation actions, however, are divided into type adaptation and instance adaptation actions as described in Section 3.3.

3.4 SUMMARY & DISCUSSION

In this chapter, the Adapt Case Modeling Language (ACML) has been introduced. The ACML consists of two different models, the Adaptation View Model, which allows the system model to be decorated with adaptation-specific information (e.g., sensors and effectors), and the Adapt Case Model, which allows the specification of the actual adaptation rules using extended UML Activities.

In Section 1.2 we identified a few high-level requirements that must be addressed by a modeling and quality assurance approach for self-adaptive systems. In the following, we recapture the requirements considering the presented Adapt Case Modeling Language (ACML):

Separation of Concerns By providing adaptivity specific views on the system (i.e. two new diagrams kinds), the ACML supports the separation of concerns. The paradigm is further leveraged in the usage of interfaces for the communication between the Adapt Cases (adaptation rules) and the Adaptation View Model (adaptation-specific view). As an integral part of separation of concerns, the ACML also supports the composition of the created models with remaining system models using standard UML features such as associations, inheritance, etc. Especially the use of UML components as a basis for the Adaptation View model allows the composition of the different concerns' views.

Analyzability The ACML is based on UML model elements, such as activities, that exhibit clear semantics. These semantics (i.e. token-offer semantics) has been formally defined using the DMM approach. The extensions of the UML that have been introduced by the ACML either do not break the UML semantics or come with clear semantics extensions within the DMM approach. Thus, the presented approach shows to be analyzable using standard techniques such as model checking (see Chapter 4).

Integrated Since the ACML is fully integrated into the UML and enables to create links to standard UML model elements such as use cases, classes, components, or the like, the ACML can be used in any UML based development process. In turn, most of the well-known development processes such as RUP, SCRUM, etc. are supported by the UML or derivatives such as the SoaML.

Intuitiveness The ACML uses standard modeling techniques that are pro-

vided by the well-known and accepted UML. The clear separation of different internal concerns such as Adapt Cases and the Adaptation View Model help mastering the complexity of specifying self-adaptive software systems. The support of reusing specification artifacts such as Adapt Cases or sensor and effector specifications support quick success. As will be shown in the evaluation in Chapter 6, the ACML is easy and intuitive to use.

Genericity The ACML is designed with genericity in mind. That is, the ACML does not contain any domain-specific modeling elements but rather provides the toolkit to create these domain-specific modeling elements from atomic generic elements. Of course these created domain-specific elements can be reused in different specifications. Since the ACML extends the UML, even UML Profiling can be applied to the ACML to create domain-specific extensions of the ACML itself. Thus, the ACML is a general-purpose language.

These high-level requirements have been refined in Section 3.2.3 and addressed in detail in Section 3.3. A detailed description of the ACML in terms of metamodels is given in the appendix (see Section A.1 and Section A.2).

On the basis of the presented ACML, we define a quality assurance framework called QUAASY that is presented in the next chapter.

4

Quality Assurance for Self-Adaptive Systems

"Quality is never an accident. It is always the result of intelligent effort."

– John Ruskin

4

ANALYZING INTERDEPENDENCES OF CONTROL LOOPS

Control loop focused languages such as the ACML that allow the specification of control loops as first class entities, advise the need for analytical methods. In particular, verification techniques are needed to evaluate the modeled control loops' behavior and detect unintended behaviors and interactions [CLG+09]. This is especially important since the creation of highly self-adaptive software systems implies the specification of *many* interacting control loops (i.e., Adapt Cases) the correctness of which can hardly be assured without any dedicated means. Hence, in this chapter, we will describe means to assure the quality of a self-adaptive software system that has been modeled using the ACML.

In Section 4.1, we will discuss the analysis subject and quality properties that are of interest for self-adaptive software systems. Further, we infer requirements and discuss related work in the area of quality assurance for self-adaptive systems. In Section 4.2 we present our approach to quality assurance for self-adaptive software systems named QUAASY. Since our approach relies on model checking, we have to pay special attention to the problem of state space explosion which will be done in Section 4.3. Finally, in Section 4.4, we summarize and discuss the chapter.

ANALYSIS 4.1

Analytical methods allow checking the quality of the modeled self-adaptive software system in spite of high complexity. Several static and dynamic analytical methods exist, including the detection of anti-patterns, testing, model checking, and simulation. Since dynamic analysis like testing requires an implementation of the system to be present, it cannot be applied in early system design phases. In contrast, static analysis like model checking allows the early quality assurance of the system to be built since it relies on the design models. Furthermore, while testing allows to test single execution paths through the system only, model checking enables checking the complete state space of the modeled system.

STATIC ANALYSIS OVER DYNAMIC ANALYSIS

In this section, we analyze the need for static analysis methods (i.e. quality assurance) for self-adaptive systems. That is, we investigate what kind of properties shall be checked for modeled self-adaptive software systems, and based on these findings and the high-level requirements defined in Section 1.2, we infer the requirements for our approach and discuss related work.

4.1.1 Static Analysis of Self-Adaptive Systems

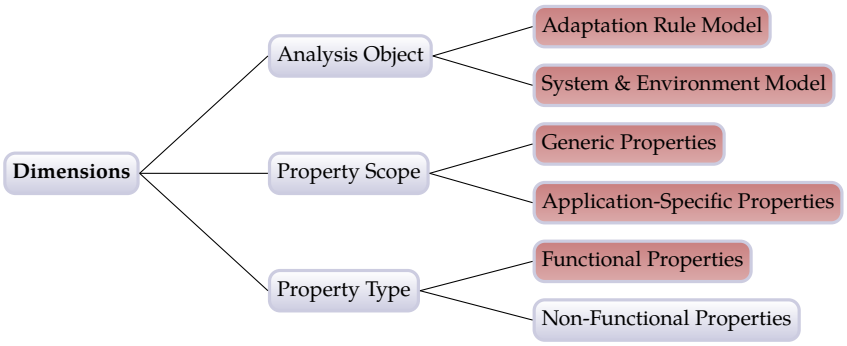


Figure 4.1. Dimensions of Properties for Self-Adaptive Software Systems

A self-adaptive software system can be described in terms of application and adaptation logic. A description of the application logic comprises the system itself and the environment the system acts in. The adaptation logic can be described by a set of adaptation rules, e.g., using the ACML. Both kinds of descriptions can be analyzed statically for various different properties. Figure 4.1 shows the dimensions of static quality assurance for self-adaptive systems:

The **Analysis Object** can be distinguished between the adaptation rules and the system & environment. That is, either the static analysis is build to check certain properties for the adaptation rules, only, or the analysis approach checks the system and its environment as well, e.g. whether or not the system is still able to run after an adaptation has been applied.

The **Property Scope** of an approach distinguishes the kind of properties that can be checked. Either, the properties are of generic character, that is, they can be applied to every self-adaptive system that can be modeled, or the properties are application-specific, that is, they can only be applied to the specific software application they were defined for.

Generic properties are not application-specific, but express properties which every self-adaptive system must assure. An example is the absence of deadlocks when adaptation rules are applied. Generic properties are defined over the meta model and are checked on instance level.

Application-specific properties are defined over a concrete system model and assure application-specific properties, which are usually referred to as invariants and pre/post conditions. An example for application-specific property is that a web server's availability must never be affected by some adaptation actions.

Finally, the properties may be of *functional* or *non-functional* **Property Type**. Functional properties include common safety and liveness properties (e.g., deadlock freedom, reaching a particular state, or never entering particular states). Non-functional quality properties are usually application-specific. That is, applied to self-adaptive systems, well-known properties (e.g., from the ISO9126) such as robustness, reliability (fault tolerance), availability (down times, recovery of data), and efficiency must be defined using application-specific metrics. It is depending on the specific application how long the system may be down, or which faults have to be tolerated.

In this thesis, we focus on assuring functional quality of the adaptation rule model as well as the system and environment model using generic properties or user-defined application-specific properties (cf. red colored-nodes in Figure 4.1). We will provide a model-checking framework for self-adaptive software systems that allows the use of arbitrary (quality) properties that can be expressed using temporal logic. Nonetheless, we will describe several example properties that are of importance for self-adaptive software systems.

For the following description of the (quality) properties, we distinguish two classes of behavior. This will be illustrated in Figure 4.2 that extends the ma-

chine model given in Figure 3.31 on Page 100 with a semantic domain we use for self-adaptive software systems that have been modeled using the ACML.

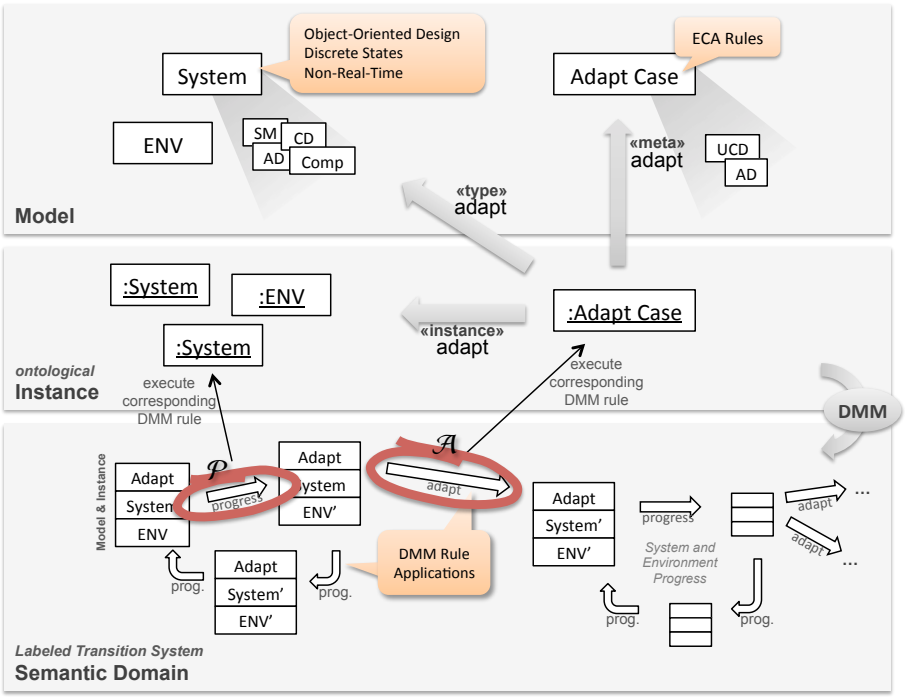


Figure 4.2. Machine Model *ACML* mapped to Semantic Domain *LTS*

LABELED TRANSITION SYSTEMS AS SEMANTIC DOMAIN

The semantic domain of ACML modeled systems are labeled transition systems (LTS). An LTS state contains the self-adaptive system's model (type) and instance (i.e., classes, activities, *and* objects) since Adapt Cases may adapt both the model *and* the instances. An instance in an LTS state is always a proper ontological instance of the corresponding system model in the same state. An LTS transition corresponds to executing the operational semantics (defined via DMM graph transformation rules) of the environment, the system, or the adaptation rules. For instance, the operational semantics for the environment define that open variables may decrease or increase arbitrarily. Hence, an LTS transition (see *progress* arrow) might express the increase of an environment variable leading into a new LTS state. If further, an Adapt Case is triggered by the increased environment variable, this Adapt Case will be executed according to the corresponding DMM semantics leading into a new LTS state that contains the adapted system (see *adapt* arrow). Thus, we distinguish two different classes of semantic behavior corresponding to the different nature of the various LTS transitions. We use the semantics function $[\![.]\!]_{DMM} :: ACML \rightarrow LTS$ to denote the mapping of our notion for self-adaptive systems (cf. Section 3.2.2) into the semantic domain of labeled transition systems. Now, the two classes of behavior are given as follows:

Adaptation Behavior describes the adaptation of the system in the light of a change in the environment or the system itself. Adaptation behavior is given by the description of Adapt Cases. We denote the set of adaptation behavior with $\mathcal{A}$. Essentially, $\mathcal{A}$ represents the semantics of a set of adaptation rules: $\mathcal{A} := [\![\textbf{ACM}]\!]_{DMM}$.

System and Environment Progress Behavior describes the change of system and environment. The change of the system considers every change which occurs due to the execution of application logic or the occurrence of errors. The change of the environment describes changing variables, signal existences, etc. in the environment. Both types of change must be specified if essential for the adaptation logic. Both types are specified using the Adaptation View Model. We denote the set of progress behavior with $\mathcal{P}$. Essentially, $\mathcal{P}$ represents a set of progress rules, i.e., rules that described how the system and the environment changes: $\mathcal{P} := [\![\textbf{AVM}]\!]_{DMM}$.

Please note that the systems that are modeled using the ACML are highly parallel. More precisely, adaptation behavior and system and environment progress behavior are performed interleaved. That is, e.g., there may be different *adapt* transitions originating from one state. Let us now describe and define the different functional properties.

Functional (Quality) Properties of Self-Adaptive Systems

Functional quality properties check whether the system (including adaptation) behaves correct according to some specification (application-specific properties). Besides the validity according to the system's requirements, we understand a self-adaptive system to be correct, if no errors, conflicts, and deadlocks exist (generic properties). In turn, the rules must terminate, be stable, and preserve specific safety and liveness properties. Additionally, properties such as confluence and determinism of a self-adaptive system's adaptation rule set contribute to the high quality of the system. In the following, we will define the properties in more detail.

Errors Traditionally, checking functional correctness requires the analysis object to be syntactically correct. However, since self-adaptive systems may change their syntactical representation at runtime, we have to reconsider this definition of functional correctness. Thus, we distinguish between semantic correctness and syntactic correctness both of which have the be achieved in order to meet the overall goal of functional correctness.

In this thesis, *errors* refer to syntactical correctness, that is syntactical er-

rors in the specification of the system, its environment, and its adaptation rules. Especially, for the various kinds of adaptation actions that can be applied to the system, it is possible to invalidate the models, e.g. by deleting the start node of an activity or leaving dangling references. These kinds of errors should be detected already during modeling time.

Compared to non-adaptive systems, the modeling of self-adaptivity induces additional complexity concerning *errors* since the *specification* of a model change cannot be checked by simple syntax checkers (e.g. meta model based validation checkers) as the changed model is not at hand during modeling time. Thus, it is important to assure that adaptation does not invalidate the system using techniques different from standard validation checkers.

Conflicts Conflicts are of special interest for self-adaptive systems since in our machine model adaptivity is highly parallel to both the system progress and to itself, i.e., multiple adaptation rules (Adapt Cases) may be executed at the same time. Thus, adaptation rules may be in conflict with each other or with the system they are adapting. A set of rules is in *conflict* if the rules cannot be executed at the same time. An example is an adaptation rule that invalidates another rule while the latter is executed. This problem occurs especially for higher-order self-adaptive systems where adaptation rules may adapt other adaptation rules.

Deadlocks A set of rules is *free of deadlocks* if no adaptation rule puts the system in a state where no other rules (out of $\mathcal{A}$ or $\mathcal{P}$) can apply. Deadlocks can be understood as a weak form of conflicts since deadlocks can usually be resolved by adding another subject- / application-specific rule that reestablishes a *good system state* whereas conflicts would rather need low-level application-unspecific rules that do not reflect any specific system requirement.

Termination Termination applies to adaptation behavior as well as system & environment behavior. Hence, termination has to be checked for every single adaptation rule in $\mathcal{A}$ and progress rule in $\mathcal{P}$. A rule does not terminate if the system's LTS contains any path where the rule is started but does not finish eventually. Termination is especially important for self-adaptive systems since the property could possibly be invalidated after each adaptation and thus has to be checked for various different system configurations.

Livelocks / Stability Opposed to deadlocks, a set of rules is in a livelock if it is not blocked or waiting for anything but has an infinite loop of rule

executions. The property of livelocks cannot be checked for environment rules since these may occur arbitrarily, even infinitely often, and thus are out of control. An adaptation rule however is in a livelock if, e.g., it does not change anything but can be applied over and over again.

The instability property is a subset of the livelock property. A set of rules is stable if no two or more rules continuously undo their adaptation actions without any changes in the environment or the system. In turn, a system is stable and free of livelocks if it can always reach a state where no adaptation rule is applicable.

Application-Specific Safety and Liveness Properties Application-specific properties are defined for a specific application. Examples are invariants, pre- and postconditions or other specific safety and liveness properties. In general, safety properties define that something bad never happens. Therefore, application specific bad states can be specified or described via invariants that are checked for each state of the system's state space.

For instance, a safety property that is specific for self-adaptive systems is the existence of transient error states. If the system is in a transient error states it is cured by taking some self-adaptation action. If for a specific application, these states should never be reached, the property can be formulated as a safety property. These states have to be defined specifically for an application (application-specific property).

If for a specific application, the system is allowed to be within a transient error state if it is able to cure itself (e.g., within a specific time frame), this can be formulated as a liveness property. Liveness properties express that under certain conditions something good happens eventually, that is, e.g., if the system is within a transient error state it will leave this state at some point in time.

Confluence The confluence property states that within a transformation rule system there are different ways of transforming a source into a target. The property is often referred to as *the diamond*. For self-adaptive systems, essentially, the property says that if more than one adaptation rules may be applied in the same state, the choice of which adaptation rule is applied to a particular system state does not matter. Further, as a more special case, the order in which adaptation rules are applied does not matter, either. Obviously, this is a desired property for a self-adaptive system to be comprehensible.

Determinism Determinism is a rather strong property for self-adaptive sys-

tems. However, if it holds for a system, comprehension and analysis is eased. A self-adaptive system is deterministic if in all states there is never more than one adaptation rule that can be applied. Obviously, if a system is deterministic, it is confluent, too. Determinism does not require the adaptation rules to be executed atomically, that is, if an adaptation rule is currently executed, another one may be started without violating the determinism property.

We will formalize these properties later in Section 4.2.2. In the next section, we infer requirements for a quality assurance approach for self-adaptive software systems with focus on functional quality and discuss related work.

4.1.2 Requirements & Related Work

In this section, we describe the requirements for a quality assurance approach for self-adaptive software systems. The requirements are inferred from the high-level requirements from Section 1.2 and additionally reflect the findings from the last Section 4.1.1. The requirements will be used to compare related work, and finally, develop a suitable quality assurance approach.

Requirements Based on the high-level requirements (Separation of Concerns, Analyzability, Integrated, Intuitiveness, and Genericity) and our notion of self-adaptivity, we derive the following requirements for a quality assurance approach for self-adaptive software systems:

Separation of Concerns The principle of separation of concerns allows the focus on a particular software concern, and thus, enables the dedicated quality assurance of that concern. As such, *separation of concerns* is the basis for a quality assurance approach. However, when separating concerns, the interdependencies between these concerns and the remaining system must be investigated as well. Therefore, a good quality assurance approach should not only analyze the concern in isolation but also its impacts to the remaining system concerns.

For our quality assurance approach, we infer the following requirements:

- QR01: The approach must allow the analysis of the adaptation rule set.

- QR02: The approach must allow the analysis of effects and impacts on system & environment when adaptation had been applied.

Analyzability A high-level requirement of the overall engineering approach for self-adaptive software systems is *analyzability*. As described in Section 4.1.1 there are many facets of quality assurance for self-adaptive software systems. We focus on the functional properties of these systems that have been enumerated above.

Thus, for our quality assurance approach, we infer the following requirement:

- QR03: The approach must allow the analysis of the system's functional quality (i.e., correctness).

Integrated An important requirement of both the modeling and quality assurance approach is its integratability with other engineering approaches. For the quality assurance approach this particularly affects the way the approach is integrated in the engineering tools and how feedback is provided for the user.

For our quality assurance approach, we infer the following requirement:

- QR04: The approach must support the immediate feedback during design-time.

Intuitiveness Most important for the acceptance of a quality assurance approach is its intuitiveness during use. This is impacted by the amount of knowledge the user has to have to be able to use the approach.

For our quality assurance approach, we infer the following requirement:

- QR05: The approach must be easy to use without knowledge of formal analysis techniques (e.g., model checking). Ideally, the user does not get aware of the concrete formal technique used.

Genericity The genericity of a quality assurance approach has an impact on its power. Is the approach usable within any domain and with any model for the particular concern?

For our quality assurance approach, we infer the following requirement:

- QR06: The approach must support checking generic and application-specific properties. Generic properties can be applied to every modeled self-adaptive software system. Application-

specific properties are defined for a specific application.

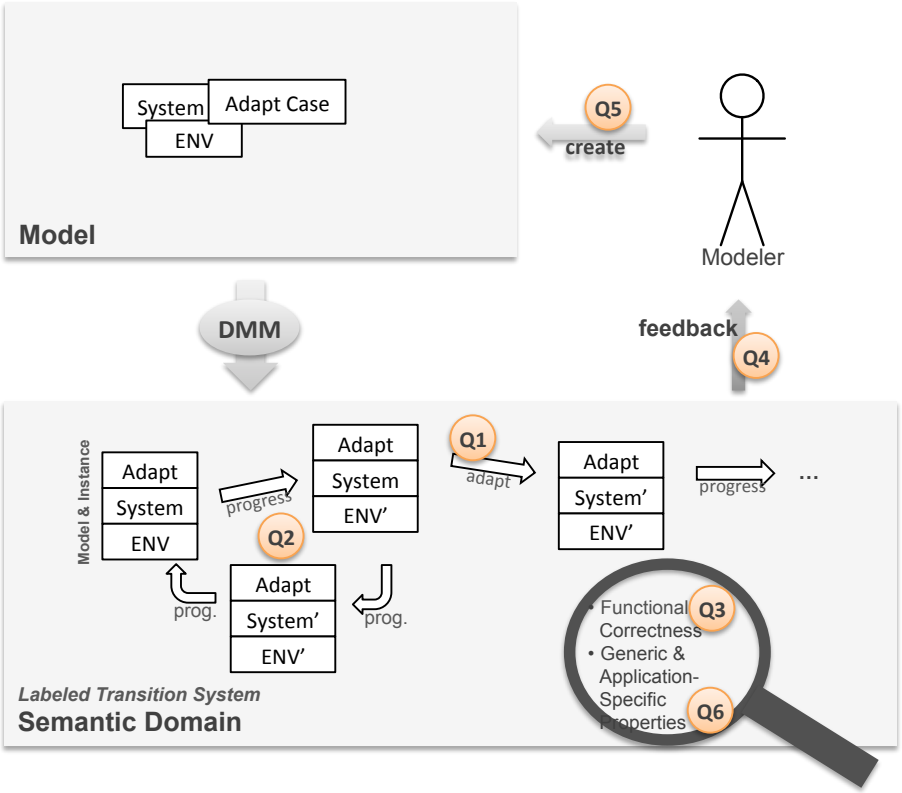


Figure 4.3. Requirements covering the Machine Model from Figure 4.2

Figure 4.3 illustrates how the requirements cover all important parts of our machine model (cf., Figure 4.2), being the functional correctness of adaptation rules (Q3, Q1), their impact on the system and its environment (Q2), the direct feedback for the modeler (Q4), the intuitive use of the approach by hiding complexity (i.e., the formal methods, Q5), and the support of both, generic and application-specific properties (Q6).

In the following, we will reflect these requirements on relevant related work and thereby, on the one hand, show how the different requirements can be fulfilled, and on the other hand, show why current approaches do not suffice.

Related Work Table 4.1 shows the requirements in the columns and the different approaches in the rows. In the following, we will describe for each requirement how it can be fulfilled by referring to the respective approaches. Additionally, we will describe how the requirement is fulfilled in our quality assurance approach for self-adaptive systems called QUAASY.

QR01: Check Adaptation Rules Adaptation rules may be checked either independently from progress rules (or at least explicitly), or adaptation

TABLE 4.1. Modeling Approaches compared to Requirements

Approaches	QR01	QR02	QR03	QR04	QR05	QR06
Bartels, 2011 [BK11]	×	○	✓	×	×	○
Cheng, 2006 [ZC06]	×	✓	✓	×	×	○
Fleurey, 2009 [FS09]	×	✓	✓	○	○	○
Adler, 2007 [ASSV07]	✓	✓	✓	○	×	✓
Nafz, 2011 [NSS+11]	×	✓	✓	×	×	○
Weyns, 2013 [GdlIW13]	✓	✓	✓	×	×	×
QUAASY [LE13]	✓	✓	✓	✓	✓	✓

rules may be checked implicitly by checking the entire system model. The approaches [BK11, ZC06, FS09] check whether the system behaves correctly under adaptation. They do not check adaptation rules explicitly, e.g. for stability. Same applies to Nafz et al. [NSS+11] who do not check the adaptation rules per se but the correctness of the complete system. In contrast, Adler et al. [ASSV07] check properties for adaptation rules, such as stability, explicitly. Due to consequently separating concerns, Weyns et al. [GdlIW13] allow the independent check of adaptation rules. QUAASY takes the same approach as Adler et al. [ASSV07] and allow to check adaptation rules explicitly for generic properties such as deadlock freedom, stability, etc.

QR02: Check Progress Rules In contrast to checking adaptation rules ($\mathcal{A}$) only, the progress rules ($\mathcal{P}$) may be checked separately, too, especially for the effects of self-adaptation. Some approaches [BK11, GdlIW13] check progress rules implicitly by checking the entire system for deadlocks and livelocks. Likewise, Cheng et al. [ZC06] use temporal logic formulas to check the correct behavior of the system. The effect of adaptation is thus checked only implicitly. Fleurey et al. [FS09] allow to specify constraints that may be checked with the Alloy Solver. These constraints may be targeted at the systems progress and the effects of adaptation. Adler et al. [ASSV07] allow to specify and check application-specific properties for the functional behavior of the system, i.e. the system progress. Just like the approach of Fleurey et al. [FS09] these properties may check the effect of adaptation. Nafz et al. [NSS+11] use a theorem prover to show functional correctness of the system, again potentially including effects of adaptation. Besides invariants that constraint the pure system progress, QUAASY explicitly checks for the effects of adaptation by the use of specific validation functions. For instance, these validation functions check whether a resulting model still is well-formed and a proper instance of its meta model. The concrete validation functions may be extended by the language designer or even the modeler himself.

QR03: Check Functional Quality We require our approach to check func-

tional quality. Non-functional quality assurance is not the focus of this theses. All presented approaches [BK11, ZC06, FS09, ASSV07, NSS+11, GdlIW13] only allow the verification of functional correctness, e.g. by using a (CSP) model checker, a solver (Alloy), or a theorem prover (KIV). QUAASY also focuses on functional properties. Therefore, we use the model checker Groove. However, the Adaptation View Model can be used for performance analysis as well. We are currently working on combining the QUAASY approach with SimuLizar [BLB13].

QR04: Direct Modeler Feedback Direct feedback is crucial when designing large software systems. Therefore, it is important to yield high performance concerning the used time. The different approaches handle feedback differently. Most of the approaches might provide direct feedback to the modeler, however, a corresponding approach is not presented in the related publications. In the CSP-based approach [BK11], the modeler receives the CSP prover's results that he has to interpret after the complete model has been created. The modeler has to create all models before any result can be generated. An approach for direct feedback provisioning has not been given explicitly. Same applies to the approach of Weyns er al. [GdlIW13]. Fleurey et al. [FS09] do not define a process to provide direct feedback, either. However, the Alloy Solver easily could be triggered during modeling. Since Adler et al. [ASSV07] use averest[1], direct feedback would be possible theoretically, too. However, the authors do not show a concrete approach towards direct feedback. Moreover, application-specific properties have to be specified manually using different languages (temporal logic) which further hardens the provisioning of direct feedback. Nafz et al. [NSS+11] cannot provide direct feedback since the models have to be proven using the manual theorem proving using the tool KIV. QUAASY provides concepts for providing direct and immediate feedback for the user. The models are translated immediately after saving and (if executable) can be checked in the background to provide the user with feedback during modeling. Special performance optimizations allow to provide feedback in a timely fashion.

QR05: No Formal Knowledge Needed An important requirement is the intuitiveness of the approach to enable non-experts the use of the quality assurance techniques. Unfortunately, most of the approaches require more or less deep knowledge in formal methods. For instance, for using the CSP approach by Bartels et al. [BK11], the modeler must have good knowledge of CSP. With Cheng et al.'s approach [ZC06], the modeler must know petri nets and temporal logics. With Adler et al.'s ap-

[1] http://www.averest.org

proach [ASSV07], for system modeling no formal knowledge is needed. However, the application-specific constraints have to be specified in temporal logic by the modeler. Using the approach by Nafz at al. [NSS+11], the definition of models and constraints is given in UML and OCL. The authors describe that common off-the-shelf constraint solvers can be used to show correctness. However, the translation into the corresponding language or the use of the theorem prover KIV requires formal knowledge. The most intuitive approach to use is the one proposed by Fleurey et al. [FS09] Their approach requires knowledge in simple logic expressions, only. The translation into Alloy is done automatically, thus no deep knowledge of formal methods is needed. QUAASY completely hides the used model checking from the user (i.e., tool-encapsulated or transparent model checking). Application-specific properties are given in a simple OCL-like language. Generic properties do not have to be specified by the modeler but come *pre-packaged*.

QR06: Support Generic and Application-Specific Constraints As described above, we distinguish generic properties from application-specific constraints. Both kinds of constraints are useful and should be supported by quality assurance approaches. While the CSP-based approach [BK11], only supports generic properties, the approaches by Cheng et al. [ZC06], Fleurey et al. [FS09], and Nafz et al. [NSS+11] only describe how to check application-specific properties. Solely the approaches by Adler et al. [ASSV07] and Weyns et al. [GdlIW13] support generic as well as application-specific constraints. QUAASY supports generic properties, such as stability, as well as application-specific constraints by enabling the definition of invariants on model level using an OCL-like language.

All in all, there is not an existing approach that fulfills all of our requirements. Therefore, in the next section, we will present our approach QUAASY that is directed towards fulfilling the requirements and build upon the presented ACML language.

4.2 QUALITY ASSURANCE FOR ADAPTIVE SYSTEMS (QUAASY)

QUAASY is our QUality Assurance approach for Adaptive SYstems that fulfills the requirements listed above. The approach relies on graph transformation based model checking that, however, is hidden from the user. The overall goal

is to provide a tool supported framework for modeling self-adaptive systems with direct feedback concerning generic and application-specific properties by checking the modeled system live in the background. QUAASY checks the adaptation rules as well as the modeled system itself.

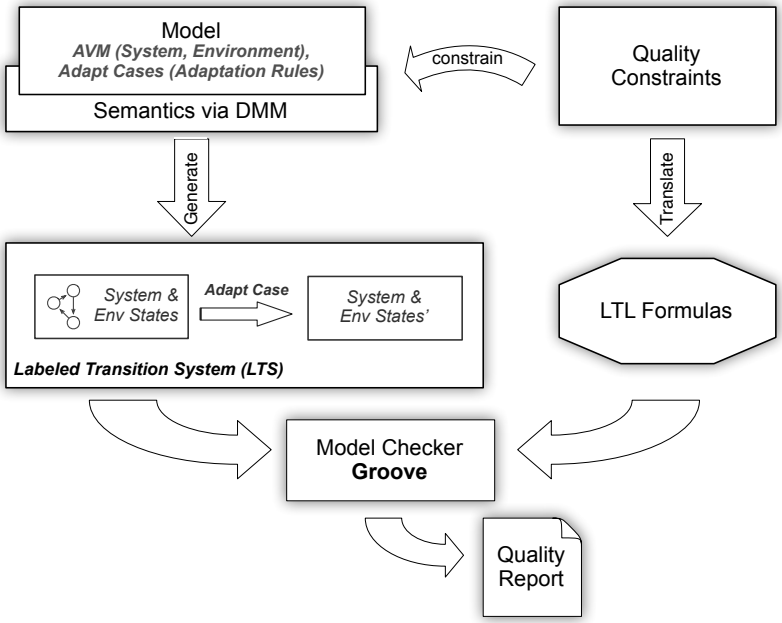


Figure 4.4. The QUAASY Approach

using DMM for specifying semantics

The basic idea of the quality assurance task is depicted in Figure 4.4. First, using the presented language (i.e., ACML including the Adapt Case Model and the Adaptation View Model introduced in Chapter 3) a concrete self-adaptive system is modeled. The ACML semantics have been defined by means of Dynamic Meta Modeling (DMM) [EHHS00], a rule-based semantics specification technique. A DMM specification together with an initial adaptation system model give rise to a labeled transition system (LTS) describing the complete system's semantics. The LTS is computed by using the state corresponding to the model as the initial state. On that state, all matching DMM rules are applied, leading to new states. This is done until no new states are found. The LTS has concrete model instances as states exposing concrete model objects and assigned attributes. Further, each state contains the models (i.e., system type) themselves since they might change over time by adaptation. The transitions are given by the applications of DMM rules. DMM rules describe the semantics of the model using graph transformation rules, e.g., a rule might move a token, change an attribute value, or execute an adaptation action. The transition system can then be analyzed using model checking techniques [ESW07]. Finally, we formulate our quality properties as temporal logic formulas, which are model checked on the computed LTS using the Groove tool set [Ren03]. If

one of our properties is violated by the model under consideration, the model checker provides a counter example which can be used by the modeler to fix the system model.

DISTINGUISH BETWEEN ADAPTATION $\mathcal{A}$ AND SYSTEM & ENVIRONMENT PROGRESS $\mathcal{P}$

The DMM semantics rules are divided according to the two classes of behavior: adaptation DMM rules ($\mathcal{A}$) and (system & environment) progress DMM rules ($\mathcal{P}$). The semantics of the Adaptation View Model (*progress DMM rules*) describe how the system and its environment change over time. For instance, open variables from the environment will be changed within their boundaries. The semantics for the Adapt Case Model (adaptation DMM rules) describe the reaction of Adapt Cases to a change in the Adaptation View Model and the effect of applying adaptation actions.

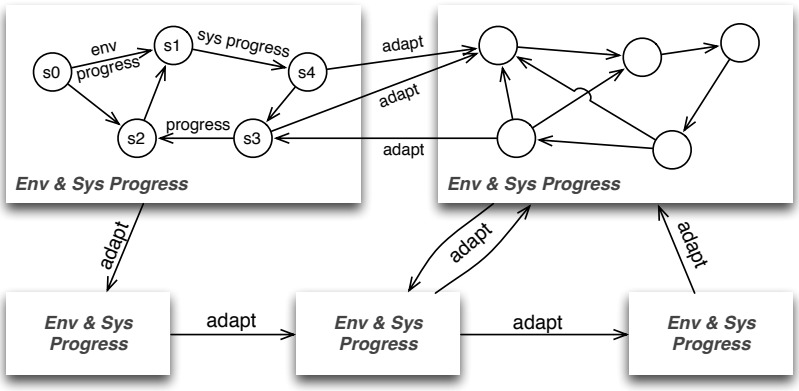


FIGURE 4.5. QUAASY Labeled Transition System

Figure 4.5 sketches a labeled transition system. The boxes are only to visually group several states. The upper left box describes a system and its progress without any adaptation. The transition from s_0 to s_1 reflects a change in the modeled environment. The transition from state s_1 to state s_4 reflects a change of the system. The complete upper left box represents a labeled transition system that would have been generated without adaptation rules. If an adaptation rule applies, it takes the system into *another box* (cf. different boxes in Figure 4.5). The environment and system progress is described by the Adaptation View Model and simulated by corresponding *progress DMM rules*. The system's adaptation is described by the Adapt Case Model and simulated by *adaptation DMM rules*, i.e., rules that described the execution of an Adapt Case.

See Figure 4.6 for an illustration of the LTS transitions. Essentially, DMM rules are graph transformation rules that are typed over a language's meta model and transform a language's models to simulate their semantics conformant change. For instance, the set of *progress DMM rules* that is depicted on the figure's left side describes how an environment sensor's property may increase: $value' := value + step$ if $value + step < max$. The application of this *progress*

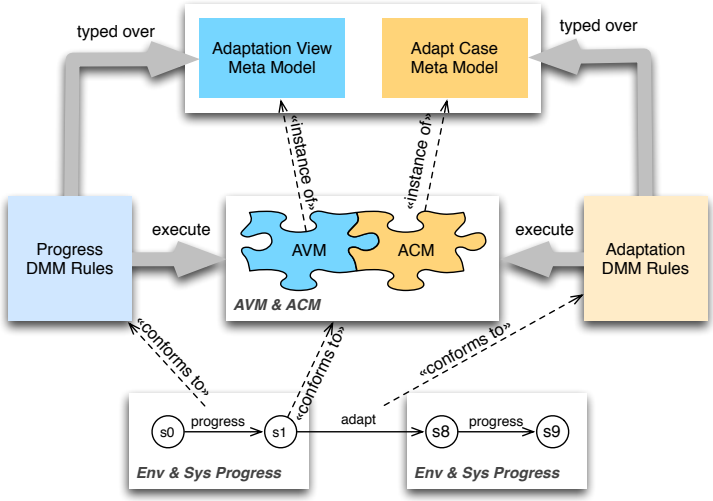


Figure 4.6.
LTS Transitions and DMM Rules

DMM rule corresponds to a transition within the system's LTS. On the other hand, the *adaptation DMM rules* describe how a concrete Adapt Case may manipulate the system (cf. Chapter 3 for the description of Adapt Cases and their semantics).

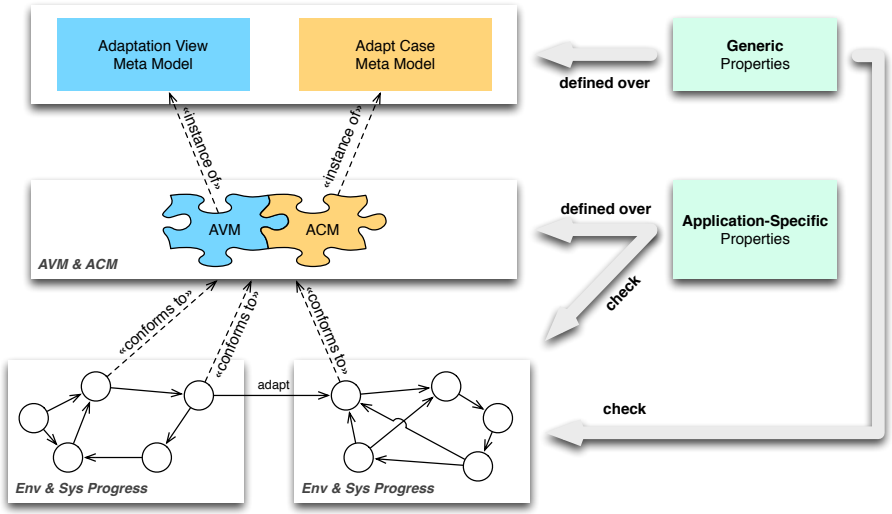


Figure 4.7.
Generic and Application-Specific Properties

Having formal semantics for all models allows the automated quality analysis with respect to the adaptive behavior using model checking techniques [ESW07]. As described before, different kinds of quality properties exist, being application-specific properties and generic properties. These different kinds are treated differently in the QUAASY approach. See Figure 4.7 for a description. As stated above, the states of the LTS are (ontological) instances of the Adaptation View Model and Adapt Case Model which in turn are (linguistic) instances of their respective meta models. Generic properties are defined

over concepts of the two meta models, that is, they can be checked for every possible Adaptation View Model and Adapt Case Model. Although generic properties are defined over concepts of the meta model level, they are checked on the LTS, i.e., on instance level. In contrast, application-specific properties are specified over specific AVMs or ACMs. These properties refer to concrete elements that are modeled with the AVM or ACM. They are checked on instance level, too.

In Section 4.2.1, we exemplary show some DMM semantics definitions. In Section 4.2.2, we provide the necessary temporal logic formulas to actually check the functional quality properties that have been described in Section 4.1.1. In Section 4.2.3, we describe how to provide the user with feedback that is easy to understand for a non-expert concerning formal methods.

4.2.1 Semantics Definition for the ACML

For the semantics definition of both the Adaptation View Model and the Adapt Case Model, we use Dynamic Meta Modeling (DMM) that has been described in Section 2.4. In this section, we will describe the semantics by the use of some representative examples. The example rules cover all relevant parts of the AVM and the ACM and provide an impression of how the UML activity semantic is defined. More precisely, we will present the following DMM rules:

- Activity Semantics
  - Token Flow Rules
- Specific Adaptation Actions
  - Write Property Action
  - Add Action to Activity
- Adapt Case Model (ACM)
  - Start Monitors
  - Trigger Adaptation (CallAdaptationActivity)
- Adaptation View Model (AVM)
  - Simulate Open Interval Properties
  - Generate & Fire Signals

Since both the AVM and the ACM heavily rely on UML activities (e.g., Monitoring & Adaptation Activities), we will first start with showing two rules that define the UML token-offer flow. Next, we will show two adaptation specific actions that allow to (a) adapt/change adaptation interface properties, and (b), add an action to an activity. Next, we will describe how Adapt Cases start their monitors and how monitors may trigger an adaptation activity using the corresponding action. Finally, we will show how the Adaptation View Model simulates open properties (i.e., properties that do not have a specification but only boundaries and a step size or a set of states without attached state machine) and how environment signals are generated and fired.

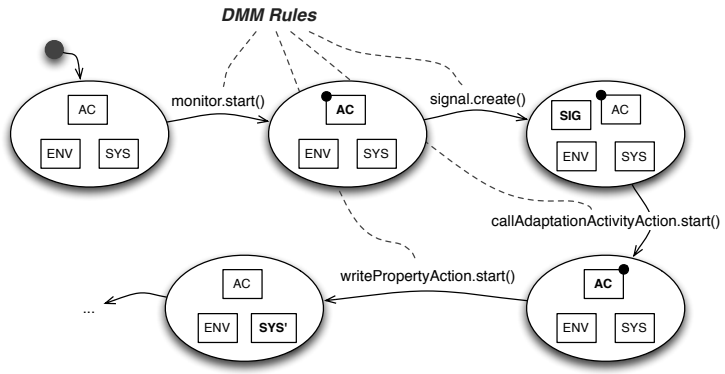


Figure 4.8. Sketched Transition System shown DMM Rule Applications

See Figure 4.8 for a sketched transition system that shows the application of DMM rules. The states illustrate instances of the modeled self-adaptive system. The transitions correspond to DMM rule applications. The transition system is not complete and intermediate steps have been hidden. In a first step, the `monitor.start()` rule activates the monitor (i.e., the Adapt Case). Next, a DMM rule creates a signal instance which, in the next step, is consumed by the Adapt Case's monitor together with triggering the adaptation activity. The activity's `WritePropertyAction` is executed by the corresponding DMM rule and the system `SYS` is adapted to `SYS'`. In the following sub sections, we will look into the DMM rules in detail.

Activity Semantics

The activity semantics conform to the UML specification [Obj10b]. The UML describes a token-offer semantics that extends the token-flow semantics that is known from petri-nets. Details of the semantics are given in [ESW07]. In the following two simple DMM rules are shown that illustrate the mechanisms of UML activities.

FIGURE 4.9.
`action.start()#`:
Starting a UML Action

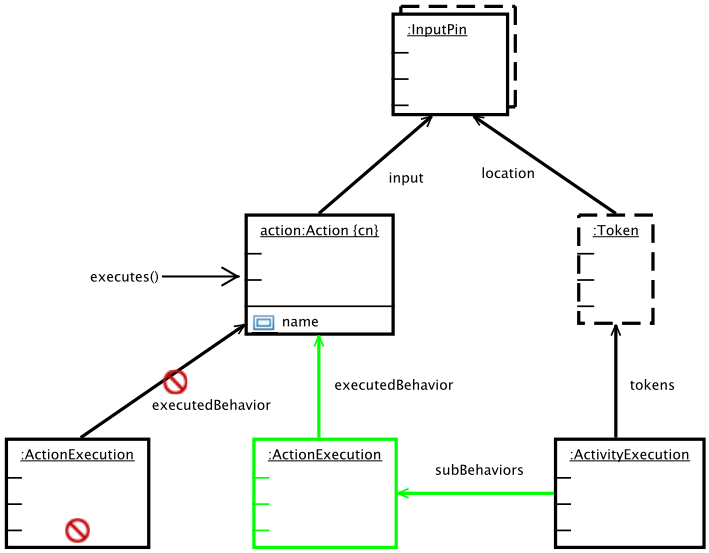


Figure 4.9 describes the DMM rule that starts the execution of a UML action. If the action has a token on every input pin and does not have an `ActionExecution`, yet, a new `ActionExecution` is created and the `action.executes()` rule is invoked. This invoked rule is overloaded for each specific UML action and actually performs the action. An example will be shown in the next section.

FIGURE 4.10.
`action.start()#`:
Executing a UML Action

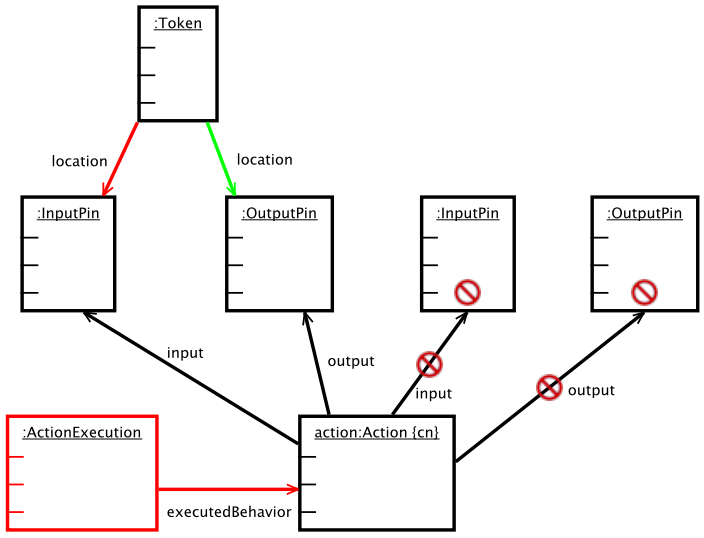


Figure 4.10 is executed if an action has an `ActionExecution` node and a token in its input node. This token is moved to its output nodes and the `ActionExecution` is deleted. Other rules take care of the transportation of the token to the next action, eventually traversing several control nodes, such as decision or fork nodes.

Specific Adaptation Action

Let us investigate the DMM rule which specifies the semantics of a adaptation-specific action, called `WritePropertyAction`, which is shown in Figure 4.11. As described in Section 3.3.2, this action simply writes the value of an `ACMLIntervalProperty`. This is denoted with the assignment *value'* := *wpa.value* which is attached to the `LiteralInteger`. The value is taken from the `WritePropertyAction`. The `InstanceSpecification` represents an object of some UML type. In the case of and `ACMLIntervalProperty`, the type is described by a sensor or effector.

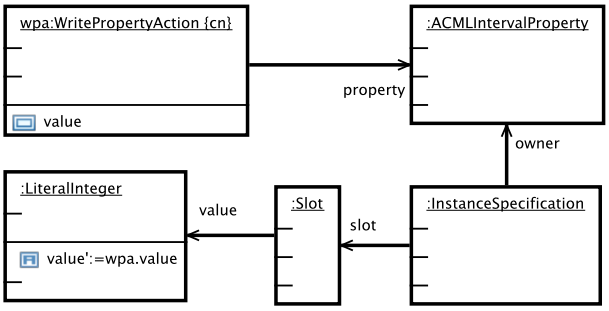


Figure 4.11. `wpa.execute()#`: DMM Rule to execute WritePropertyAction

Figure 4.12 shows the semantics of another adaptation-specific action. The action allows the insertion of a new UML action within an existing control flow. The original control flow is attached to the newly created action which in turn is connected to the original target action. The action is configured using the set of parameters that are given by the `AddActionAction`. The `SpecifiedAction` is a placeholder that is replaced by the create SmallStepRule. This rule is overloaded for each particular type of action.

Adapt Case Model (ACM)

Monitors are constantly running. Since monitors are specific UML activities, they may use activity features to influence the type of activation. For instance, a monitoring activity may define an `AcceptEventAction` that listens for the occurrence of a specific event such as signals or variable changes. Alternatively, the monitoring activity could use the `TimeEvent` to repeatedly start some monitoring actions. If the monitor triggers an adaptation activity and no other tokens are alive, it is terminated and restarted if the adaptation activity finished. See Figure 4.13 for the corresponding DMM rule.

The DMM rule matches those Adapt Cases that have no running monitoring and adaptation activities and activates the corresponding monitor by attach-

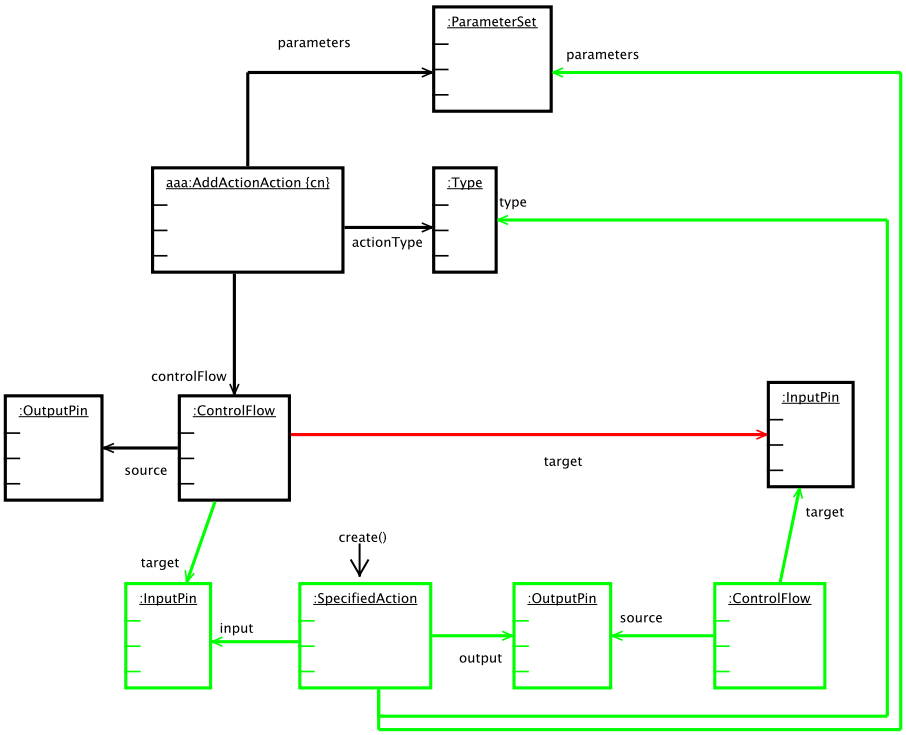


FIGURE 4.12.
DMM Rule that adds an Action into an Activity

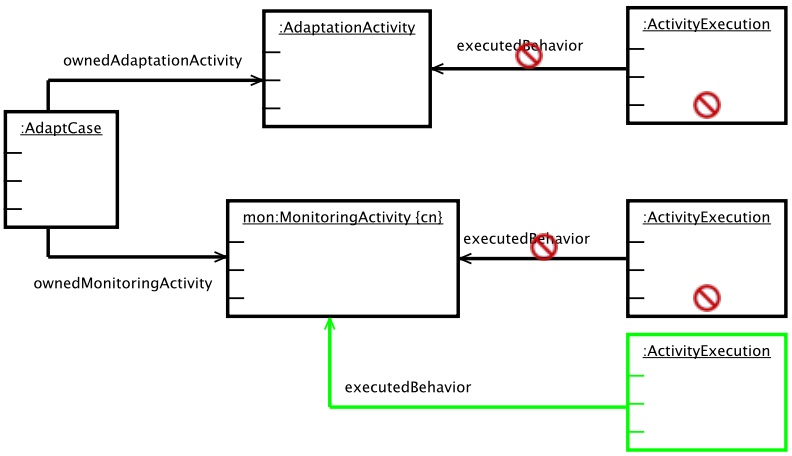


FIGURE 4.13.
DMM Rule that starts Monitors

ing an `ActivityExecution`. The `ActivityExecution` is matched by the activity semantics which execute the corresponding activity.

The next DMM rule describes the hand-over between monitoring and adaptation activity. As described in Section 3.3.2, we introduced a specialized action for that purpose. Hence, Figure 4.14 describes the semantics of the `CallAdaptationActivityAction`. The rule overloads the `action.executes()` SmallStepRule and simply creates an `ActionExecution` for the corresponding `AdaptationActivity`.

Adaptation View Model (AVM)

The Adaptation View Model heavily reuses the semantics of UML class diagrams and activities. That is, the semantics of classes, interfaces, etc. (e.g., semantics of cardinalities) remain unchanged. There are, however, additional semantic rules that define the behavior of new notational elements. Examples include the definition of open properties, i.e. properties that have a lower and upper bound that constrain the concrete value. These properties are meant to simulate the uncertainty of the environment or may even be used to underspecify the system. Therefore, the semantic rules have to increase and decrease the value of these properties.

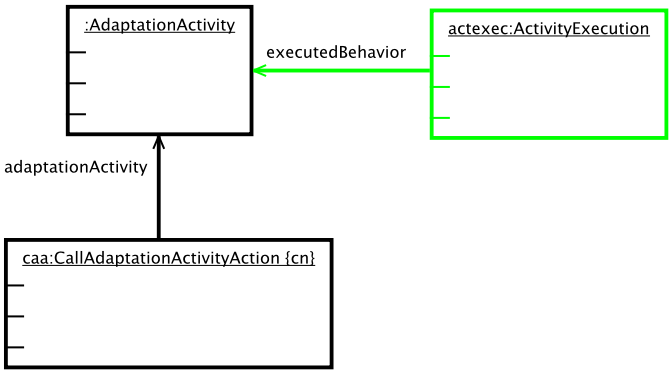
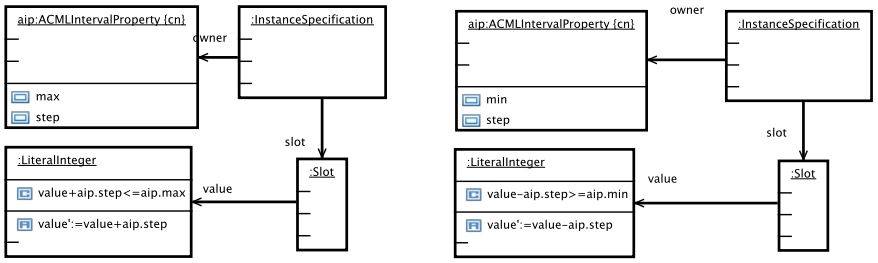


Figure 4.14. `ac.callAdaptationActivity()`: Executing CallAdaptationActivity Action

Figure 4.15 shows the rules that increase and decrease the value of an `ACMLIntervalProperty`. Therefore, they match the corresponding `InstanceSpecification` and increase/decrease its value by the step size if it is still in the allowed range given by the min and max values. Together, the rules simulate and arbitrary increase and decrease open properties in the environment or system.

Finally, Figure 4.16 shows the DMM rule that generates signals. The signals that may be thrown are specified in the Adaptation View Model for environ-

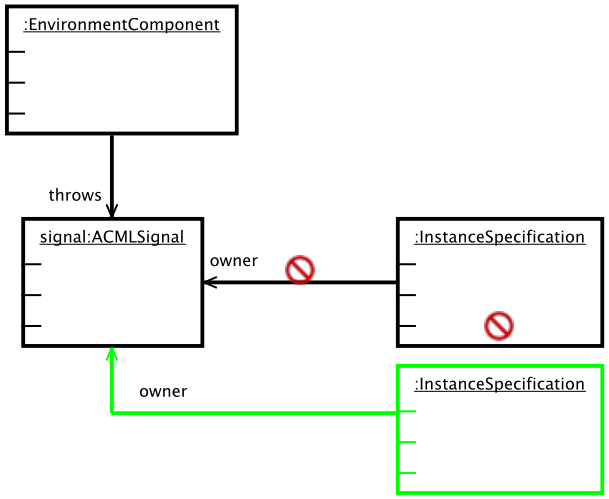
FIGURE 4.15.
DMM Rules that simulate an IntervalProperty



(a) Increase of open Value (b) Decrease of open Value

ment components. The DMM rule creates `InstanceSpecifications` for the signal if not yet existing. This `InstanceSpecification` is *consumed* by the DMM rule that accepts a signal event.

FIGURE 4.16.
DMM Rule that creates a new Environment Signal



In this section, we presented a few representative DMM semantic rules that describe (a) how the Adaptation View Model is simulated (*progress*) to generate a state space, and (b), how the Adapt Case Model is executed (*adapt*) to monitor and adapt the Adaptation View Model. In the next section, we will describe how the quality properties from Section 4.1.1 are formalized and checked using the GROOVE model checker [Ren03].

4.2.2 Quality Property Formalization

In order to reach our overall goal of an integrated modeling and analysis workbench for self-adaptive systems, we need to employ automatic analysis of self-adaptive system models. That is, we need to describe how quality properties of self-adaptive systems can be checked within our approach.

We recapture the two behavior classes of self-adaptive software system, system and environment behavior and adaptation behavior. These two behaviors are described by DMM rules as shown in the last section. We will use the two sets $\mathcal{P}$ and $\mathcal{A}$ as representatives for the corresponding DMM rule sets. With these two sets of DMM rules, we can distinguish between standard operation of the system and its environment, and the adaptation of the system within our temporal logic formulas.

USING THE GROOVE MODEL CHECKER

Since all properties are checked via model checking, we further define the labeled transition system (LTS). Let *LTS* be the LTS computed from the set of DMM rules (see Section 4.2.1) applied to the Adaptation View Model and Adapt Case Model. Further, recall from Section 2.4 that Groove gives rise to labeled transition systems where states are typed graphs and transitions are applications of graph transformation rules; the transitions 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 $\mathsf{A} := \vee_{\alpha \in \mathcal{A}} \alpha$. Given a state s, A will be true iff at least one of the adaptation rules can be applied to s. Same applies to progress rules $p \in \mathcal{P}$. Here, we define $\mathsf{P} := \vee_{p \in \mathcal{P}}(p)$.

In the following sub sections, we will formally describe the properties introduced in Section 4.1.1 using our notion described in Section 3.2.2. Additionally, we will define the corresponding temporal logic formula if it significantly differs from the first formalization. Please read on for further details.

Errors

Recall that errors are checked in each possible system state. Therefore, using the notion introduced in Section 3.2.2, a validation checker is specified as fol-

lows:

$$\texttt{valid}_{err} :: \mathbf{VR} \times \mathbf{AVM} \times \mathbf{ACM} \rightarrow \{0, 1\} \tag{4.1}$$

The set of validation rules **VR** is checking a system and environment state (**AVM**) as well as the set of adaptation rules (**ACM**). The set of validation rules can be user defined or static, i.e., pre-defined. Since we have another property class which describes user-defined validation rules (see specific safety and liveness properties), we include only static rules in the set of validation rules.

The set of validation rules **VR** consists of DMM property rules that, e.g., check whether an initial node is included in an activity diagram.

Definition 1 ***(Errors)*** *A system definition is free of (pre-defined) errors (i.e., successfully validated) iff in all states the system may be in, none of the validation rules out of* **VR** *applies.*

This definition requires the validation rules to be the negation of the desired property. The LTL property to check errors is defined as follows:

$$\forall r \in \mathbf{VR} : \Box \neg r \tag{4.2}$$

The formula is true if for all rules in **VR** there is no trace in the LTS where r holds in any state.

Conflicts

A set of rules does not contain conflicts iff whenever a rule α_1 is running and a second rule α_2 starts, both rules are able to finish their execution.

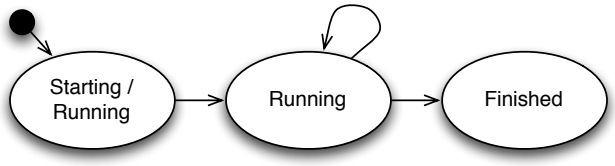


FIGURE 4.17. Lifecycle of an Adaptation or Progress Rule r ($r \in \mathcal{A} \cup \mathcal{P}$)

To formally describe this property, we introduce the notion of a rule's state. That is, each rule out of $\mathcal{A} \cup \mathcal{P}$ has an own lifecycle that is depicted in Figure 4.17. In its first state, a rule is starting and running. Next, the rule may run an arbitrary amount of time. Finally, the rule enters its finished state. To describe the state of an adaptation rule, we define three functions (`starting`,

`running`, `finished`), that take a rule $r \in \mathcal{A} \cup \mathcal{P}$ as input and return true if that rule is in the corresponding state. In a more detailed version of our machine model from Figure 4.2, each transition would be divided into several small step transitions with intermediate states. As a consequence, parallel execution of Adapt Cases, the system processes, etc. is actually performed interleaved at the level of these small step transitions. Now, we define a set of rules to be free of conflicts iff the following holds:

Definition 2 ***(Conflict)*** *A set of rules is free of conflicts iff for all system states when one of two arbitrary rules in $\mathcal{A}$ is starting while another one is already running, both rules are able to finish in some future system states.*

$$\begin{aligned} \forall \alpha_i, \alpha_k \in \mathcal{A} : \Box \ & \mathit{running}(\alpha_i) \wedge \mathit{starting}(\alpha_k) \\ & \rightarrow \Diamond \ \mathit{finished}(\alpha_i) \wedge \Diamond \ \mathit{finished}(\alpha_k) \end{aligned} \tag{4.3}$$

The LTL formula has to be checked for every pair of adaptation rules in $\mathcal{A}$. It is true iff the LTS does not contain any trace that contains a state where a_i is running and a_k is starting and at least one of the two adaptation rules is not finished at some future time. The three functions `starting`, `running`, and `finished` are expressed using DMM property rules. As such, the above formula can directly be used with our model checker.

Termination

Similarly, termination is defined as follows.

Definition 3 ***(Termination)*** *For the termination property to be true, every single behavior that is started at some point in time has to have an option to finish at some future point in time.*

We use CTL to formulate this property formally:

$$\forall r \in \mathcal{A} \cup \mathcal{P} : \mathbf{AG}\ \mathtt{starting}(r) \rightarrow \mathbf{EF}\ \mathtt{finished}(r) \tag{4.4}$$

The formula is true if for every state in the LTS, there is a rule that is currently starting, then there is a reachable state where the rule is finished.

Stability

For the check with a model checker, we define two different types of stability, namely the stability of single rules and the stability of the complete rule set.

Definition 4 ***(Rule Stability)*** *A single adaptation rule $\alpha \in \mathcal{A}$ is* stable *if the LTS does not contain paths such that α is applied infinitely often, but no other rule $p \in \mathcal{P}$ or $\alpha' \in \mathcal{A}$ with $\alpha \neq \alpha'$ is applied in between.*

$$\neg \exists \alpha \in \mathcal{A}, \forall p \in \mathcal{P} : \Box \; \mathit{running}(\alpha) \wedge \neg p \tag{4.5}$$

If a single rule is unstable, it might lead to situations where this rule is applied over and over again, e.g. continuously increasing some value of the system's state. The above definition covers this situation. Similar, a set of adaptation rules is stable according to the following definition.

Definition 5 ***(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 $p \in \mathcal{P}$ is applied in between.*

$$\neg \exists \alpha \subseteq \mathcal{A}, \forall p \in \mathcal{P} : \Box \; \mathit{running}(\alpha) \wedge \neg p \tag{4.6}$$

The second definition captures more complex, but still problematic situations such as the one presented above. The function `running` applied to a set is defined to return the disjunction of the contained elements. Note that if the rule set $\mathcal{A}$ is stable, it immediately follows that each rule $\alpha \in \mathcal{A}$ is stable.

Let us now formulate the LTL property used to verify rule stability. Let α be an arbitrary rule of our set of adaptation rules $\mathcal{A}$. α is stable if the following temporal logic formula holds for our LTS:

$$\neg \Diamond \Box \, \alpha \tag{4.7}$$

The interpretation of the formula is straight-forward: it is true if there is no trace such that rule α is always applied from one point in time on, realizing our requirements formulated in Definition 4.

We now turn to the LTL property used to verify rule set stability. We define A to be an arbitrary set of adaptation rules α that are connected with logical

ors: $A = \alpha_1 \vee \alpha_2 \vee \ldots \alpha_n$. A set of adaptation rules A is stable if the following formula holds for our LTS:

$$\neg \Diamond \Box A \tag{4.8}$$

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 A are applied, and therefore realizes the requirements of Definition 5.

Specific Liveness & Safety Properties

Specific safety and liveness properties are specified within the Adaptation View Model. For checking these properties, we define the $\texttt{valid}_{spec}$ function to take as input only the **AVM**:

$$\texttt{valid}_{spec} :: \textbf{AVM} \rightarrow \{0, 1\} \tag{4.9}$$

Properties may either be safety expressions that constrain the state of the system or liveness execution traces that require the system state space to include specific traces for particular specified behavior. Liveness properties are specified using ACMLConstraints in the Adaptation View Model. These constraints contain generic traces specified using the trace language as described in [Sol13]. Basically, the generic traces are process patterns that are transformed to LTL expressions f. Safety properties are specified using constraints, too. However, instead of traces, these constraints contain expressions e that have to evaluate to true to be fulfilled. Therefore, the expressions e are encoded in so-called *safe rules* $\mathcal{SR}$ using DMM property rules.

Definition 6 ***(Specific Liveness)*** *An adaptive system model is specifically lively if its LTS models f, i.e., $LTS \models f$.*

For model checking specific liveness, the traces are simply transformed to LTL formulas using DMM capabilities.

Definition 7 ***(Specific Safety)*** *An adaptive system model is specifically safe if its LTS contains no state s such that any safe rule is not applicable to s.*

Let us now formulate the LTL property used to verify specific safety:

$$\Box \bigwedge_{sr \in \mathcal{SR}} sr \tag{4.10}$$

The formula's interpretation is as follows: It is true iff for all states s, the formula $\bigvee_{sr \in \mathcal{SR}}(sr)$ is true, i.e., all rules from $\mathcal{SR}$ are applicable to that state.

Deadlocks

A set of rules is defined to be deadlock free iff

$$\forall \alpha_i \in \mathcal{A}, \exists \alpha_k \in \mathcal{A}, p_k \in \mathcal{P} : \texttt{starting}(\alpha_i) \rightarrow \Box \Diamond \alpha_k \vee p_k \tag{4.11}$$

holds, that is, if after an adaptation rule is applied either another adaptation rule or any progress rule is applicable. For model checking, we slightly expand the definition of deadlocks as follows.

Definition 8 ***(Deadlock)*** *An adaptive system model contains a deadlock if its LTS contains at least one state s such that no adaptation rule α or progress rule p is applicable to any of the states reachable from s.*

The LTL property to verify the absence of deadlocks is defined as follows:

$$\Box \Diamond \left(\bigvee_{r \in \mathcal{A} \cup \mathcal{P}} r \right) \tag{4.12}$$

The formula's interpretation is as follows: It is true iff for all states, a state is reachable such that $\bigvee_{r \in \mathcal{A} \cup \mathcal{P}}(r)$ is true, i.e., at least one of the rules from $\mathcal{A}$ or $\mathcal{P}$ is applicable to that state. It is easy to see that this is exactly the definition of absence of deadlocks we have seen above. Note that this includes any deadlock the system may produce. However, since the input models are adaptation-related models, only, the formula verifies the creation of deadlocks caused by adaptation or adaptation-related processes.

Confluence

To describe the confluence property, let us define concrete system states to be $s_0, s_1, \ldots$. Further, let A be a set of adaptation rules $\alpha_0, \alpha_1, \ldots$ with $A \subseteq \mathcal{A}$. We denote the application of an adaptation rule α_0 to a state s_0 yielding a new state s_1 as $s_0 \xrightarrow{\alpha_0} s_1$. If there is a set of adaptation rules A_0 that transforms some system state s_0 to s_i, we denote this as $s_0 \xrightarrow{A_0} s_i$. Now, we can define the confluence property for adaptation rule sets.

A set of adaptation rules is confluent if the following holds.

Definition 9 ***(Confluence)*** *If there are two non-equal sets of adaptation rules A_0 and A_1 that can be applied to a system state s_0 to yield the system states s_i ($s_0 \xrightarrow{A_0} s_i$) and s_j ($s_0 \xrightarrow{A_1} s_j$), then there is another state s_k and two other sets A_2 and A_3 such that $s_i \xrightarrow{A_2} s_k$ and $s_j \xrightarrow{A_3} s_k$.*

$$\begin{aligned} \forall\, s_0, s_i, s_j, A_0, A_1 \wedge s_0 \xrightarrow{A_0} s_i \wedge s_0 \xrightarrow{A_1} s_j \\ \rightarrow \exists\, s_k, A_2, A_3 : s_i \xrightarrow{A_2} s_k \wedge s_j \xrightarrow{A_3} s_k \end{aligned} \tag{4.13}$$

To specify a temporal logical formula for confluence, we first have to clarify some prerequisites. First, we are using CTL^* for the definition of the formula. Second, we define an additional set of DMM rules used within the formula, the property rules ρ. Property rules do not change a particular system state but only check for the existence of of particular state properties. The DMM semantics for the ACML include property rule definitions such that each state can be identified uniquely via the application of a specific property rule. Thereby, we can check whether two states that are identified by the application of property rules are the same.

Using these prerequisites, we can define the CTL^* formula as follows.

$$\alpha_1 \wedge \alpha_2 \wedge \alpha_1 \neq \alpha_2 \rightarrow ((\alpha_1 \rightarrow \mathbf{X}\,\mathbf{EF}\,\rho_1) \wedge (\alpha_2 \rightarrow \mathbf{X}\,\mathbf{EF}\,\rho_2) \wedge \rho_1 = \rho_2) \tag{4.14}$$

The formula's interpretation is as follows: If there is a state where both α_1 and α_2 may be applied and these rules are different (branch into the confluence diamond), then after the application of α_1 and α_2 respectively, there are two states identified via ρ_1 and ρ_2 that are identical (join of the confluence diamond).

Determinism

Let $m(\alpha)$ be the monitor of the adaptation rule α. Formally described, the determinism property is defined as follows:

Definition 10 ***(Determinism)*** *An* ACML *model is deterministic, if there is no Adaptation View Model* AVM *where more than one Adapt Case α can be applied at the same time.*

$$\neg\exists\, \mathit{AVM}, \alpha_i, \alpha_j : \alpha_i \neq \alpha_j \wedge m(\alpha_i)(\mathit{AVM}) \wedge m(\alpha_j)(\mathit{AVM}) \tag{4.15}$$

The CTL formula for determinism is straight-forward. Since the adaptation rules α_i are only applied in the LTS if the monitoring result was positive, we can simply check for states where two adaptation rules match:

$$\mathbf{AG}\ \neg\ (\alpha_1 \wedge \alpha_2 \wedge \alpha_1 \neq \alpha_2) \tag{4.16}$$

Having defined all properties formally, we can now turn to the actual model checking and how we use the model checkers results to support the modeler.

4.2.3 Model Checking and User Feedback

SINGLE STEPS OF AN ADAPT CASE

Using the formulas defined above we can support the designer of self-adaptive systems with direct feedback on the quality of the model at hand. However, to use these formulas with the GROOVE model checker we have to map them to the syntax of concrete LTSs. That is, within the formulas presented above, we refer to single adaptation actions α while meaning the complete execution of an Adapt Case's adaptation activity **AdaptationActivity**, including several adaptation actions **Action**. However, in the LTS, the Adapt Case is separated into several transitions as indicated in the following:

$$\mathit{AdaptCase} = \mathit{Mon.start} \rightarrow ... \rightarrow \mathit{AA.start} \rightarrow \mathit{A.exec} \rightarrow ... \rightarrow \mathit{AA.finished}$$

An Adapt Case's execution is divided into the monitor's start following by several transitions for the analysis. If the monitor starts an adaptation activity, the latter executes several adaptation actions until it finally finishes and the overall Adapt Case is finished as well.

To translate these transition sequences (traces) into the "α-terms" (e.g., $\alpha_1, \alpha_2, \ldots$), we use property rules that reflect the adaptation rules' states that have been introduced in Figure 4.17 on Page 130.

$$AdaptCase\ \alpha = \underbrace{Mon.start \rightarrow ...}_{\varepsilon} \rightarrow \overbrace{\underbrace{AA.start}_{\texttt{starting}} \rightarrow A.exec \rightarrow ... \rightarrow \underbrace{AA.finished}_{\texttt{finished}}}^{\texttt{running}}$$

Recall that property rules are special DMM rules that may match and be applied to a particular state but do not perform any modifications. In addition, any DMM rule may expose attributes of the matched elements, so-called *emphasized attributes*. These attributes are appended to the transition label and thus can be matched in temporal logic formulas. If for instance the name of the matched Adapt Case is emphasized, the LTS contains rule applications of form `starting`($name_1$), `running`($name_1$), and `finished`($name_1$), where $name_1$ corresponds to the name of a particular existing Adapt Case and `starting`, `running`, and `finished` are the corresponding property rules. Note, that we are not interested in the monitoring execution, thus we do not create any property rule that capture the state of monitors hence ignoring monitors completely if we only check on property rules.

TRANSLATION TO TEMPORAL LOGIC

The temporal logic formulas defined in the previous section can now be translated as shown in the following.

$$\begin{aligned} &\textit{LTL:} && \alpha = \texttt{starting}(\alpha_{name}) \wedge (\texttt{running}(\alpha_{name})\ \mathbf{U}\ \texttt{finished}(\alpha_{name})) \\ &\textit{CTL}^{(*)}: && \alpha = \texttt{starting}(\alpha_{name}) \wedge \mathbf{A}\ (\texttt{running}(\alpha_{name})\ \mathbf{U}\ \texttt{finished}(\alpha_{name})) \end{aligned}$$

The formulas interpretation is as follows. A rule α is active if it has been started and as long it is running until it has been finished. The QUAASY tool does automatically create these formulas for the modeled system.

For the quality properties discussed above, the feedback process works as follows:

1. The modeling environment verifies whether the complete modeled system models the properties given in temporal logic. If this is the case, the ruleset does not give rise to any design flaw, and we are done.
2. Otherwise, QUAASY automatically identifies the subset of rules which cause the system to expose the undesired property (see below for details).
3. The system designer fixes the models.
4. Continue with Step 1.

In Step 2, we need to find the concrete rules which together make our system

behave undesired. That is because the model checker returns a counterexample (i.e., a trace of rule applications) that might not be the simplest possible counterexample making it sometimes difficult to interpret and find the error's source. Thus, we use another small algorithm that in case of an error may be executed to further drill down to the error's source. The algorithm is shown in Figure 4.18.

FIGURE 4.18. Algorithm to find erroneous rules

```
Let A_|i = {s ∈ 2^A with |s| = i}
For i = 1..|A|
   For each t ∈ A_|i
      if ¬ (t models properties) then
         Report t to be conflicting rules
```

Starting with sets of adaptation rules of size 1, we verify whether one of these rules behaves undesired. If this is not the case, we continue with the sets of size 2 and so on. Note that since we know that the set of adaptation rules itself behaves undesired, the algorithm will always result in a set of erroneous rules. However, in the worst case this will be the set of all adaptation rules. Note also that a set of adaptation rules might contain more than one set of erroneous rules. As such, after we have fixed the problematic rules, we will start over the verification process again. If the set of rules we identified and fixed in the first place was the only set of erroneous rules contained in our set of rules, and if we have properly fixed the problematic rules, the second check will report that the set of rules behaves as desired. Otherwise, we will continue fixing our rules and verifying the rule set until the rule set indeed behaves as desired.

USING THE DMM DEBUGGER TO ANIMATE THE COUNTEREXAMPLE

It remains to show how we support the designer of self-adaptive software systems 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 will use that information to help the modeler understand why the system model violates a quality property, and how the system model can be fixed. For this, the DMM simulation capabilities [BSE11] can be reused 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.

OPTIMIZING QUAASY 4.3

A problem that usually occurs when applying model checking, is state space explosion. On the one hand, in our approach the problem is less severe since the model checking is triggered automatically during modeling at design-time and performed in the background. Therefore, a fast response time is less important. On the other hand, a fast validation response would increase the positive user experience, and thus the usability of our approach. Hence, we took several actions to address the state space explosion problem.

THE PROBLEM OF STATE SPACE EXPLOSION

The reason for an exploding state space is the multiplication of possible system states. Therefore, we adjusted the semantics intelligently to reduce the possible states in the first validation iterations. For instance, in a first model checking run, we can abstract from parallel executed adaptations. A found error in this abstracted setting will occur in the non-abstracted setting as well and thus the validation is stopped in the first run and provides fast feedback for the user. In the following, we will describe the various actions that have been performed to encounter the state space explosion problem. Some of the findings presented in this section are based on a master thesis [Tha12].

4.3.1 Adapt Case Intermediate Language (ACIL)

One of the core reasons for large state spaces with the ACML is the use of the bloated UML semantics on the one hand, and the (unnecessary) multiplication of possible ACMLProperty states on the other hand. To alleviate these problems within QUAASY, the ACML is translated into an intermediate language which is optimized for model checking, the ACIL. Figure 4.19 sketches the setting. The translation comes with *no* loss of information compared to model checking the original ACML.

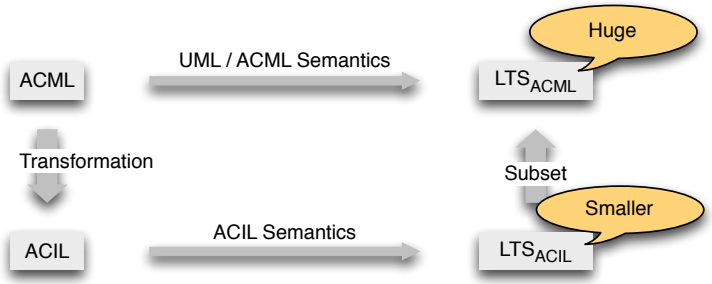


Figure 4.19. Adapt Case Intermediate Language Concept

In particular, the ACIL abstracts from the bloated token-offer-semantics as proposed by the UML and additionally separates the model into dependency groups that are model checked separately. Details are given in the following sections.

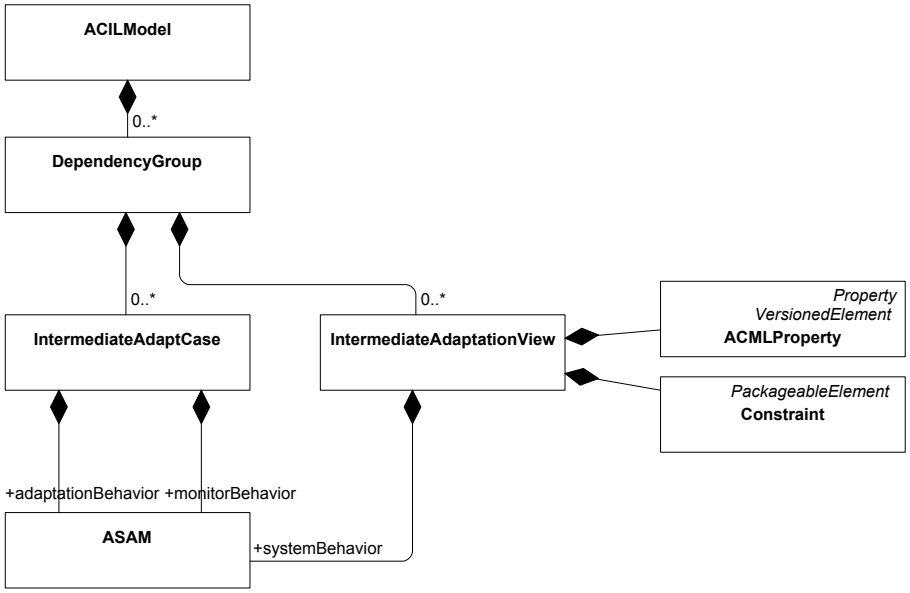


FIGURE 4.20. Adapt Case Intermediate Language (ACIL) Meta Model

Figure 4.20 shows the ACIL meta model. An ACIL model contains an arbitrary number of dependency groups. Dependency Groups in turn contain intermediate Adapt Cases (*IntermediateAdaptCase*) and an intermediate Adaptation View Model (*IntermediateAdaptationView*) the latter of which contains ACML-Properties and constraints. ACMLProperties are simulated during model checking, i.e., they are either computed if a sensor specification exists, or they are simulated with arbitrary values if they are open properties with range and step size. ACMLProperties and constraints are simply copied from the ACML model. Intermediate Adapt Cases contain Action Sequence Automata Models (ASAM) which are optimized automata that describe the behavior of monitoring and adaptation activities. Dependency groups and ASAMs are further detailed in the following.

Dependency Groups

Dependency groups separate the AVM and ACM into several single models.

Each dependency group contains a complete model without any dangling reference. In turn, each dependency group is as small as possible, i.e., there are no two elements in one of the groups that are not (transitively) linked to each other. Figure 4.21 illustrates the concept of dependency groups. *Adapt Case 1*

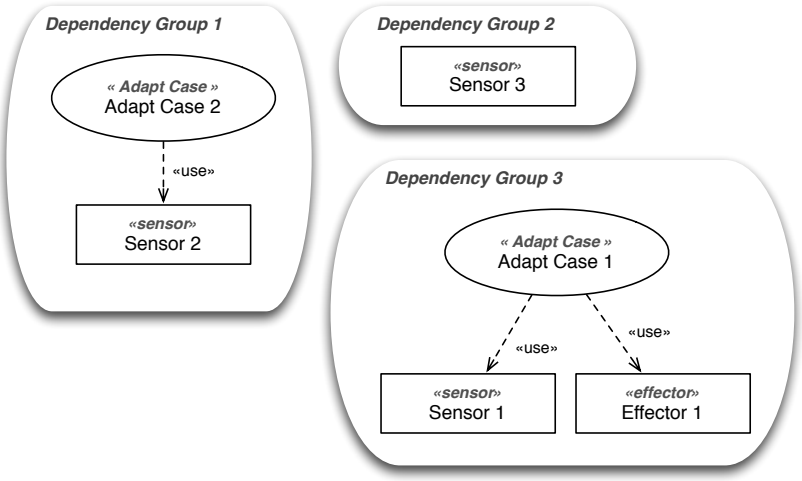


Figure 4.21. ACIL Dependency Groups

uses *Sensor 1* and *Effector 1*. None of the three elements are linked to *Sensor 2* or *Sensor 3*. *Sensor 3* is not used at all by any Adapt Case. In this sketch, we end up with three dependency groups which can be model checked separately.

The resulting benefit is depicted in Figure 4.22. While *Sensor 1* produces four possible states, *Sensor 2* produces 11. If both would be combined into one dependency group, we end up with 44 states in the LTS. Thus, if both sensors are independent from each other, it makes perfectly sense to check both model fragments separately.

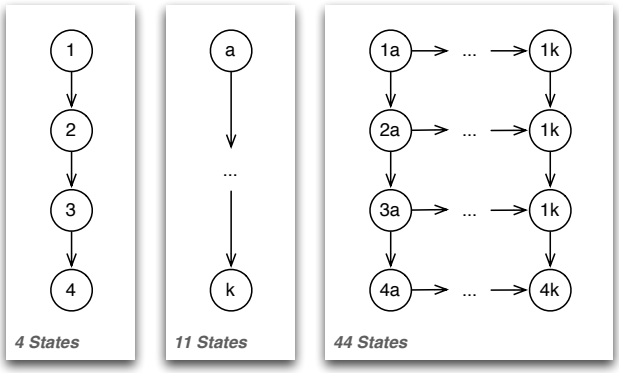


Figure 4.22. ACIL Dependency Groups Illustration

Action Sequence Automata Model

Action Sequence Automata Models (ASAM) are targeted at model checking. Since these models are optimized to be as small as possible, they might be hard to understand. As such, they are only a representation of adaptation behavior, such as monitoring or adaptation activities, but do not replace them.

ASAMs can be compared to byte code that is an optimized and executable representation of JAVA code.

To understand the need for ASAMs, let us look at the token-offer-semantics that has been used by the UML for activity diagrams (and thus for monitoring and adaptation activities) together with the corresponding LTS fragment.

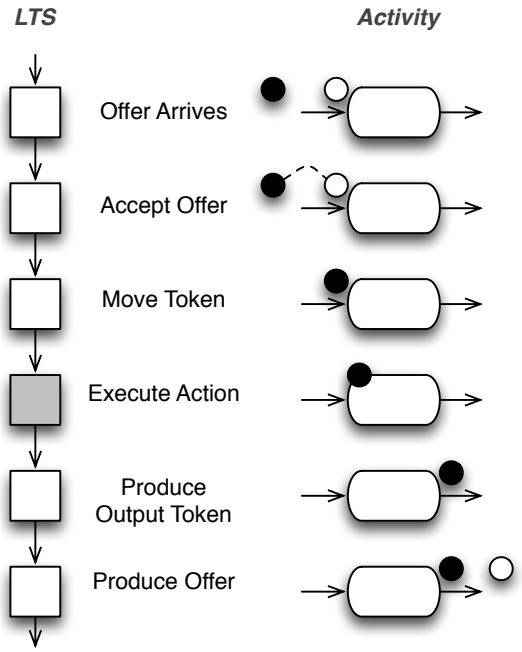


FIGURE 4.23.
UML Token Offer Semantics

Figure 4.23 shows the LTS on the left and the activity on the right. As shown in the first two and the last step, tokens are preceded by offers which can be accepted by nodes. If an offer is accepted, the token flows to the accepting node and is consumed which eventually includes the execution of an action. All in all, performing a single action produces five LTS states which might be even more complex if control nodes would be involved.

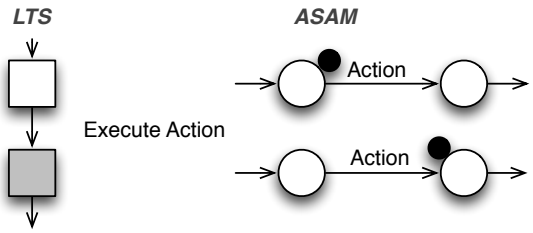


FIGURE 4.24.
ASAM Token Flow

In contrast, as depicted in Figure 4.24, ASAMs produce a single state which simply executes the action and passes the token.

The meta model for ASAMs is shown in Figure 4.25. An ASAM contains a UML state machine which describes all possible traces of actions that have

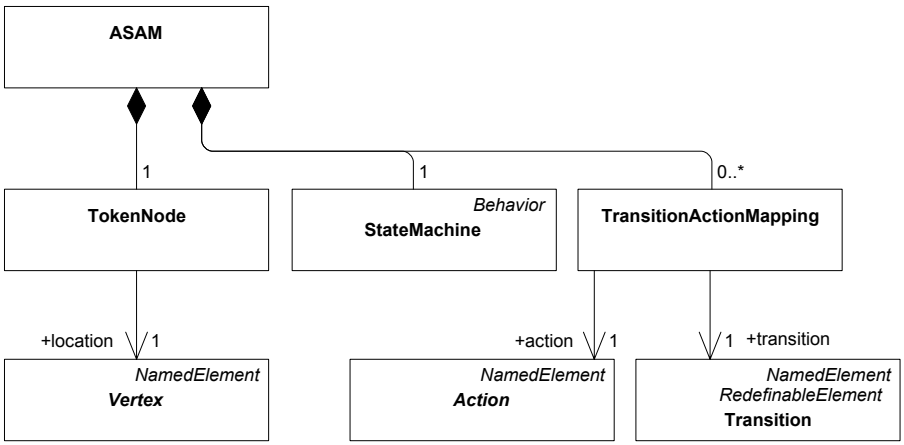


Figure 4.25. Action Sequence Automata Meta Model

been modeled with the corresponding activity. Further, the ASAM adds a token node that allows the description of state machine runtime states. Finally, a *TransitionActionMapping* maps a state machine transition to a UML action which, e.g., adapts the system model by calling an effector.

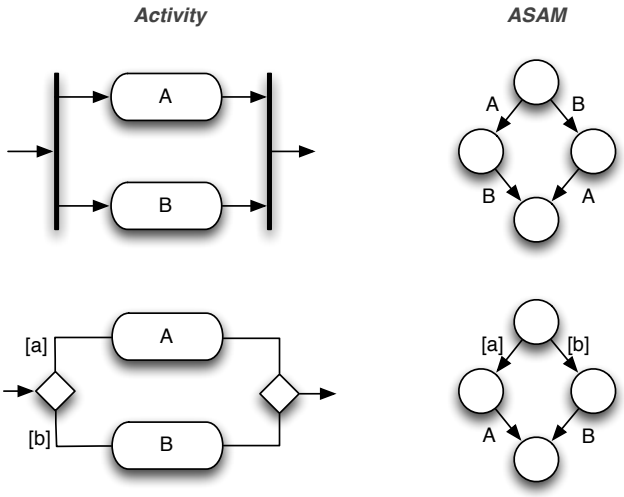


Figure 4.26. Comparison of Activities and ASAMs

Figure 4.26 shows example activities on the left and ASAMs on the right. Since the ASAM is very close to the LTS itself, the semantics are very simple. Two actions that are executed in parallel produce two paths in the state machine which described the two possible orders. If two actions are executed conditioned, two paths are generated as well, the first transition of each only can be taken if the corresponding condition evaluates to *true*.

An ASAM is constructed as described in Figure 4.27. First, the activity diagram is transformed into an LTS using DMM and an arbitrary activity semantics, e.g., as given in [ESW07]. Thereby, actions are not executed, but only the control and object flow is simulated. Next, the LTS is made more compact by

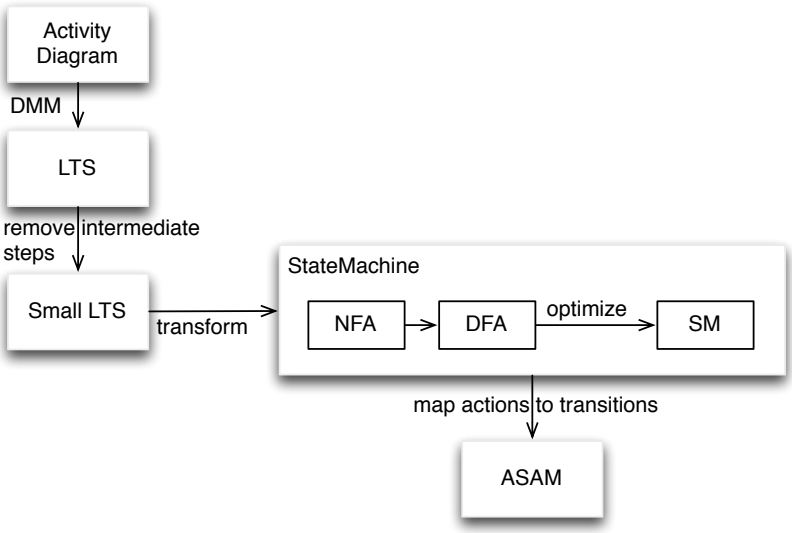


FIGURE 4.27. ASAM Construction Process

removing all intermediate steps that do not execute an action or evaluate conditions. Candidates include token-offer-semantic transitions. The removal is illustrated in Figure 4.28, the corresponding algorithm is straight-forward.

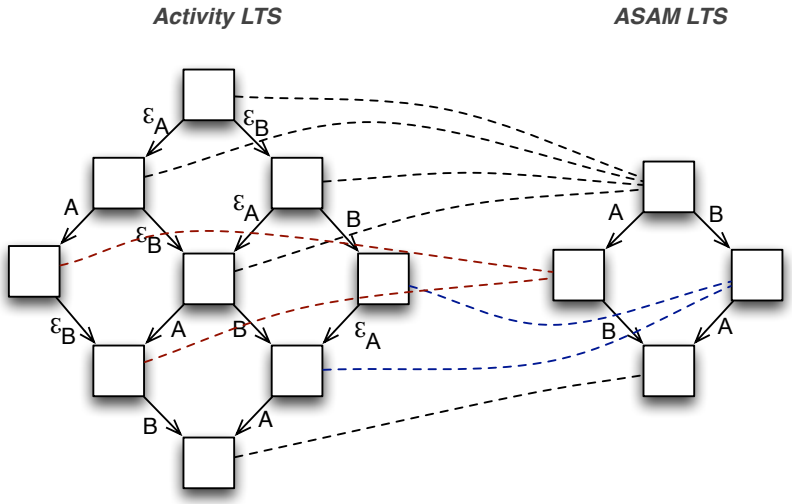


FIGURE 4.28. Remove Activity Intermediate Steps in ASAMs

After removing the intermediate steps, the LTS is transformed into an Nondeterministic Finite Automaton (NFA) data structure that is transformed into a Deterministic Finite Automaton (DFA) using the powerset construction algorithm [RS59]. Next, the DFA is minimized using the Hopcraft algorithm [Hop71], and finally, the DFA is transformed into an ASAM and the original UML actions are linked to the transitions.

In the end, an ACML model has been transformed into an ACIL model that is divided into several dependency groups and uses ASAMs to describe behavior.

In Section 4.3.3, we show the performance benefits of using the ACIL over the ACML.

4.3.2 Multi-Staged Model Checking

The Multi-Staged Model Checking (MSMC) approach further speeds up feedback provisioning for the designer. Besides using the optimized ACIL for model checking, the checking is performed in different stages as illustrated in Figure 4.29.

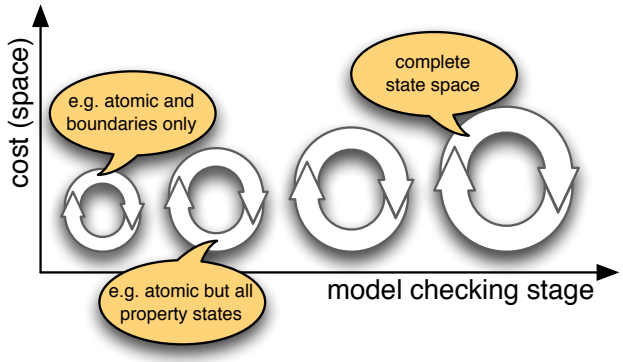


Figure 4.29. Model Checking Runs in MSMC-QUAASY

Earlier stages abstract from the original ACIL model to gain speed and reduce costs in terms of space (i.e., for the state space). Errors or other properties found in early stages are included in the original ACIL model as well. Thus, the model checking may be stopped during an early stage if an error has been found saving the time of constructing the complete state space. As depicted in Figure 4.29, the stages abstract the ACIL along different dimensions, e.g., treating adaptation activities atomic or checking only the boundaries of open properties. The dimensions used may be chosen arbitrarily with the only restriction that the found properties have to be preserved in the non-abstracted setting. We say the dimension is *property preserving*.

EARLIER STAGES RUN FASTER BUT FIND LESS ERRORS

In this thesis, two concrete abstraction dimensions are presented for the use with MSMC-QUAASY, the atomicity of adaptation activities and the boundary states of open properties. They will be described in the following.

Boundary Abstraction of open Properties

If applying boundary abstraction, open properties are only checked for their boundaries. See Figure 4.30 for an example. The sensor "Crisis" has a prop-

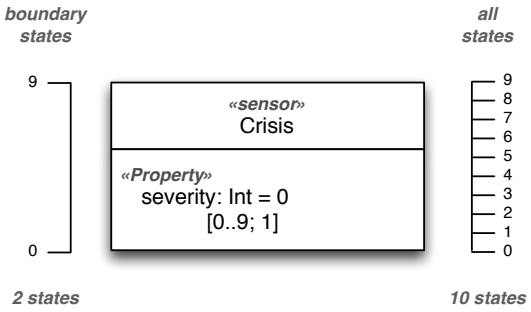


FIGURE 4.30. Boundary Abstraction of open Properties

erty named *severity* which ranges between 0 and 9 with a step size of 1. The default value is 0. In the abstracted setting, this open property would generate two states, i.e., doubling all other existing states in the dependency group (cf. Section 4.3.1). In the non-abstracted setting, the property would generate 10 states, i.e., multiplying the number of existing states with 10. Hence, in the abstracted setting, we reduce the number of states in the LTS by a factor of 5 for only *one* property. An evaluation of the benefits in terms of space is given in Section 4.3.3.

Atomicity Abstraction

The second abstraction dimension treats adaptation activities to be atomic.

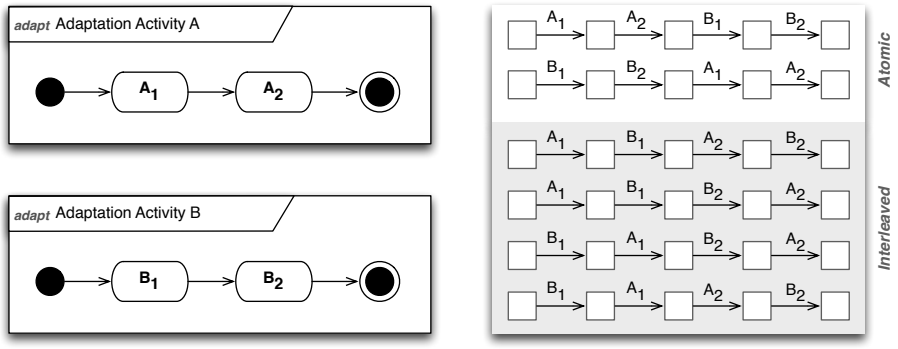


FIGURE 4.31. Atomicity Abstraction of Adaptation Behavior

This dimensions is illustrated in Figure 4.31. On the left side, we have two adaptation activities that belong to two different Adapt Cases. Both adaptation activities have two adaptation actions, A_1, A_2 and B_1, B_2. In the original ACML/ACIL semantics, these adaptation activities can be executed inter-

leaved, leading to 6 traces in the LTS, as shown on the figure's right side. If the atomicity abstraction dimension is applied, the interleaved cases (see bottom four traces in Figure 4.31) are ignored and A_2 always follows directly after A_1, same being the case for B_2 and B_1. Thereby, we decrease the number of traces to two. Since the two traces are also included in the complete LTS, errors found in the abstracted setting will exist in the non-abstracted setting, too. Thus, the dimension is property preserving.

Stage Hierarchies and their Construction

Theoretically, an arbitrary number of abstraction dimensions can be used with the MSMC approach. Some properties may have to be checked on every stage, while for others, it might be sufficient to use an abstracted LTS. Further, an increased number of dimensions leads to an increased number of model checking runs. If the benefit gained from one of the dimensions is not large enough, the average used time of the overall model checking procedure might be larger than the originally used time. For instance, if most of the errors are found in early stages, the used time and space is successfully reduced. If, however, most errors are found late, e.g., because of bad chosen dimensions, there will be a disadvantage in time.

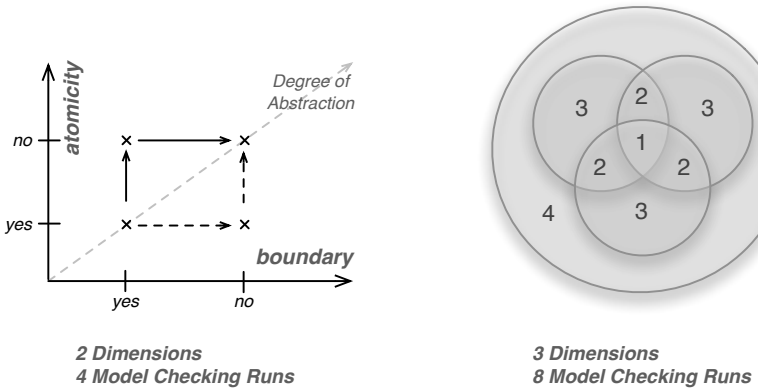


Figure 4.32. Stage Hierarchies based on the Degree of Abstraction

Thus, a careful selection of dimensions and planning of the stages is important for the success of the MSMC approach. Figure 4.32 illustrates the dependency of the number of dimensions on the number of model checking runs. On the left hand side, we used the two dimensions atomicity and boundary stages. In the first stage, both dimensions are applied. In the second stage, each of the two dimensions is applied separately. In the last stage, non of the dimensions is applied. Overall, we have four model checking runs. On the figure's right side, a stage hierarchy with three dimensions is illustrated. First, all three dimensions are applied ①. Next, each two dimensions are applied combined

②, followed by each dimension applied separately ③. Finally, the complete LTS is generated ④. Thus, the LTSs that are generated lose abstraction with each step, such that the first uses less space than the second and so on. The use of three dimensions results in 8 model checking runs. Of course, single model checking runs could be removed if for some reason they do not make sense or do not add value.

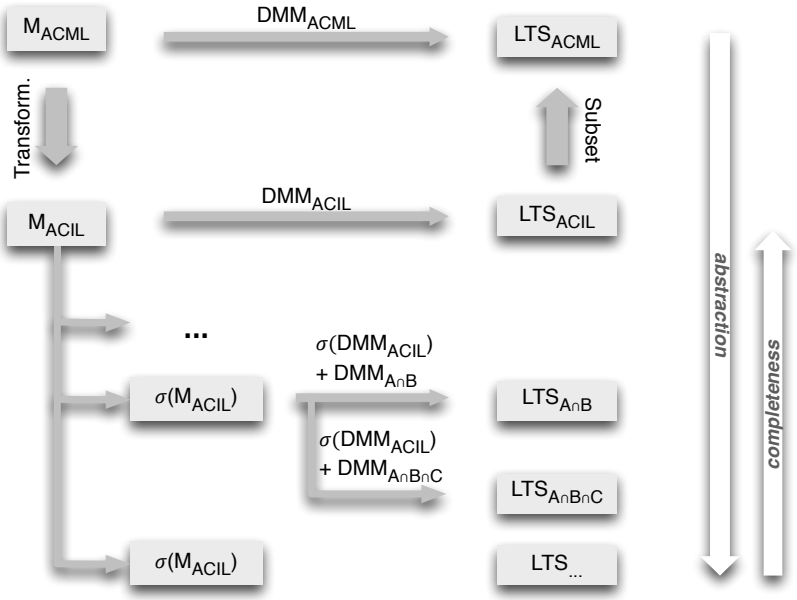


FIGURE 4.33. Stage Hierarchy Construction

Figure 4.33 describes how the stages are constructed. The first and second row correspond to Figure 4.19. An ACML model that would have been translated into an LTS using DMM semantics is now first translated into an ACIL model. The corresponding ACIL semantics produce a smaller LTS.

PARAMETERS FOR STAGE HIERARCHIES

To implement the stage hierarchy, there are different parameters that can be changed. On the one hand, the model can be changed or sliced. Thus, we select only particular information from the model: $\sigma(M_{ACIL})$. An example would be to select only particular dependency groups for a particular stage. On the other hand, the DMM semantics can be changed themselves. First the semantics can be sliced to simulate only a specific part of the model or a specific aspect. For instance, adaptation activities could be deactivated for specific runs. Second, the semantics can be altered or extended to reflect specific constraints. One of those constraints could be an additional rule that assures that only one adaptation activity is activated per time, thus implementing the atomicity dimension. Another constraint could assure that only boundary states are generated, thus implementing the boundary dimension. Combining these two constraints results in the semantics $DMM_{A \cap B}$ that implements Stage 1 of the setting shown in Figure 4.32, left.

The more constraining semantic rules are added to the semantics rule set, the higher the Degree of Abstraction (DoA), and the less complete the resulting LTS.

In the next section we will present some figures gathered during an evaluation of the MSMC-QUAASY approach.

4.3.3 Performance Evaluation

In this section, we present an evaluation of the optimizations for QUAASY. We will define the evaluation in terms of research questions and hypotheses, describe its design in terms of the used scenario, present the execution results, and interpret and discuss theses results. Finally, we will present known threats to validity.

Evaluation Definition

The evaluation shall provide evidence that the construction of the labeled transition systems requires less space. This is the main objective of the optimizations tackling the problem of state space explosion. The research question is as follows.

RQ1 How large are the advantages in terms of space if the optimizations for QUAASY are applied.

We state the hypothesis that the optimizations save more than 50% of space compared to the naïve approach without any optimizations.

Evaluation Design

In the following, we briefly describe the scenario used for the evaluation. It is based on the bCMS scenario given in Section 2.1.

Figure 4.34 shows the Adaptation View Model for the scenario. It describes an environment component named *Crisis* that has one effector and two sensors. The effector allows to assign or retract vehicles from the crisis. The *Severity* sensor reflects the crisis' severity ranging from 0 to 30 with a step size of 10. Finally, the *Balance* sensor is computed from the *Vehicle* effector and the *Severity*

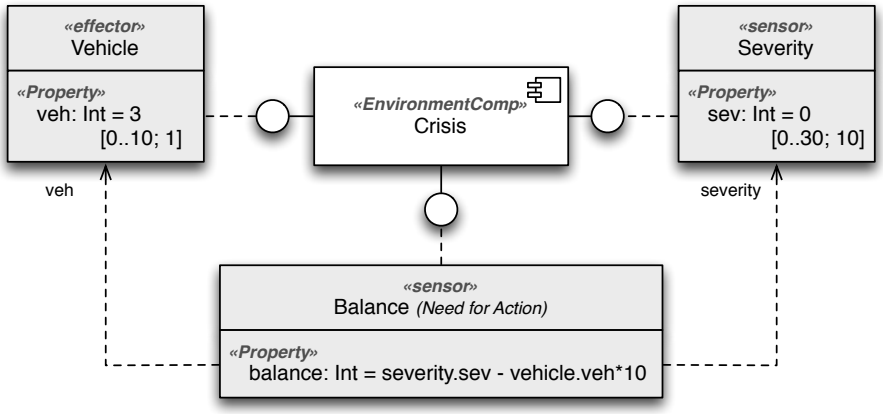


FIGURE 4.34.
Simple Adaptation View Model for bCMS

sensor and judges about the balance between the crisis' severity and the number of vehicles that are assigned. If the balance value is positive, more vehicles should be assigned, if it is negative, less vehicles than assigned are necessary. It is important to notice, that each real crisis instantiates the *Crisis* environment component and has separate values for sensors and effectors.

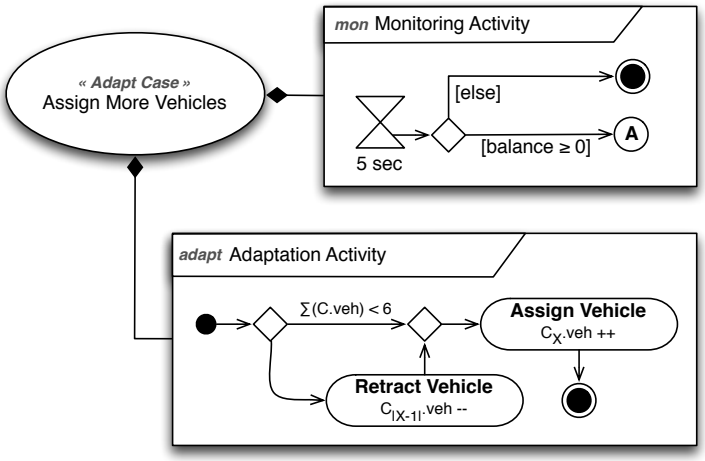


FIGURE 4.35.
Simple Adapt Case for Vehicle Assignment in bCMS

Figure 4.35 shows a simple Adapt Case that assigns more vehicles if the balance is positive. The Adapt Case's monitor is started for each crisis. The crisis observed is denoted with C_X with X being the crisis' number. For simplicity, we start the monitor after each change in the system or environment. Further, the Adapt Case assumes that no more than 6 vehicles are available. If less than 6 vehicles have been assigned, a new vehicle is assigned to the corresponding crisis in the adaptation activity. If all 6 vehicles are already assigned, a vehicle is retracted from another crisis before.

For the evaluation, we instantiate the system with two crises. With the described Adapt Case, this small scenario leads into an unstable system state if

both crises want to assign a new vehicle. This will be demonstrated in the next section.

Evaluation Execution

Before describing the concrete scenario, let us look at Figure 4.36 that shows the state space (LTS) for the small example with the original ACML semantics.

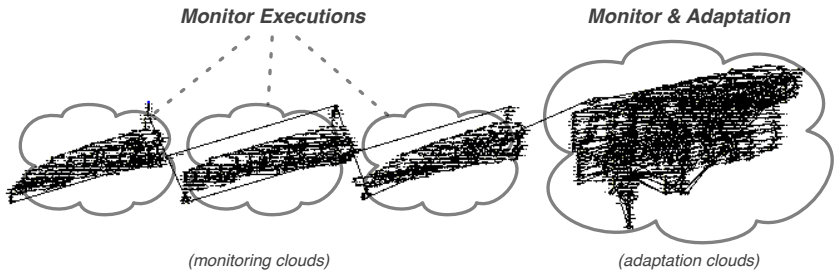


Figure 4.36. Original Transition System - 1270 states, 2188 transitions

We observe 4 almost independent state clouds, the first three of which describe single monitor executions and the last of which describes monitor and adaptation executions. The state space consists of 1270 states and 2188 transition which is remarkable much for this rather small scenario.

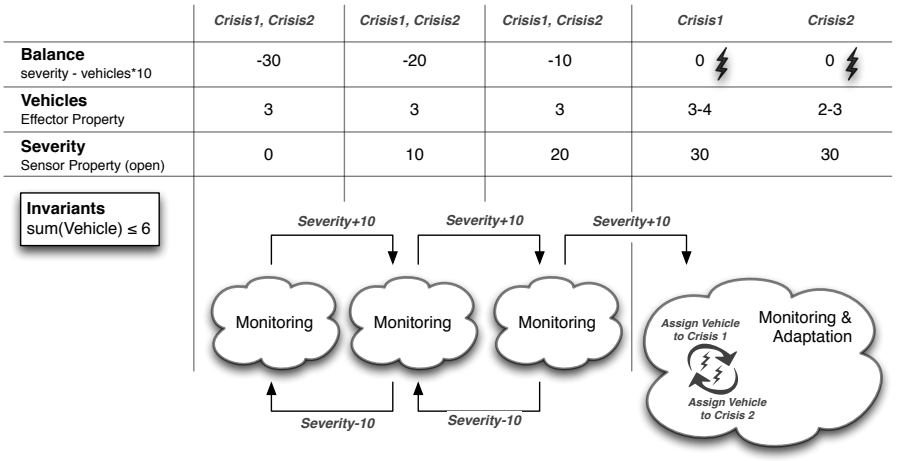


Figure 4.37. Sketch of the Transition System

See Figure 4.37 for a schema that describes how the LTS is constructed. We start with an initial system configuration of two crises that both have 3 vehicles assigned and a severity of 0. Thus, the balance is −30. In the scenario setting, we assumed that the severity of both crises increases and decreases simultaneously. Further, there is an invariant that in total at most 6 vehicles are assigned. As we have seen above, the Adapt Case takes care of this invariant.

If in a first step, the severity of the crises increases by 10 (step size), the balance decreases to −20. After a change in the environment, we again start the monitors resulting into the second small cloud. Same applies to the third cloud. Next, if the severity increases to 30, the balance reaches 0 and the monitors trigger the adaptation activities. Since all 6 vehicles are assigned, the Adapt Cases retract a vehicle from the other crisis which, in turn, leads to retraction of vehicles from the original crisis, and so on. Thus, the system reached an unstable system state that cannot be left since adaptation has priority over system and environment progress.

In the following, we will show the results from the applied MSMC-QUAASY. We use three stages. In the first stage, we use the atomicity and boundary abstractions. In stage two, we construct two transition systems, one for the atomicity abstraction and the other for the boundary abstraction. Finally, in the third stage, we construct the complete state space. For all constructions, we use the ACIL over the ACML. The stability issue can be found in all stages.

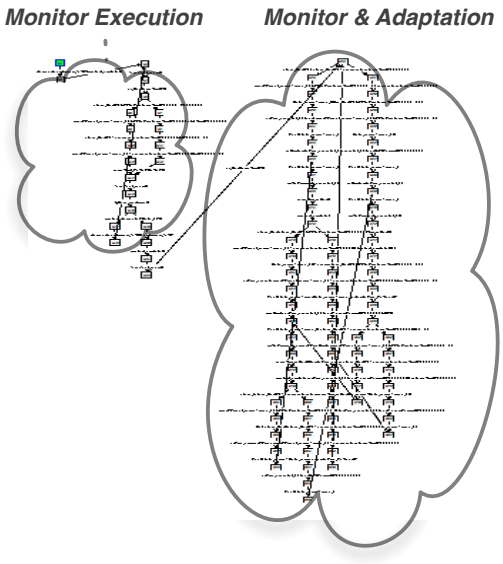


FIGURE 4.38.
Stage 1 LTS (Atomicity and Boundary Abstraction) - 98 states, 104 transitions

Stage 1 Transition Systems Figure 4.38 shows the state space with both abstractions applied. As obvious, the state space is far smaller than the original one that has been created using the ACML. In detail, the two remaining clouds contain less states since the Adapt Cases are executed atomically. Additionally, the ACIL transformation leads to smaller state spaces since UML offers are neglected and dependency groups are checked separately. In addition to the size of clouds that has decreased, the number of clouds decreased as well. This is because of the boundary abstraction. The state space consists of 98 states and

104 transitions.

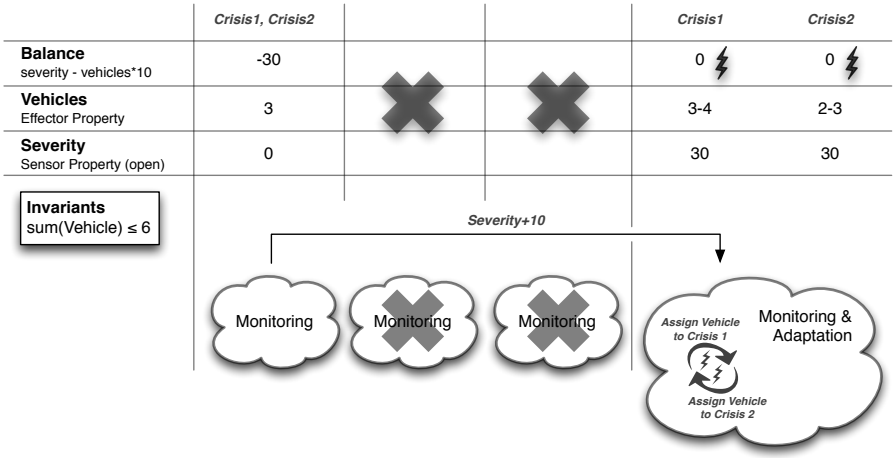


Figure 4.39. The Effect of Boundary Abstractions

Figure 4.39 shows the LTS schema for the boundary abstraction case. Since the severity ranges between 0 and 30 we only consider the respective cases and neglect the intermediate steps 10 and 20. Of course, if the sensor would be more fine-grained, e.g. ranging from 0 to 50 with a step size of 1, the benefit of boundary abstraction would be even larger.

Stage 2 Transition Systems On Stage 2 we first apply the atomicity abstraction followed by the boundary abstraction.

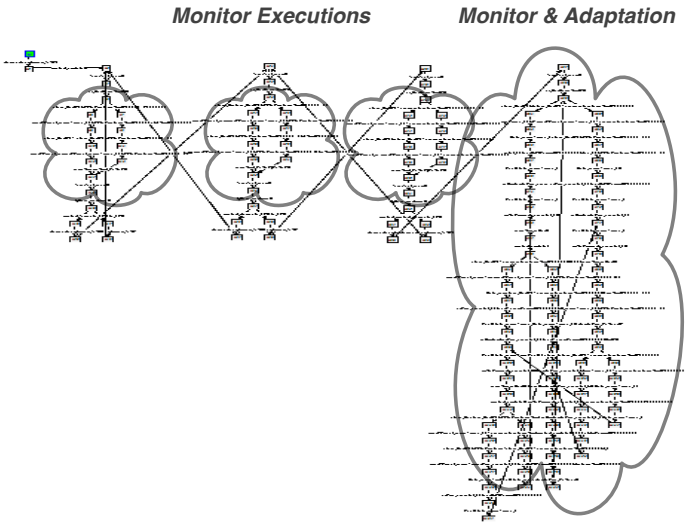


Figure 4.40. Stage 2 LTS (ACIL, Atomicity Abstraction) - 134 states, 144 transitions

Figure 4.40 shows the state space for the atomicity abstraction only. Again, the clouds are rather small but all 3 small monitor clouds exist instead of only the

first like on Stage 1. The state space consists of 134 states and 144 transitions which still is very small compared to the original one.

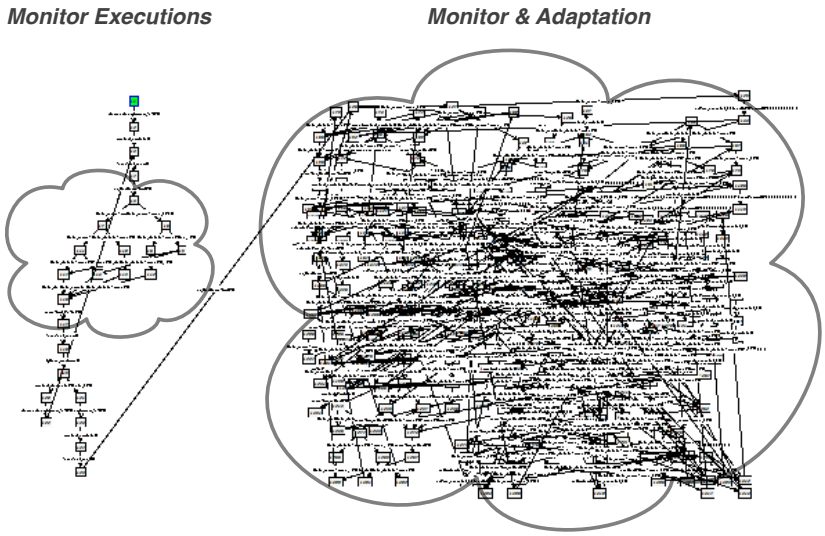


FIGURE 4.41.
Stage 2 LTS (ACIL, Boundary Abstraction) - 393 states, 683 transitions

Figure 4.41 shows the state space with the boundary abstraction applied only. While the two intermediate monitor clouds are neglected again, the clouds themselves are larger since monitoring and adaptation activities are executed interleaved. This results in a small explosion of states. The resulting state space consists of 393 states and 683 transition.

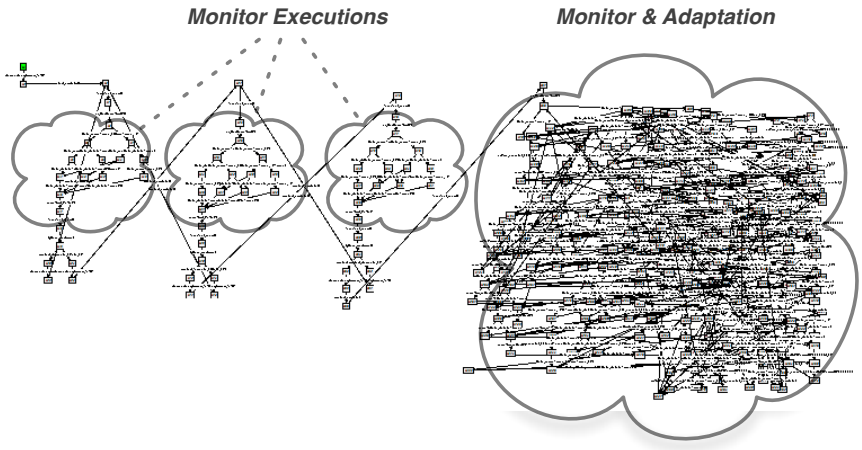


FIGURE 4.42.
Stage 3 LTS (ACIL Complete) - 435 states, 737 transitions

Stage 3 Transition Systems Finally, on Stage 3, the complete state space is constructed without any abstractions. Thus all 4 clouds exist and the clouds are rather large in their number of states. However with 435 states and 737

transitions it is still small compared to the original ACML state space which is because of the use of ACIL instead of ACML.

Evaluation Results & Discussion

space reduction of more the 50%

The results of the evaluation are quite impressive. All state spaces decreased in size more than 50%. The concrete figures are listed in Table 4.2. All model checking runs are given with the applied abstraction dimensions, the number of resulted states and transitions, as well as the sum of states and transitions that is used as an indicator for complexity. The last row shows the benefit (percentage) compared to the original model checking approach that is based on the ACML. The original approach's figures are given in the first column. With no abstraction dimensions applied, the resulting state space consists of 1270 states and 2118 transitions. The next four columns describe the three stages' resulting figures. As expected, the greatest benefit can be obtained by applying both abstraction dimensions. This has been done in Stage 1 achieving a benefit of 94%. Stages 2 and 3 gained a benefit of 90% and 68% respectively. Finally, using plain ACIL without any abstraction led to a benefit of 65%. All in all, with a minimum benefit of 65%, the results look really promising. It appears that the atomicity abstraction results in much larger benefits than the boundary abstraction. This is due to the configuration of the scenario. If the step size of the open sensor would have been smaller or more sensors would have been taken into account, the benefit would have been larger correspondingly. Thus, the benefits of particular abstraction dimensions depends on the concrete models at hand. Of course, the MSMC approach can be equipped with additional abstraction dimensions to form even larger stage hierarchies. Often, the advantage of an abstraction dimension depends on the quality properties that shall be proven.

	Original (ACML)	Stage 1 (ACIL)	Stage 2 (ACIL)	Stage 2 (ACIL)	Stage 3 (ACIL)
Abstraction Dimension	none	Atomic & Boundary	Atomic	Boundary	none
States	1270	98	162	393	435
Transitions	2118	104	180	683	737
Total	3388	202	342	1076	1172
cp. ACML	0	94	90	68	65

Table 4.2. Comparison of the different Abstraction Dimensions

Threats to Validity

The evaluation, designed and executed as described above, gives rise to a few threats to validity. First, the scenario might be picked with the abstraction dimensions in mind. This threat was approached by defining the scenario upfront, i.e., before the MSMC approach has been developed. Also, the scenario not only shows benefits but also disadvantages, e.g., the fact that boundary abstraction does not give much advantage in complexity in this scenario. Another threat to validity might be the scenario's size which is rather small. However, we tried to use a realistic example that already would provide valuable results in productive use. That is, the shown scenario reveals the problem of instability although the model size is kept within bounds.

4.3.4 Discussion

CONCERN-SPECIFIC OPTIMIZATIONS

The presented techniques to tackle the problem of state space explosion are concern- and language-specific. That is, we used knowledge about the semantics of the language to construct smaller state spaces. Of course there are concern and language-independent techniques to alleviate the problem of state space explosion. Especially on the last abstraction level (Stage 3 in our scenario), we use further techniques given by DMM and Groove themselves. First, DMM State Graphs only contain model elements that are influenced by some application of DMM rules. All other elements are ignored. Therefore, DMM state graphs are usually rather small. Second, the Groove researchers provide various techniques to reduce the size of the state space, e.g., by using abstract shape graphs [Ren04] that are automatically obtained from arbitrary state spaces, or (similarly) graph abstractions that "mitigate the combinatorial explosion inherent to model checking" [ZR11]. By the use of these techniques, we can keep the runtime of our model checking approach sufficiently small.

ADDITIONAL OPTIMIZATION TECHNIQUES NECESSARY

Nonetheless, models tend to get very large. The more powerful the language, the more sophisticated the created models. Thus, in future it will be inevitable to come up with additional means to tackle the state space explosion problem. One solution can be the definition of additional abstraction dimension which in our experience is not a trivial task since they should be designed to be property preserving, i.e., errors found in early stages must be contained in the complete state space. In the end, tackling the state space explosion problem stays a research field on its own.

summary & discussion 4.4

In this chapter, we presented a quality assurance approach for self-adaptive software systems that uses model checking based on the ACML. Further, we presented how the approach can be optimized to tackle the state space explosion problem. In Section 4.1 we identified several requirements for the quality assurance approach. In the following, we will summarize how the requirements are met.

QR01: Check Adaptation Rules QUAASY provides formalized semantics of explicit adaptation rules. Thus, adaptation rules can be checked for certain properties on different abstraction levels. The semantics allow to analyze every possible interaction between adaptation rules and find unintended interrelations.

QR02: Check Progress Rules QUAASY defines the semantics for the system as well. This also allows to simulate the system and the effect of adaptation, and thus, the interrelations between application and adaptation logic. Further, QUAASY defines several validation routines that check the adapted system for several properties such as well-formedness, etc.

QR03: Check Functional Quality QUAASY translates the created models into a model checking problem. The used model checker Groove allows the use of LTL and CTL for checking functional correctness of the system and thus indicate functional quality. QUAASY defines several generic functional quality properties such as stability and deadlock freedom. Application-specific functional quality properties can be defined using `ACMLConstraints`.

QR04: Direct Modeler Feedback QUAASY uses techniques to offline create parts of the state space. Further, several optimizations have been defined to allow a very fast model checking on high abstraction levels which may indicate first modeling flaws. Combining these techniques, QUAASY is possible to provide the modeler with almost immediate quality feedback after he is saving a consistent state of his models.

QR05: No Formal Knowledge Needed The use of the model checker Groove and the DMM techniques is hidden from the modeler (transparent or tool-encapsulated model checking). That is, the modeler only uses the ACML to model the system and an OCL-like language to define application-specific quality properties. The model checking is triggered

transparently and the results are presented in terms of the ACML again. Thus, no knowledge of formal techniques is required.

QR06: Support Generic and Application-Specific Constraints QUAASY comes equipped with a range of predefined generic quality properties such as stability. Further, QUAASY allows the modeler to define application-specific properties using an OCL-like language within the ACML models.

All in all, the quality assurance approach meets all requirements and thus greatly supports the designer in modeling a self-adaptive software system. By now, the approach only supports functional properties. Non-functional properties are not support by QUAASY. However, ongoing work investigates the combination of ACML, QUAASY, and SimuLizar, a performance analysis approach for self-adaptive software system [BLB13].

5

Engineering Self-Adaptive Systems

"Verbal and nonverbal activity is a unified whole, and theory and methodology should be organized or created to treat it as such."

– Kenneth L. Pike

5

A thoughtful development process is the basis for good software engineering. Thus, in this chapter, we will give a brief overview of how the Adapt Case Modeling Language can be integrated into standard development processes. Further, using the SPEM Profile (Software Process Engineering Metamodel), we will sketch a development processes that uses goal-oriented requirements engineering and is based on the UML for system design.

Figure 5.1 sketches the development process that will be described in the following. We assume that the process uses a V-Model like structure, that is a waterfall phase for the design and implementation of the system and a matching testing branch in the second half of the project. A project's requirements are specified using a goal-oriented requirements engineering approach (GORE, here KAOS [vL09]). Further requirements are given using controlled natural language (CNL). An extension to standard CNL schemes is RELAX [WSB+09] that allows relaxing requirements and thus allowing the system to adapt within a predefined corridor. Besides KAOS, CNL, and RELAX, the UML is used for requirements and system design.

Our particular focus is on the late requirements phase and the system design phase which consists of two sub phases, the logical and the technical design phase. The logical design phase is driven towards a platform-independent design while the technical design phase is targeted at a platform-specific designs. Of course the two sub phases gradually fade into each other. Depending on the concrete development process paradigm, these phases are repeated iteratively, incrementally, or even in a waterfall kind of manner. No matter of the chosen paradigm, the main artifacts of the two phases remain the same.

REQUIREMENTS

During requirements engineering, a single overall System Goal is defined in the first step. In a second step, this goal is refined to Requirements, e.g. using a goal-oriented approach such as presented in [CSBW09]. In goal-

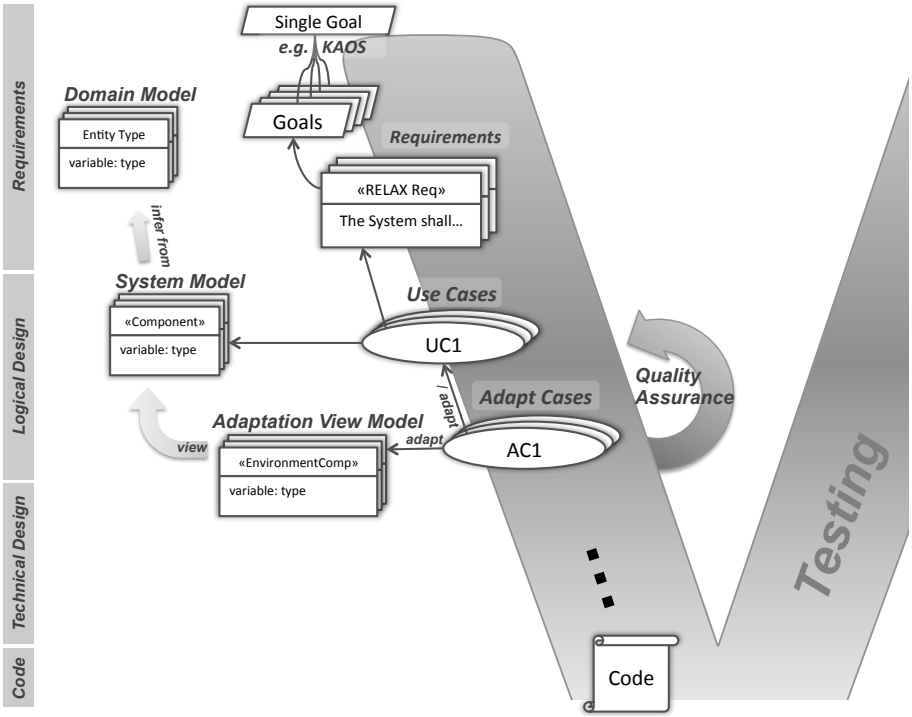


FIGURE 5.1.
The ACML used within the V-Model

oriented approaches, a goal-lattice is created that reflects the system's purpose. Usually, leaves in this lattice (i.e., goal tree) are the concrete system requirements. According to [CSBW09], these requirements may be relaxed using RELAX [WSB+09]. Although requirements describe the WHAT of a system rather than the HOW, in late requirements engineering, the requirements may include explicit adaptation requirements. This is commonly seen in practice but still not recommended since adaptation requirements usually describe a concrete solution for a specific situation the system may be in. The correct root phase for self-adaptivity is the design, where requirements are operationalized. Finally, in the requirements phase a very high-level domain model is created that describes the main entities for the target domain.

LOGICAL DESIGN

The next process activity (logical system design) contains two tasks, *a*) the definition of the core business logic and *b*) the definition of the adaptation logic. The core business logic is described using use cases and UML component and class diagrams. Use Cases are a first operational description of how requirements can be realized. Many use cases may realize many requirements (n:m). A class or component diagram defines a first logical structural view onto the system (i.e., system model). Often, the class or component diagram is inferred from the domain model. Thus, the input for the first task are (relaxed) requirements and the domain model. If adaptation requirements exist, they may be neglected here and specified in the adaptation logic. Also, specific alternatives

BUSINESS LOGIC

or exceptions may be left for definition in the adaptation logic. Thus, the created use cases and the system model alone might not fulfill *all* requirements.

ADAPTATION LOGIC

The second design task is the definition of adaptation logic. Adaptation logic usually refers to business logic. Thus, *first* the business logic has to be defined which *then* may be adapted by adaptation logic. However, business logic and adaptation is usually defined in parallel since the designer might have at least basic adaptation rules in mind while creating the business logic.

For the definition of adaptation logic, Adapt Cases and the Adaptation View Model are used. Adapt Cases externalize specifications of adaptation on use case level. Indicators for adaptivity are alternative and exceptional flows in use cases, relaxed requirements, and of course, adaptation requirements. However, as discussed in Section 2.2.2 the distinction of adaptation and application logic is up to the designer.

The Adaptation View Model introduces several new aspects that allow to clearly define adaptivity specific elements. Examples include EnvironmentComponents, SystemComponents, Sensors, Effectors, etc. The Adaptation View Model is described in detail in Chapter 3. Usually, the Adaptation View Model is an extended view onto the system model. That is, relevant entities of the system model are visualized within the AVM and decorated with additional information, e.g. adaptation interface properties.

TECHNICAL DESIGN

During design phase, use cases and system model are refined towards a more detailed and technical description. In the same manner, Adapt Cases and Adaptation View Model are refined. Finally, Use Cases, Adapt Cases, System Model, and Adaptation View Model are handed over to implementation. In rare cases, these models are interpreted directly. A concrete example is the use of process models and related Adapt Cases in process execution engines [Res12].

The design using Adapt Cases enables early quality assurance on a high-level of abstraction. Generic quality constraints ensure the non-application-specific correctness of the adaptation specification. Examples include stability, deadlock-freedom, etc. Application-specific quality constraints ensure the correct realization of requirements. Application-specific quality constraints are inferred from/take into account the requirements.

The process description given above is rather an overview of where the ACML integrates into arbitrary UML-based processes. To give a more detailed view of using the ACML, in the next section, we will define an engineering process for self-adaptive systems using the *Software Process Engineering Metamodel*

(SPEM) [Obj08]. The defined SPEM artifacts can be used with others that describe another process such as RUP or SCRUM and thus help to obtain a complete formal method description for engineering self-adaptive software systems.

A SPEM ENGINEERING PROCESS DEFINITION 5.1

SPEM = SOFTWARE PROCESS ENGINEERING METAMODEL

In this section, we describe a basic engineering process for self-adaptive software systems using the Software Process Engineering Metamodel (SPEM). The SPEM was developed to enable the modeling and exchange of software engineering processes. The main artifacts of SPEM are roles, tasks, and work products as show in Figure 5.2. Roles perform special tasks that take as input several work products and produce several work products as outputs. This triple defines reusable tasks that can be orchestrated in arbitrary engineering processes. Optionally, tasks may be further refined by steps. For the description of the steps' ordering, activity diagrams can be used since steps inherit from UML actions. An engineering process is described using SPEM activities (upper left in Figure 5.2) that inherit from UML activities. Thereby, tasks are orchestrated by the use of UML activity notation. The orchestrated tasks are given by *Task Use*s, the instantiation of task definitions.

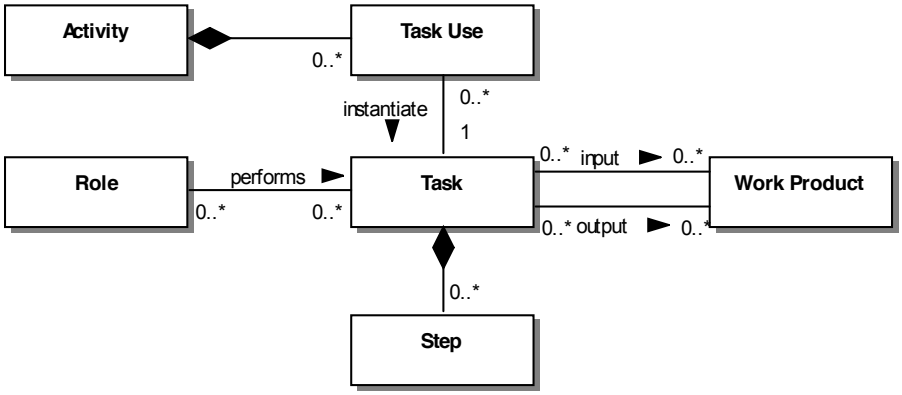


FIGURE 5.2.
SPEM Metamodel

For our engineering process for self-adaptive software systems, we will first describe the definition of roles and tasks which are then orchestrated in an engineering process. The process is based on standard engineering processes. These standard engineering processes are reflected in the use of standard roles such as *Analyst*, *Architect*, *Designer*, *Developer*, and *Tester*. To take the specific requirements of engineering self-adaptive systems into account, an additional

role, the *Adaptation Engineer* is introduced, that is divided into further sub roles, the *Adaptation Analyst*, the *Adaptation Architect*, and the *Adaptation Designer* (cf. Figure 5.3). As known from standard engineering processes such as the Rational Unified Process [Kru03], the analyst is responsible for requirements engineering, the architect creates an overall system architecture, and the designer details the architecture, e.g., with platform-specific details, for implementation.

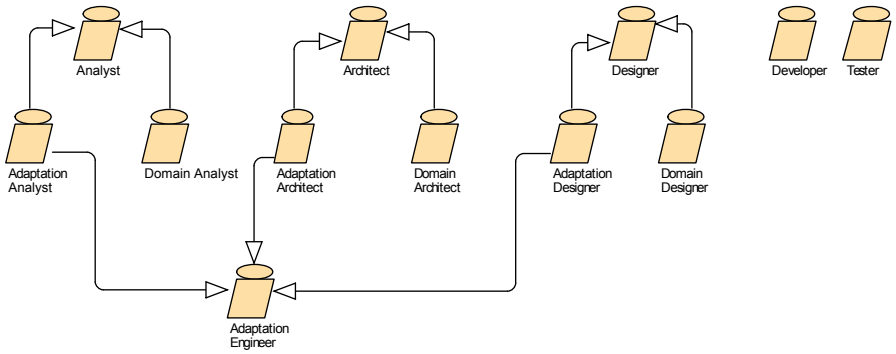


FIGURE 5.3. Extended Role Model for Engineering Self-Adaptive Software Systems

In the following figures, we describe the tasks the different roles are responsible for. Thereby, we focus on the tasks that were added or are vitally important for the purpose of engineering *self-adaptive* software systems. The remaining tasks are defined in standard development processes and can be obtained, e.g., from the Internet at `http://epf.eclipse.org/wikis/openup`.

Figure 5.4 describes the tasks, the domain analyst and the adaptation analyst are responsible for. The first task is performed by the domain analyst. He elicits the overall project or system goals and describes the context the system acts in. Goals and context are input for the following two tasks. The requirements analysis performed by the domain analyst results in a set of use cases and a set of overall system requirements, i.e., the requirements that cannot be represented by use cases, e.g., since they do not describe external system behavior. The adaptation analyst performs a requirements analysis dedicated to the concern of self-adaptation resulting in a set of Adapt Cases (i.e., high-level adaptation rules) and uncertainties. These uncertainties describe *open variables* in the context, i.e., context variables that cannot be specified precisely, e.g., since they are outside the system scope. Of course, it is recommended that both the domain analyst and the adaptation analyst cooperate on their models.

Figure 5.5 details the task *Adaptation Requirements Analysis* with the notion of steps. As shown in the figure, the steps are suggested to be performed in sequence. After having identified the uncertain context variables (i.e., open variables) they are detailed by defining a range (if known), a frequency of change

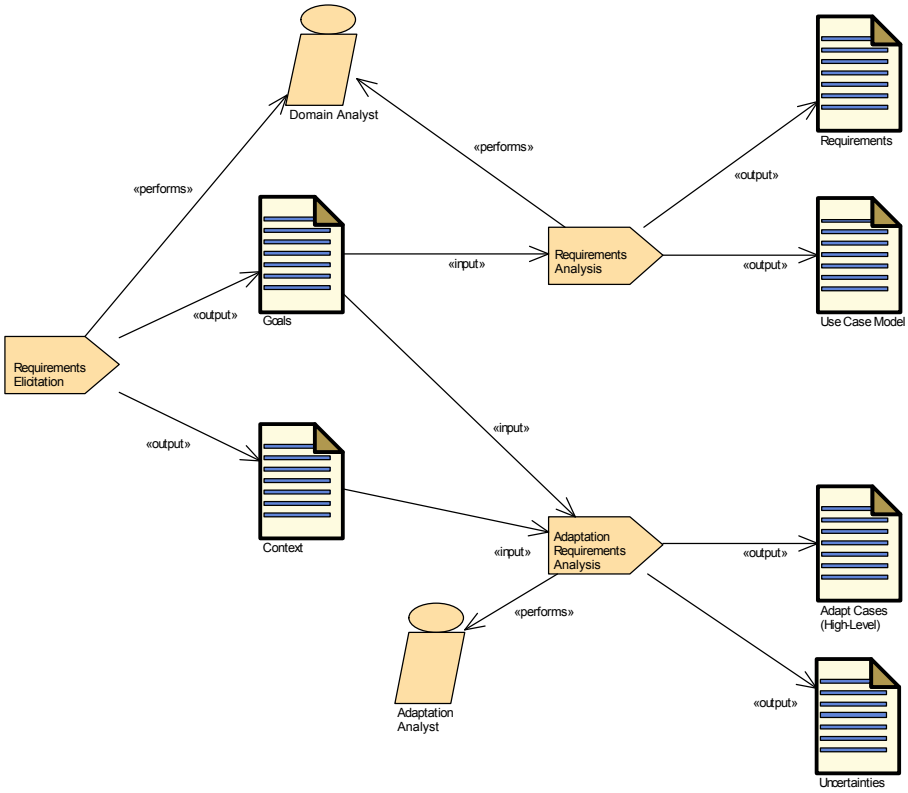


FIGURE 5.4.
Tasks of Domain and Adaptation Analyst

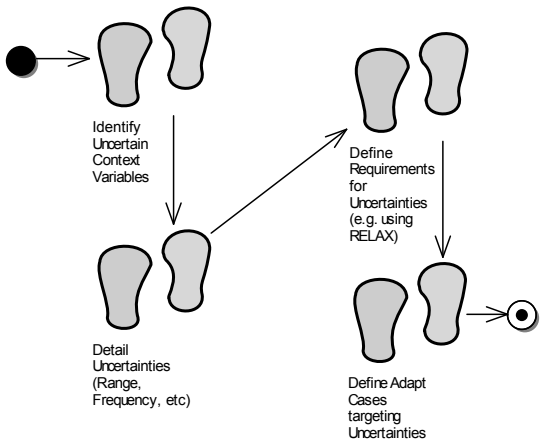


FIGURE 5.5.
Task: Adaptation Requirements Analysis

(if known), and a step size. The latter is used for simulating the open context variable during quality assurance. The uncertainties may require certain system functionality to be present, e.g., handling an uncertainty or constraining the effects that might occur due to the uncertainty. This functionality must be described using additional requirements or adjusting existing requirements. For considering the uncertain context, the controlled natural requirements language RELAX [WSB+09] can be used. Finally, in the last step, Adapt Cases can be defined that describe how the uncertainties are handled on a high-level of abstraction.

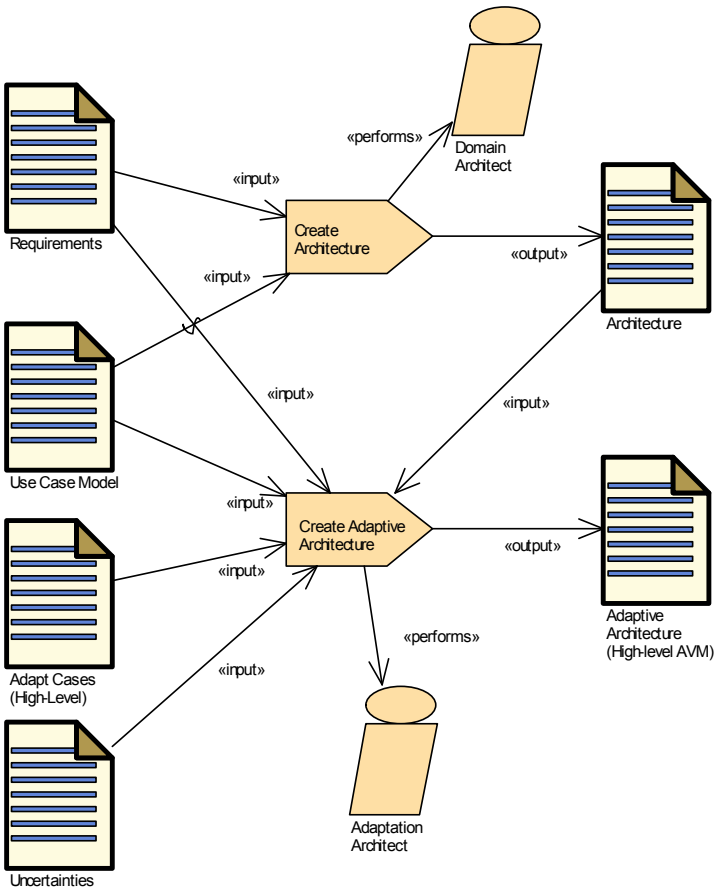


FIGURE 5.6.
Tasks of Domain and Adaptation Architects

The next tasks, the domain and adaptation architects are responsible for, are shown in Figure 5.7. The resulting artifact of the task *Create Architecture* performed by the domain architect is a system architecture which does not yet consider any adaptation-specifics. The adaptation architect creates an adaptation architecture using the Adaptation View Model (AVM). Therefore, he considers the created architecture that must be reflected by the AVM. The AVM contains first high-level definitions of adaptation interfaces (sensors and effectors) that are used by the high-level Adapt Cases. Of course, this might include a feed-

back loop with the adaptation analyst to adjust the high-level Adapt Cases.

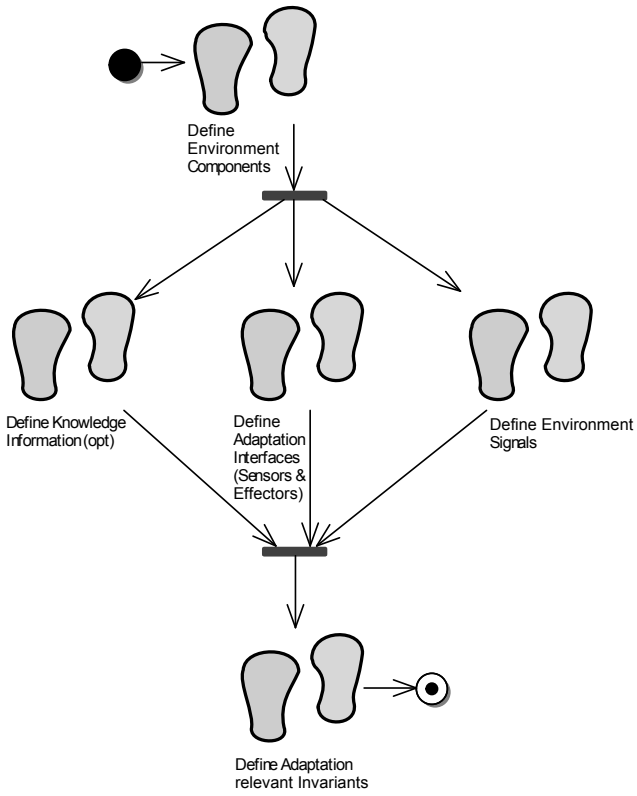


Figure 5.7.
Task: Create Adaptive Architecture

As shown in Figure 5.7, creating an Adaptation View Model includes *a*) the definition of environment components which are inferred from the context description, *b*) the definition of sensors and effectors as well as additional knowledge such as computations, *c*) the definition of signals that may originate from the environment, and finally, *d*) the definition of invariants that become relevant when considering self-adaptation.

If an architecture has been created, it can already by analyzed using appropriate tools. As shown in Figure 5.8, the analysis may be performed by the adaptation architect who does not need to be a QA specialist due to the tool-encapsulated transparency of the quality assurance approach (cf. Chapter 4). The analysis is performed using the QUAASY approach that needs as input *a*) the Adaptation View Model and *b*) the Adapt Case Model and produce an *Adaptive Architecture Analysis Report* that can be used to revise the adaptive architecture. Of course, other tools can be used here, e.g. the SimuLizar tool that allows for performance analysis of self-adaptive systems [BLB13].

Note at this place that not all tasks have been described in detail above since in general they are very similar to the shown definitions. Especially, the de-

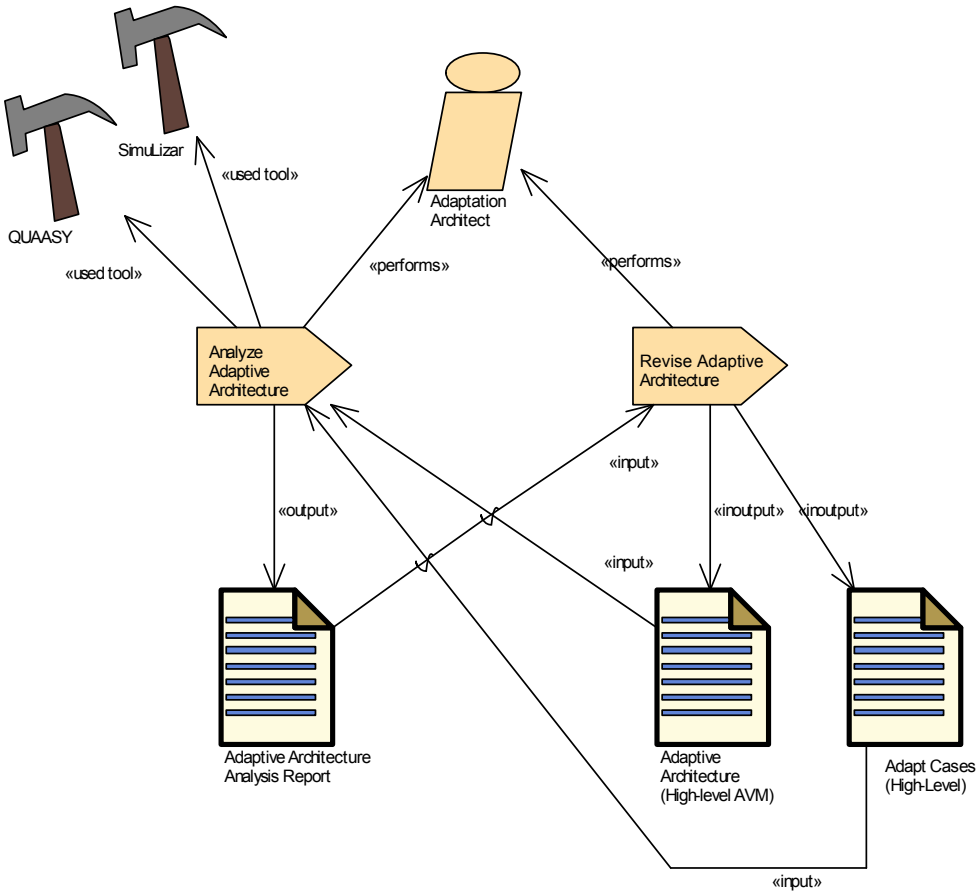


FIGURE 5.8.
Tasks for Adaptive Architecture Analysis

sign tasks the domain and adaptation designers are responsible for are very similar to the architecture tasks despite the fact that the models are detailed for instance by using platform-specific information. If, however, all definitions are in place, the process for self-adaptive software engineering can be defined as shown in Figure 5.9.

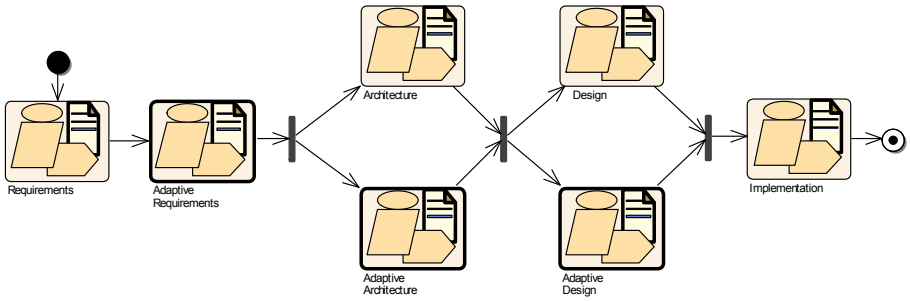


Figure 5.9. Self-Adaptive Software Engineering Process

Figure 5.9 shows several SPEM activities that define an explicit order of the tasks that have been defined above. While the requirements analysis of adaptivity is performed after standard requirements analysis, the adaptation specific activities for architecturing and designing are performed in parallel to their corresponding domain activities. The reason is that requirements that concern the self-adaptivity of the system already described the "How" of the system since adaptivity is a solution for demands of (non-functional) requirements. Thus, requirements analysis concerning the system's adaptivity is part of very late requirements engineering, and hence, is defined to be an activity subsequent to the standard requirements analysis. Obviously, feedback loops to prior activities always exist.

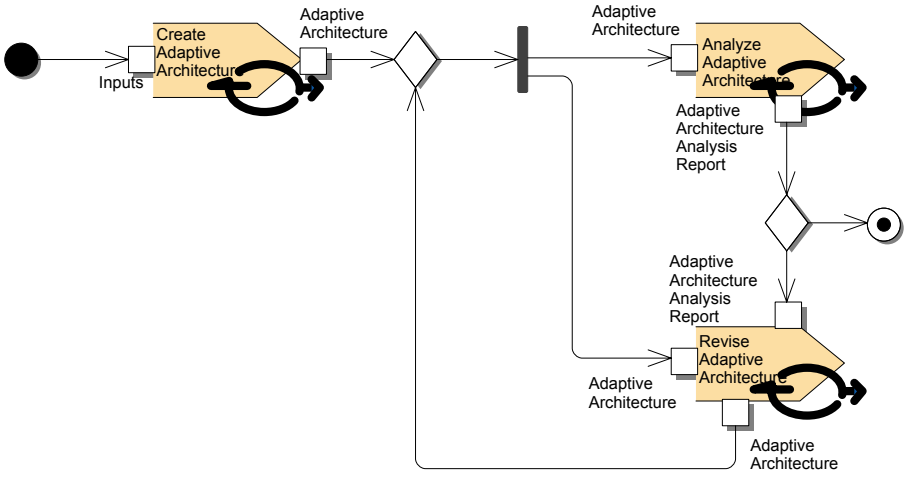


Figure 5.10. Process Activity Definition: Adaptive Architecture

Finally, the detailed definition of the activities shown in Figure 5.9 are given using activity diagrams that orchestrate the tasks defined above. An example activity definition is given in Figure 5.10 that shows the definition of the

process activity *Adaptive Architecture*. According to this definition, first the architecture is created initially, followed by its analysis and in case of recognized flaws it is revised and analyzed again. If no more flaws are identified or it is deliberately decided to not revise the model any more, the activity is terminated and the next process activities (i.e., *Design* and *Adaptive Design*) are executed. Likewise, the other process activities are defined in detail.

5.2 RELATED WORK

There is few research about engineering of self-adaptive software systems. However, the closely related field of self-organizing multi-agent systems (MAS) is more mature in providing (SPEM-based) engineering process descriptions [GCGC08, DW07, DMSFR10]. Since from a process perspective self-organizing MAS and self-adaptive software systems are very similar, in the following we will described some of the works provided for *both* paradigms.

The authors of [GCGC08] propose the ADELFE process for agent-based systems. They distinguish four main work definitions prior to implementation and test being *Preliminary Requirements*, *Final Requirements*, *Analysis*, and *Design*. Similar to our approach, they first investigate requirements for context uncertainties during *Final Requirements* which corresponds to our late requirements phase. In this phase, they suggest to characterize the environment by qualifying determinism, dynamicity, unexpected situations and harmfulness of these situations. During *Analysis* the ADELFE process suggests to identify agents and their relationships. Since agents are the self-adaptive entities the system consists of, this process task is similar to our task of defining Adapt Cases and the Adaptation View Model with sensors, effectors, etc. Studying the relationships between agents corresponds to our quality assurance tasks that can be first applied in this development phase. Finally, in ADELFE's *Design* phase, the agents are designed in detail and first fast prototypes are created. This phase corresponds to our design phase where detailed (platform-specific) models are created to described the system's self-adaptivity. All in all, the ADELFE process that is targeted at agent-based engineering of emergent systems is very similar to our process for self-adaptive software systems, the only big difference being the paradigm chosen (agent-based systems vs. rule-based systems).

Another approach, named MetaSelf [DMSFR10], considers a self-organizing

system as a set of loosely coupled components that act autonomously. The MetaSelf development process focuses on self-* requirements during requirements and analysis and on self-organization mechanisms that are based on agents and policy models during design. The most important tasks within MetaSelf considering self-adaptation are *a*) the definition of adaptation and coordination mechanisms out of self-* requirements, and based thereon, *b*) the definition of policy models and agent models during design. Both tasks reflect the creation of the Adaptation View Model and the Adapt Case Model in our presented process.

Seebach et al. [SNSR10] propose a software engineering guideline using SPEM that helps engineering Organic Computing (OC) systems of self-x systems. Especially, their approach is directed towards resource-flow systems, i.e., several stations within a production street which have several capabilities and may change their state / mode based on current needs, e.g., reacting to a failure of a particular station. The proposed process intends the modeler to first model a non-OC-system which later on is extended to have OC capabilities. This is different from our approach that allows both the business logic and the adaptation logic to be designed in parallel. Of course, this greater flexibility come with the cost of higher efforts in synchronizing modeling activities.

Andersson et al. [ABB+13] describe more generally how traditional software engineering processes have to be reconceptualized to distinguish between development-time and run-time adaptation activities. They extend the SPEM meta model to properly express the new concepts of *online* and *offline* activities within an engineering process for self-adaptive systems. The approach allows to describe self-adaptation activities as engineering activities at run-time using the SPEM meta model. In this chapter, we only describe offline activities whereas online activities are described separately using the concern-specific modeling language ACML.+

Summary & Discussion 5.3

In this chapter we presented a basic software engineering process described with SPEM based on the unified process. Tasks are defined as reusable units that can be orchestrated in arbitrary engineering processes (e.g., waterfall, iterative, agile, etc.). The presented process is particularly interesting in the course of this thesis since it clarifies the use of the presented language (Chapter 3) and

techniques (Chapter 4) within standard software engineering processes. The presented process introduces additional concern-specific roles (e.g., adaptation architect), work products (e.g., adaptive architecture), and tasks (e.g., create adaptive architecture) which are described in detail with the use of SPEM steps.

The definition of this process inherits several benefits. First, the process extension is not invasive at all. That is, it is easy to integrate in different kinds of software engineering processes, e.g., agile processes like SCRUM. Second, by the used concept of roles, the presented process does not require more people to be employed but rather systematically structures the work for existing employees. Of course, additional models (ACM, AVM) require additional efforts, however, the benefit of early quality assurance appears, helping taming the software's inherent complexity. All in all, when defining a software engineering method, the presented extension can be included optionally with low efforts but high benefits when designing self-adaptive software systems.

6

Evaluation

"True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information."

– Winston Churchill

6

THE last three chapters described a modeling language for self-adaptive software systems, a corresponding quality assurance approach, and an engineering process fragment that uses both the language and the quality assurance approach. In this chapter, we will describe the evaluation approach that has been taken to show the approach's suitability. The overall evaluation goal is directed towards different dimensions including a) the language's state of the art (language features) regarding the target concern *self-adaptivity*, b) the language's expressiveness, usability, and applicability as well as comprehensibility compared to plain UML, c) the language's extensibility to different domains, and d) a comparison to other languages with particular focus on composition techniques.

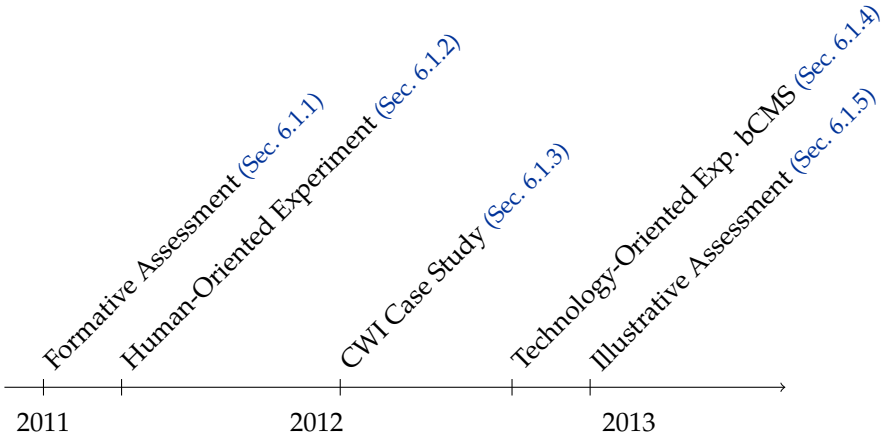


FIGURE 6.1.
All Evaluation Approaches on a Time Line

Therefore, the Adapt Case Modeling Language has been evaluated using a combination of different approaches including formative evaluations that perform evaluation and language engineering in parallel aiming at continuous improvement of the language. As shown in temporal order in Figure 6.1, the evaluations include

Two Assessments in Section 6.1.1 and Section 6.1.5. An assessment is the process of documenting characteristics of the evaluation subject using measurable terms. A formative assessment is carried out throughout the subject's engineering impacting decisions taken during engineering [Bos02]. A summative assessment is used to characterize the evaluation subject according to defined characteristics after it has been engineered.

Two Experiments in Section 6.1.2 and Section 6.1.4. An experiment is a well-defined, focused study for the establishment of a hypothesis [Bas07]. Human-oriented experiments make humans apply different treatments to objects, while in technology-oriented experiments, the focus is on technical treatments instead of humans [WRH+00].

An Illustrative Case Study in Section 6.1.3. An illustrative case study utilizes an instance of the evaluation subject to point out certain properties. Usually, case studies are aimed at establishing relationships between different attributes. A case study is an observational study while the experiment is a controlled study [WRH+00].

The first two evaluations (Section 6.1.1 and Section 6.1.2) have been carried out in the course of a master thesis [Bis11] the results of which will be sketched briefly with the corresponding design. The case study is described in terms of the resulting models which are discussed to point out several interesting findings. The technology-oriented experiment focuses on the comparison of using the ACML versus using plain UML. The illustrative assessment aims at comparing the ACML to other languages using a questionnaire. For all evaluations we mainly focus on the *evaluation design* description to enable a thorough repetition of the evaluations. It is part of obligatory future work to repeat these evaluations on a broader basis especially regarding size.

6.1 EVALUATION APPROACHES

To achieve the structure necessary for repetition of the evaluations, we establish a basic structure of the following evaluation descriptions that is based on an article by Robert K. Yin [Yin91]. It contains the following key elements:

- **Evaluation Design.**
  - **Evaluation Questions.** Description of the research questions that drive the evaluation.
  - **Evaluation Proposition.** Description of the propositions, a statement of assertion that expresses an opinion or judgment, related to the posed research questions.
  - **Units of Analysis.** Description of the concrete units that will be analyzed to answer the research questions.
  - **Linking Data to Propositions.** Description of how the resulting data can support the propositions made.
  - **Interpretation Criteria.** Description of the criteria used for interpreting the results.
- **Evaluation Execution.** (opt) Description of the execution of the evaluation and depiction of the created artifacts, models, or figures if the evaluation has been carried out.
- **Evaluation Results & Discussion.** (opt) Description of the results if the evaluation has been carried out and discussions of interpreted results.

All five evaluations are described in the following sections using this structure.

6.1.1 FORMATIVE ASSESSMENT: LANGUAGE FEATURES

The formative assessment approach was carried out as part of a master's thesis [Bis11] with the goal of assessing the language regarding the current state of the art for modeling self-adaptive software systems. In the following, we describe the assessment's design.

Assessment Design

Questions The assessment's main goal is to mirror the ACML at common sense of modeling self-adaptive software systems that has been pre-

sented in Section 2.2 and the original source requirements as presented in [LNGE11]. As a side goal of this assessment, basic usability criteria were used for the assessment of the language. Therefore, the assessment's research questions are as follows.

RQ1 Does the ACML meet common criteria posed by the research community for modeling self-adaptive software systems [ALMW09]?

RQ2 Does the ACML meet its original source requirements posed in [LNGE11]?

RQ3 Does the ACML meet common usability criteria given in [Moo09]?

Proposition The proposition is that the ACML either meets the criteria or provides good reasons for not meeting the criteria:

PR1 The ACML meets most criteria for modeling self-adaptive software systems posed by literature and all its source requirements. The ACML is usable considering common criteria.

PR2 Lack of fulfillment is due to design decisions regarding trade-offs.

PR3 Lacking features can be overcome by workarounds or may be added in future.

Units of Analysis The ACML is checked including all of its language features, e.g., adaptation actions. The criteria to mirror at are gathered from current literature that describes ontologies, meta-models or the like for self-adaptive software systems. The criteria are shown to be fulfilled by providing concrete examples.

Linkage of Data to Propositions The resulting data is the degree of requirement fulfillment (on a qualitative scale) with justifications given by example and explanations. The proposition is best fulfilled if all criteria are fulfilled by the ACML. Lacking fulfillments must be traced down to trade-off decisions. Lacking features must be redeemed with workarounds or descriptions how the feature can be included in future.

Interpretation Criteria For rating the fulfillment of criteria, we use an ordinal 5-level scale as shown in Table 6.1. Both descriptive and numeric values can be used, starting with 2 (unacceptable) as the lowest value and 10 (excellent) as the highest value.

The criteria's degree of satisfaction is *unacceptable* if a requirement cannot be fulfilled by the ACML at all. The degree is *excellent* if a requirement has been fulfilled by the ACML. If it is not fulfilled but can be fulfilled

in future with reasonable small effort, the degree of satisfaction is *acceptable*. If a requirement could be fulfilled in theory but with great effort, the degree of satisfaction is *moderate*. Finally, the degree is *good* if the requirement is not completely fulfilled but with small variations.

TABLE 6.1. Measurement 5-level Scale

Descriptive	Numeric
Unacceptable	2
Moderate	4
Acceptable	6
Good	8
Excellent	10

Assessment Execution

The assessment was conducted during a master's thesis. The results of the assessment have been valuable for creating and updating the language's design. However, due to the limited time and depth of a master's thesis, the assessment is considered as an *assessment sketch* here, and thus, is planned to be re-executed in future on a more thorough basis. In the following, we discuss the assessment's execution details followed by a discussion of results in the next section.

The different criteria that were used to create an assessment form were taken from three different sources. The first source are the modeling dimensions that have been proposed by Andersson et al. [ALMW09]. This source has been taken since the modeling dimensions are well-accepted in the SEAMS research community that deals with self-adaptive systems. Hence, these dimensions provide a solid base of the community's perception of self-adaptive systems and thus form a group of assessment criteria that judge about the adaptation description capabilities of a language. The second source that judges about a language's usability is the work about different principles for designing cognitively effective visual notations by Danial Moody [Moo09], a standard evaluation framework for usability of (visual) languages. Finally, the third source of criteria was the original statements that were made in the first paper about the ACML [LNGE11]. Thus, the ACML is checked against its own source requirements. The following list presents an excerpt of the inferred evaluation criteria (EC).

Criteria related to Adaptation Description These criteria origin from the modeling dimensions proposed by Andersson et al. [ALMW09]. They concern

the expression / modeling of self-adaptive systems. Five of the criteria are as follows.

EC2 *Flexibility.* To what degree allows the ACML to describe the level of uncertainty that is associated with system goals?

EC8 *Frequency.* Can the frequency of change occurrence (e.g., rare or frequent) be described?

EC9 *Anticipation.* Are different types of change prediction (e.g., foreseen or unforeseen) supported by the ACML?

EC11 *Autonomy.* Does the ACML allow for different degrees of outside/human intervention? (e.g., autonomous or assisted)

EC16 *Triggering.* Are different types of adaptation triggering supported? (e.g., event triggered or time triggered)

Criteria related to Language Usability These criteria stem from the cognitive dimensions framework proposed by Danial Moody [Moo09]. They concern the language's usability. Two of the criteria are as follows.

EC21 *Visual Expressiveness.* How many different visual variables [Moo09] are used?

EC24 *Perceptual immediacy.* How good is the association between concepts of the ACML and their notation?

Criteria related to the ACML's Source Requirements These criteria are concerning the first original source requirements of Adapt Cases as presented in [LNGE11]. Three of the criteria are as follows.

EC26 *Separation of Concerns.* To what degree does the ACML allow the separate description of adaptation logic and their relations to application logic?

EC28 *Integration / UML Consistency.* To what degree is the ACML consistent with the UML and UML-based processes?

EC29 *Control Loops.* To what degree are control loops expressible using the ACML?

Assessment Subject: Modeled Example Scenario Using these criteria or questions, particular language features were evaluated using the example of an adaptive rack server system. 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.

In the end, for each criterion, the answer could be equipped with a concrete example if fulfilled or counter examples or argumentations if not fulfilled.

Assessment Results & Discussion

The results were edited and presented using spider web diagrams as depicted in Figure 6.2. All spider web diagrams and details regarding their discussion can be found in [Bis11]. The results' investigation led to several findings which were used to further improve the ACML.

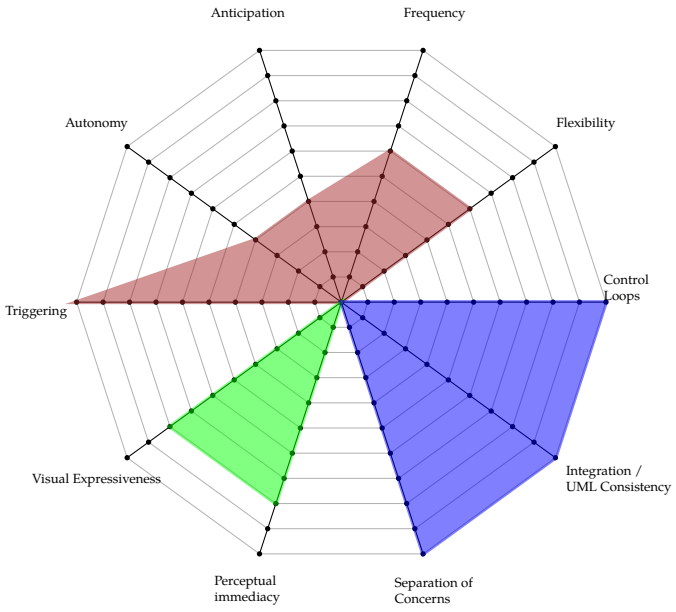


FIGURE 6.2. Spiderweb Diagram with a subset of the Assessment Criteria

All in all, the assessment and modeling using the example scenario showed the ACML to be expressive, usable, and fulfilling its requirements. Some shortcomings that have been identified could be eliminated during later language engineering iterations. A concrete example for shortcomings concerned the semantics of the ACML in general and the timing issues in mechatronic systems in particular. If for example an Adapt Case activates additional fans to alleviate the problem of high temperature, the effect of lowering temperature is delayed by physical properties. Using the original semantics, the Adapt Case would have been applied over and over again. Thus, we changed the semantics to not restart a monitoring activity before the corresponding adaptation activity has been finished. Thereby, the adaptation activity may use a time delay action to force the Adapt Case wait for the adaptation's effect before reapplying eventually. Concerning usability, the resulting data suggested to increase the notation's pop-out to ease the quick distinction of different elements on first sight. We introduced extra speaking icons for adaptation actions to follow this suggestion. A more detailed discussion of the assessment's findings can be obtained from [Bis11].

6.1.2 Human-Oriented Experiment: Usability & Expressiveness

The second evaluation approach involved a group of 10 students who used the ACML to model the adaptivity of a real-world web-based learning system. Focus of this experiment is usability & expressiveness of the language.

In the following, we briefly describe the experiment's design, execution, and results.

Experiment Design

Questions The main goal of the experiment was to investigate the usability and expressiveness of the ACML if applied by non-experts (knowledged and laymen). The two research questions are as follows:

RQ1 Is the ACML sufficiently expressive to model the example system, a web-based learning system?

RQ2 How is the usability and perceptual immediacy of the ACML (cf. [Moo09])?

Proposition The proposition is that the ACML is indeed applicable by non-experts and is sufficiently expressible to model the example system:

PR1 The ACML is sufficiently expressive to easily model the example system.

PR2 The participants are able to use the ACML without any further teaching.

PR3 The participants correctly use the language elements.

Units of Analysis To start the experiment, the ACML and an example self-adaptive software system, the *Adaptive Learning System*, are briefly described and handed out to the participants. The participants create models for the example system. These models are the main units of analysis. A supporting questionnaire about the expressiveness and usability is filled-in by the participants after creating the models. The filled-in questionnaires are units of analysis, too.

Linkage of Data to Propositions If the resulting models sufficiently describe the example system in the limited amount of time, the expressiveness and usability are said to be sufficient. Further, the filled-in questionnaires explicitly elicit the strength and weaknesses of the ACML regarding usability and expressiveness.

Interpretation Criteria Again, for rating the satisfaction of our propositions, we use an ordinal 5-level scale that is shown in Table 6.1. Both descriptive and numeric values can be used, starting with 2 (unacceptable) as the lowest value and 10 (excellent) as the highest value.

The degree of satisfaction is *unacceptable* if the system could not be modeled, at all. It is *moderate* if the system was modeled but including severe flaws and thus is not understandable without asking the modeler. The degree of satisfaction is *acceptable* if the system was modeled but includes flaws that can be traced back to the ACML itself. The degree is *good* if the system was modeled correctly but more circuitous or complex as it could be. Finally, the degree of satisfaction is *excellent* if the system was modeled correctly and straight-forward.

Experiment Execution

The experiment was performed with a group of 10 master students who where recently starting a project group that dealt with the extension of the ACML for process-oriented systems. Thus, the students had a rough knowledge of the ACML while the level of knowledge strongly differed among the group. The group was asked to model an adaptive learning system, a web-based system that adapts the difficulty level of learning material according to the student's knowledge which can be inferred from his grades and the exercises performed so far. After having modeled the system, the students were asked to fill out a questionnaire that was meant to capture the students comprehension and mood while using the ACML. Questions of the questionnaire included the following:

1. Is the border between functional and adaptive behavior clear?
2. Can the ACML describe **time** triggered adaptation?
3. Can the ACML describe **event** triggered adaptation?
4. Can the ACML model consequences of adaptation failures?
5. How good can the ACML model adaptivity?
6. How good can the notation be mapped to the problem world?
7. How difficult is the distinction of symbols in the ACML?
8. How difficult is it to understand models created with the ACML?

Both the models and the filled-in questionnaires were used for analysis.

Experiment Results & Discussion

Again, the language proved to be sufficient in modeling the adaptive learning system. This experiment's main result was that the ACML's operations seemed to be to low-level. Therefore, we supported the creation of reusable compound operations that reflect typical adaptation pattern. Another important finding was that using the ACML was way easier when the MAPE-K pattern was known. More precisely, teaching the modeler in the four MAPE phases helps with the distinction of the monitoring and adaptation activities. This correlates with finding presented in [WIS13]. A detailed report and additional findings are given in [Bis11].

6.1.3 CASE STUDY CWI: EXTENSIBILITY & APPLICABILITY

The CWI case study is about a car window insurance process. This case study has been developed with an industrial partner and is closely related to one of their projects. The CWI process was designed to be self-adaptive and was modeled with a domain-specific extension of the ACML. Focus of this study was the extensibility and applicability of the ACML for a specific domain, i.e., process-oriented service-based software.

Case Study Design

Questions The main concern of this case study is the extensibility of the ACML towards a specific domain. In particular, this case study investigates the modeling of process-oriented self-adaptive systems. An important fact to investigate is whether the ACML is sufficiently generic to allow domain specialization without breaking existing features of modeling or quality assurance. The research questions are as follows.

RQ1 Is it possible to extend the ACML with domain-specific elements to support a specific domain and ease specification?

RQ2 Is it possible to model a process-oriented self-adaptive system using the extended ACML?

Proposition The propositions are twofold just like the research questions and basically are about to positively answer the questions:

PR1 It is easily possible to extend the ACML with domain-specific elements, e.g., using UML Profiling mechanisms.

PR2 The resulting language allows to model process-oriented self-adaptive systems in a domain-specific manner.

PR3 The extension does not break any language or quality assurance features.

The first proposition handles the generic case and is refined by the second proposition which relates to process-oriented self-adaptive systems. The last proposition considers the case of proper genericity of the ACML.

Units of Analysis The ACML is extended to address process-oriented self-adaptive systems and applied to a case study, named *Car Window In-*

surance (CWI). The resulting extended language and possibly extended quality assurance techniques are subject of analysis.

Linkage of Data to Propositions The amount of changes to be made to the original ACML must be minimal or zero to fulfill the requirement for easy extension (PR1, RQ1). Further, extending the language should not break any methods or techniques that have been created for the ACML. Therefore, the existing quality assurance techniques shall be used with the extended modeling language. If all techniques are still applicable, proposition PR3 is fulfilled, if not, research question RQ1 must be answered negatively. Finally, the created models for the Car Window Insurance project may support proposition PR2 and research question RQ2.

Interpretation Criteria Research question RQ1 is positively answered if the number of changes necessarily to be applied to the ACML is zero and the number of ACML & QUAASY features that are broken by extension are zero, too. Answering research question RQ2, again relies on subjective judgment in this thesis. If the modeled system can be represented using domain-specific means, RQ2 can be answered positively. The degree of satisfaction requires a more extensive evaluation.

Case Study Execution

The case study was performed by the project group MEPASO in our research group. A project group is a group of usually eight to 15 students who together conduct a project that lasts one full year. The task of the project group MEPASO [Res12] was to create a modeling and quality assurance workbench for self-adaptive process-based service compositions. Therefore, the students reused and extended the ACML by domain-specific elements to model the adaptation of the process and the service bindings. Further documents related to the results of the project group can be obtained from the website at [Luc13].

In particular, the ACML is extended to specify the adaptation of processes, i. e. the behavioral description of software systems. Therefore, the process modeling language BPMN [Bus09] was incorporated into the ACML and Adapt Cases were extended to allow specification of adaptation of BPMN models. The extended Adapt Cases are named Business Process Adapt Cases (BPAC) and basically consist of domain-specific actions that allow the manipulation of BPMN processes. Further, the QUAASY approach was used and slightly extended to enable the assurance of self-adaptive BPMN processes.

The realistic software project that was used for the evaluation of the ACML

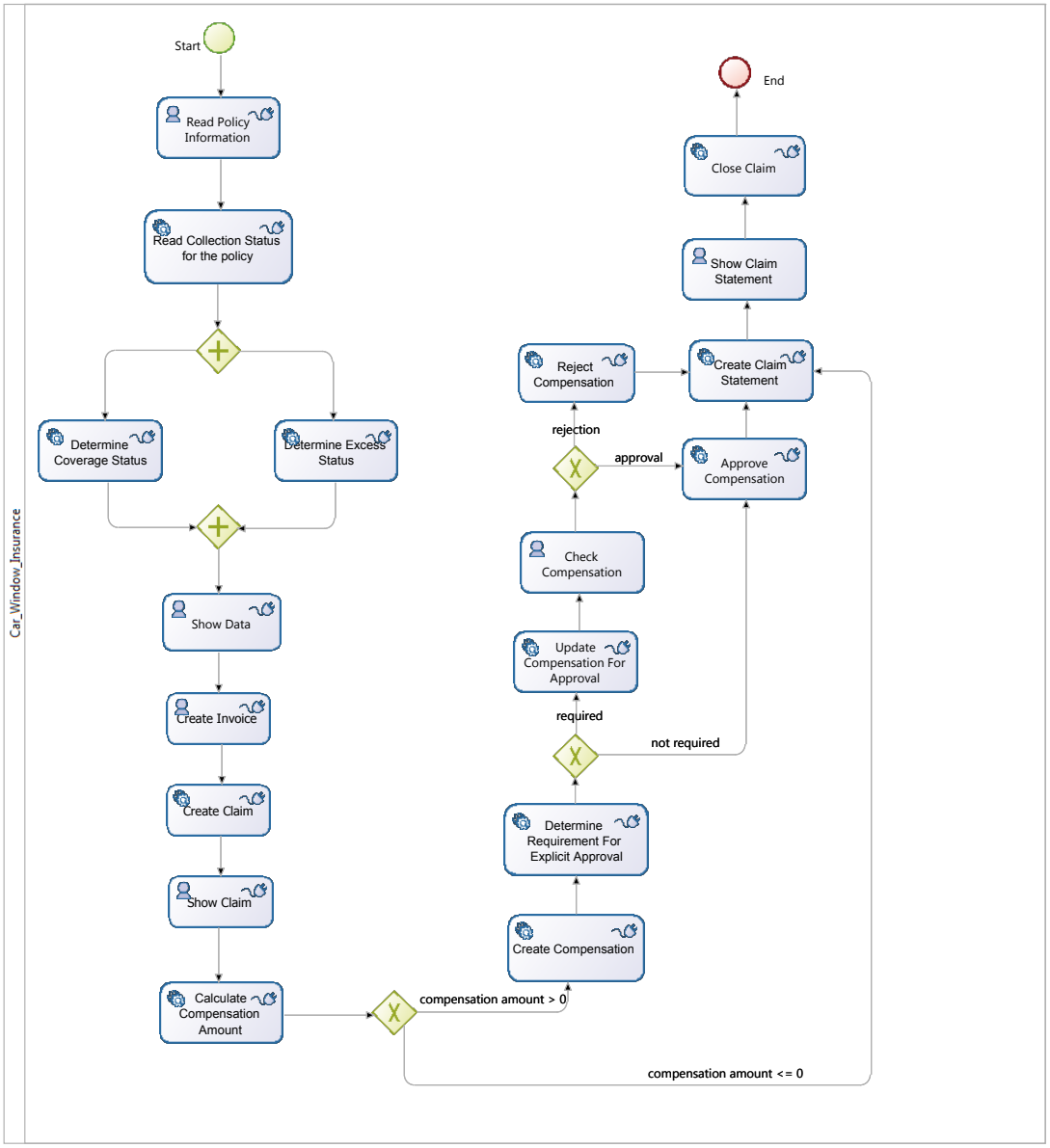


FIGURE 6.3.
CWI Business Process taken from [RRM+12]

was given by one of our industrial project partners and originates from the domain of car window insurance.

Figure 6.3 shows the business process that describes the car window insurance system. The process contains human and automatic tasks. The human tasks are supported by user interfaces, e.g., using a portal server. Automatic tasks are processed by attached web services. As such, the process orchestrates the use of several web services and input of data by human using specific portlets within a portal server.

In case of web service failures, the self-adaptive process-based system shall automatically exchange the used web service. Therefore, several Business Process Adapt Cases have been defined one of which is shown in Figure 6.4. The BPAC accepts *ServiceNA* signals and checks whether the *ClaimsRulesService* that is attached to the action *Read collection status for policy* is not available anymore. If so, the adaptation activity is triggered. This activity first retries the web service at hand. If not successful, the service is exchanged and executed. If no other web service is available, a human task is inserted into the process instance instead. All actions used are domain-specific actions that have been defined within the ACML extension *BPAC* specifically for adapting BPMN processes.

A more detailed description of the models created within this case study as well as several other Business Process Adapt Cases are given in [RRM+12] which can be found on the website at [Luc13].

Case Study Results & Discussion

The project was particularly interesting because of its large size and therefore the findings related to the modeling complexity and the state space explosion problem.

First of all, the case study showed that it is indeed possible to extend the ACML with domain-specific elements. In fact, the extension is not difficult at all, since UML profiling mechanisms can be used and the ACML is sufficiently generic to be extended for a specific domain. Thereby, we could answer research question RQ1 positively. Further, the models showed that it is possible to model the by industry provided case study in a domain-specific manner. Especially, the additionally provided domain-specific actions and the two fixed partitions (process layer partition and service layer partition) support the creation of domain-specific models. Thus, research question RQ2 can be answered positively, too. However, the extension did break some of the quality assurance features of the generic QUAASY approach, such that a few quality properties could not

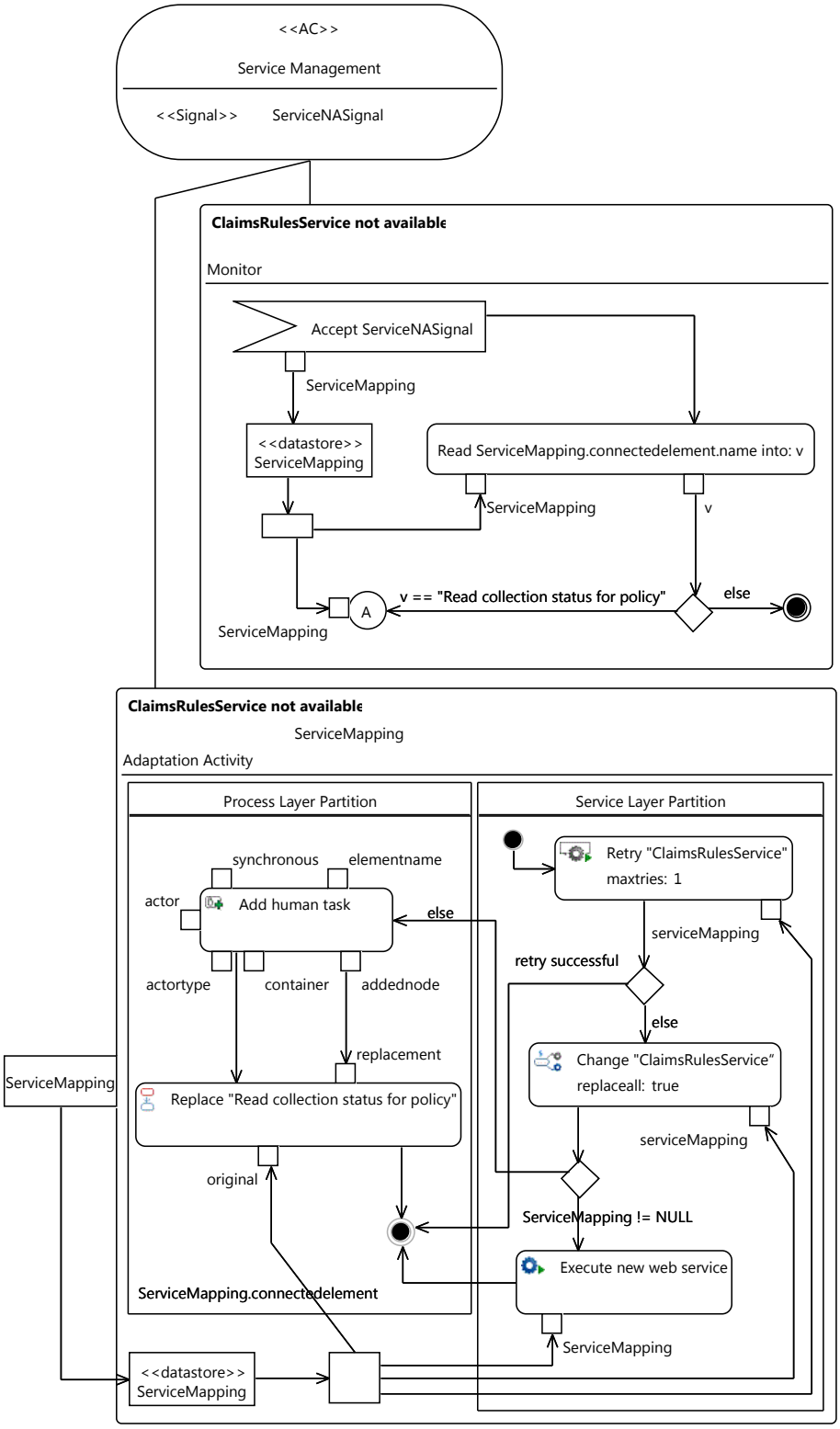


FIGURE 6.4.
BPAC Claims Rules Service Not Available taken from [RRM+12]

be shown or proved any more. Thus, the quality assurance approach QUAASY was slightly adjusted by the project group and made even more generic later on.

All in all, the case study provided by the project group was very successful in showing the usability and extensibility of the ACML within a very domain-specific application scenario. The two industrial partners involved in the project group plan to utilize the results generated by the project group.

6.1.4 Technology-Oriented Experiment bCMS: Applicability & Comprehensibility

The bCMS experiment involves creating a set of models first using the ACML and second using plain UML. The resulting models are compared regarding comprehensibility. When this experiment was first executed, the ACML models have been submitted as case study to the CMA workshop 2012. The interesting part about this study was that originally, the bCMS system was not explicitly defined to be self-adaptive. Hence the case study helped identifying and defining the engineering process for self-adaptive software systems and especially the adaptation related tasks for the case that a system specification with mixed concerns exists. This experiment was conducted by a modeling and language expert who is highly knowledged concerning the ACML. This is because this experiment was designed to evaluate the language from a technology point of view where human aspects such as skill level should rather be neglected.

Experiment Design

Questions The experiment's main goal is to show the applicability of the ACML to an arbitrary large software-intensive system. To a certain degree, the experiment shall answer the question of efficiency the language introduces for self-adaptivity design, i.e., its comprehensibility over plain UML. The research questions are as follows.

RQ1 Is the ACML's expressiveness sufficient to model a large software-intensive system?

RQ2 Is the design of the system more easy to create, better to under-

stand, and easier to maintain than without the ACML?

Especially the the second question RQ2 is hard to support quantitatively and/or objectively. Nevertheless, the experiment shall give some indicators why research question RQ2 can be answered positively.

Proposition The proposition is that both research questions can be answered positively. Especially regarding question RQ2, we expect good results regarding increased comprehensibility of models. To support the propositions, the experiment has to provide models that use the ACML for describing self-adaptivity features and other models that use plain UML for description. We infer the following two propositions.

PR1 The ACML is able to model self-adaptivity features of a large software-intensive system.

PR2 The models created with the ACML are easy to create, understand, and maintain.

Units of Analysis The ACML is used to create models for the bCMS experiment [CCG+12]. The resulting models are analyzed and compared to plain UML models that have the same meaning. Further, the models are submitted to the *Comparing Modeling Approaches (CMA) Workshop 2012*, the reviews of which are considered for analysis.

Linkage of Data to Propositions If the created models are reviewed and accepted by the CMA'12 program committee, the ACML is said to be able to model the system easing understandability and thus maintainability. The models created should show the difference of using or not using the ACML, e.g., by directly comparing two versions of specifications.

Interpretation Criteria The criteria for interpretation regarding research question RQ1 are given by the external reviews provided. The comparison of models for answering RQ2 should be empirically supported on its own. Since the ACML heavily relies on UML activities, valid metrics would be the complexity based on the number of decision nodes, hierarchy levels, loops, etc. However, in this thesis, we rely on the subjective judgment of the author and the CMA'12 workshop's reviewers regarding the understandability of the created models compared to the use of plain UML. The use of well-known and usability-improving-accepted paradigms such as *separation of concerns* and control loops [WIS13] in the ACML further supports the positive answer of RQ2.

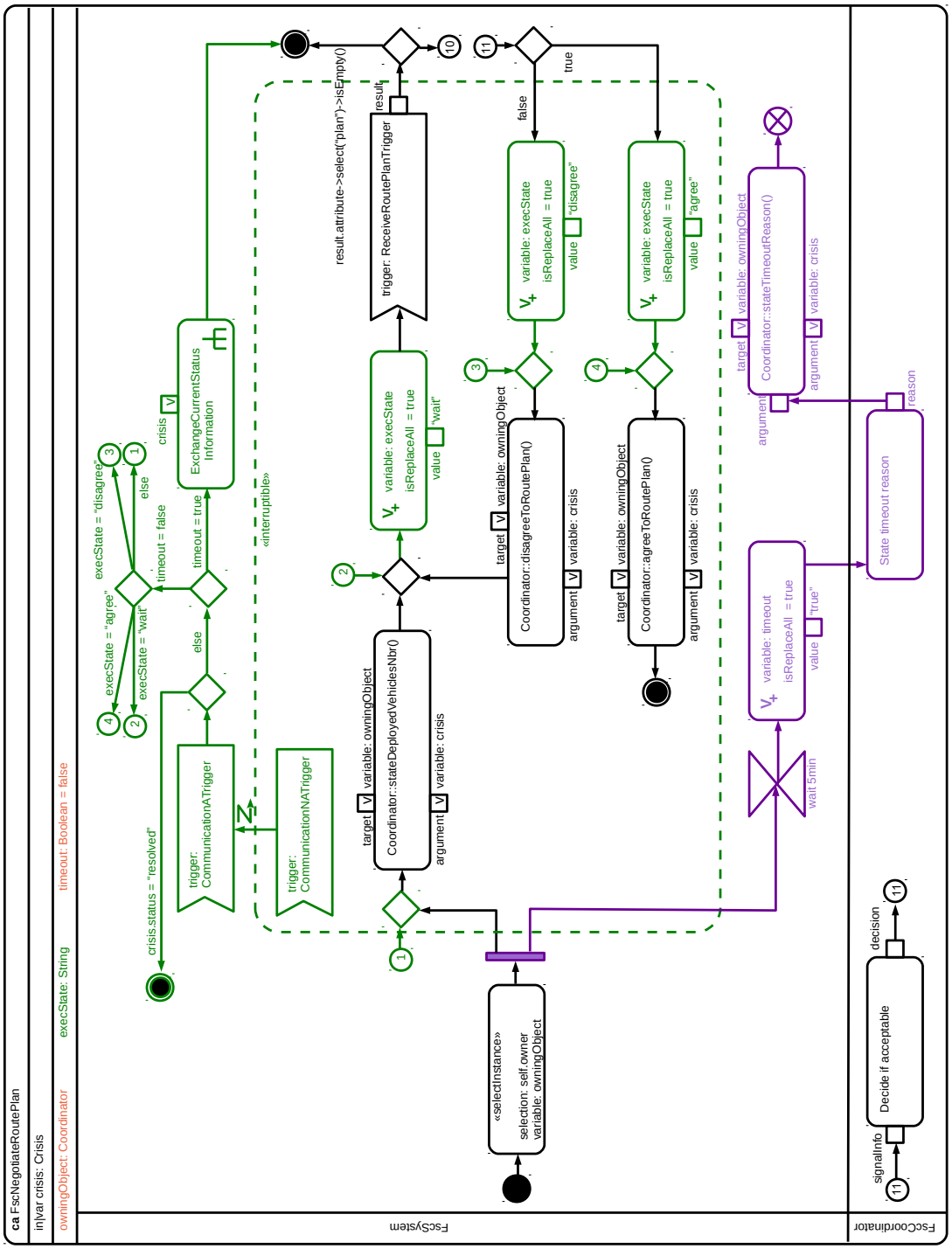


Figure 6.5.
Classical Modeling: FSC Route Negotiation

Experiment Execution

For the execution of this experiment, the bCMS experiment that was briefly described in Section 2.1 and in [CCG+12] was extensively modeled in detail and submitted for acceptance for the CMA'12 workshop [LM12]. The models were formally reviewed by three reviewers and based on these reviews it was accepted for presentation.

In the following, we will show a small subset of the submitted models to give an intuition of their contents and to show the difference to classical modeling, i.e., without using the ACML. The complete set of submitted models can be obtained from [LM12] and the website at [Luc13].

Figure 6.5 schematically shows the activity that describes the route negotiation process within the bCMS experiment from the perspective of a fire station coordinator (FSC). The activity has been modeled without using the features of the ACML and without separating adaptation and application behavior. The activity's main functionality is described by the black colored actions that state the number of deployed vehicles, receive a route plan from the police station coordinator (PSC), and either agree or disagree to that route plan. The actions that require human interaction are contained in the lower swim lane *FSCCoordinator*, the remainder of the activity is performed automatically. The main functionality's actions are contained in an interruptible region that together with the green colored actions take care of the communication availability and in case of a *non-availability period* pause and continue the process. The purple colored actions take care of the timeout requirement, i.e., they cancel the activity if the route plan has not been agreed on within a period of 5 minutes.

As encoded with colors, this activity mixes different kinds of functionality. Especially, the functionality of handling communication availability and timeouts can be considered as self-adaptive behavior.

Figure 6.6 shows the core actions of the FSC route negotiation activity. The remaining self-adaptivity behavior has been separated using Adapt Cases, e.g., as shown in Figure 6.7 for communication availability. The Adapt Case waits for *communication not available* signals and if received starts the adaptation activity *Switch CMS Coordination off*. This activity uses a policy to pause all processes (see [LM12] for details). Likewise, the reception of a *communication available* signal leads to adapting the process again, i.e., continuing the process. Obviously, separating the adaptation behavior from the remaining system specification helps understanding the models.

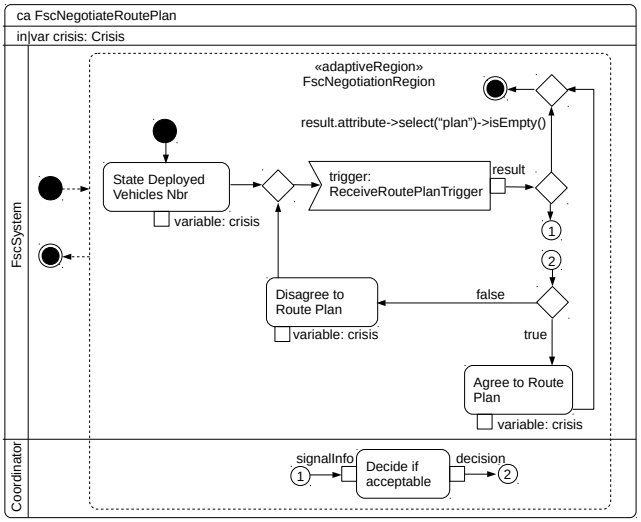


Figure 6.6.
ACML Modeling: FSC Route Negotiation

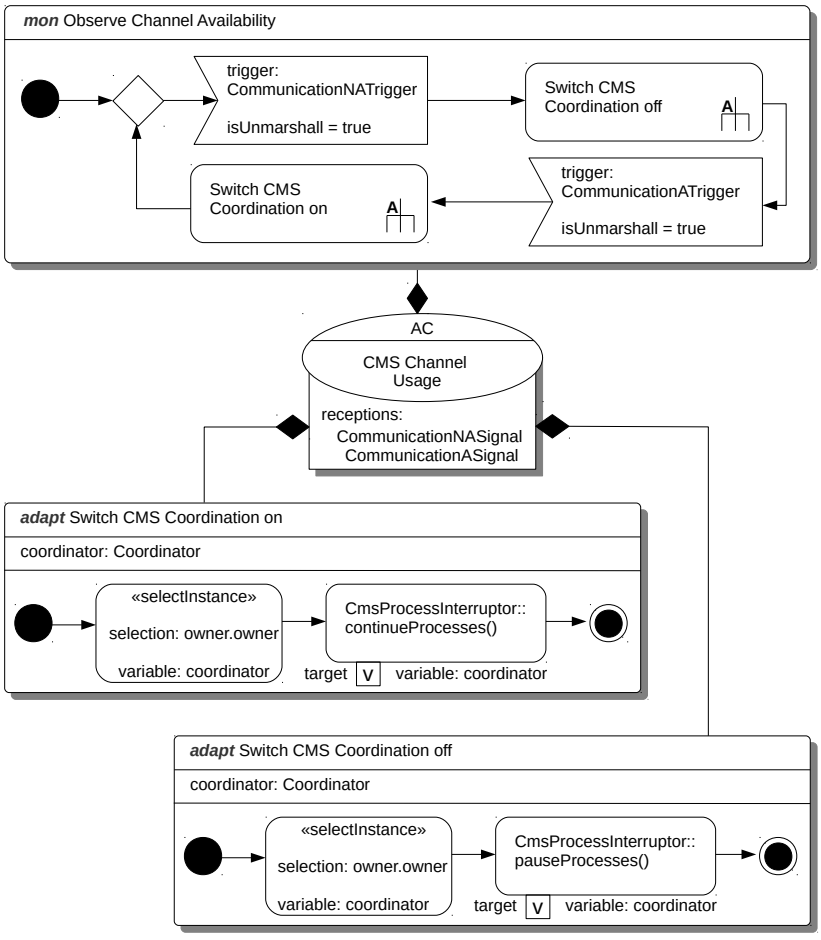


Figure 6.7.
Adapt Case: Communication Availability

Experiment Results & Discussion

In general, the reviewers' feedback for the submitted models was very positive. To give an example that reflects the reviewers' mood, consider the following citation of an unknown reviewer: "Adapt Cases is suited to define adaptivity in object-oriented high-level architecture and low level design." All three reviews accepted the models what we interpret as a positive reaction to research question RQ1 and partially to research question RQ2.

Concerning research question RQ2, we performed an even more thorough comparison of using the ACML or not during a master thesis (cf. [Mut12]). Several differences are obvious. While classical modeling mixes concerns, the ACML allows the separation of adaptivity concerns from the remaining specification. Using the ACML, functionality description does not overlap nor must it be weaved together at design time. Separation of adaptivity concerns greatly reduced complexity as indicated in Figure 6.8. The different colors describe auxiliary behavior that is spread over the system description. With the ACML, the different auxiliary behavior has been extracted and grouped together which eases comprehension of both the auxiliary behavior (i.e., adaptation behavior) and the core system behavior.

Besides all the advantages of using the ACML, it requires a few more steps during the development process. That is, more models have to be created and refined. An Adaptation View Model must be created as a view onto the system model and an Adapt Case Model must be created to describe adaptation behavior. Both kinds of models must be refined and kept up to date in addition to the classical standard models. In turn, however, the system models decrease in size and complexity since auxiliary functionality is now externalized.

Moreover, the behaviors defined in the ACML ease the understanding of what is actually happening since they reflect concern-specific actions. Actions that *pause*, *continue*, or *stop* a process are more easy to grasp and understand than the same functionality expressed using signals and exceptions as it would be without the additional ACML actions. Further, concern-specific notation such as effectors and sensors showed to be useful to clearly specify and understand the self-adaptive software system.

Another downside of using the ACML is the necessity of externalizing information to make it available for external Adapt Cases. That is, information must be provided to external model units such as Adapt Cases by the use of sensor and effector interfaces. Using the mixed version of the behavior descriptions, this information had been just in place.

Overall, the conclusion of this experiment is that the ACML helps to create more readable, focused, and understandable versions of the system specification. This is mainly due to the separation of concerns and the possibility to easily build up adaptation action hierarchies (i.e., build high-level actions from low-level actions) and foster reuse without losing comprehensibility. Table 6.2 gives a short comparison of classical UML modeling and ACML modeling (based on [Mut12]).

Table 6.2. Comparison of Classical UML Modeling and the ACML

	Classical Modeling		ACML Approach
	Development process from high-level to low-level design		Development process from high-level to low-level design
–	Mixed definition of different system concerns	+	Separate definition of adaptivity concern
–	Distribution of concerns	+	Concentration of adaptivity concern
–	Only low-level generic (unintuitive) modeling elements	+	Intuitive concern-specific low and high-level modeling elements
		–	Additional Models
		–	Additional information retrieval required

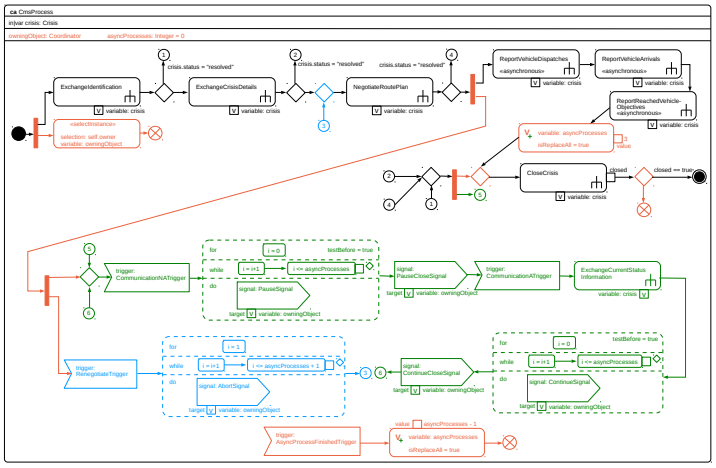
6.1.5 Illustrative Assessment: Composition Techniques

This illustrative assessment was conducted for submission to the Comparing Modeling Approaches (CMA) workshop 2012. The assessment focuses on used composition techniques in the ACML and allows comparison with plenty of other languages that have been assessed.

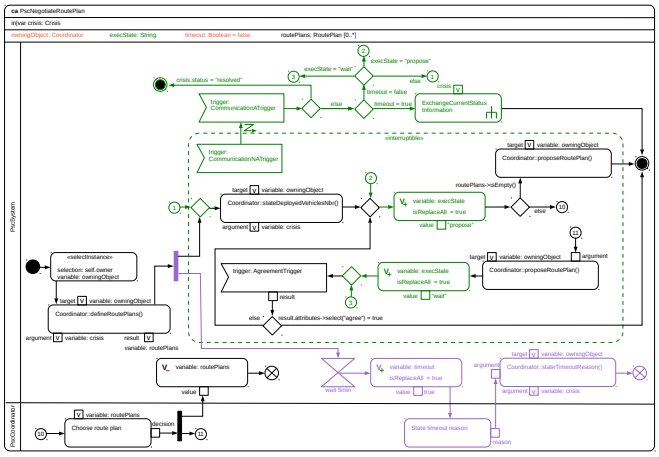
Assessment Design

All documents related with this assessment can be obtained from the website at [Luc13].

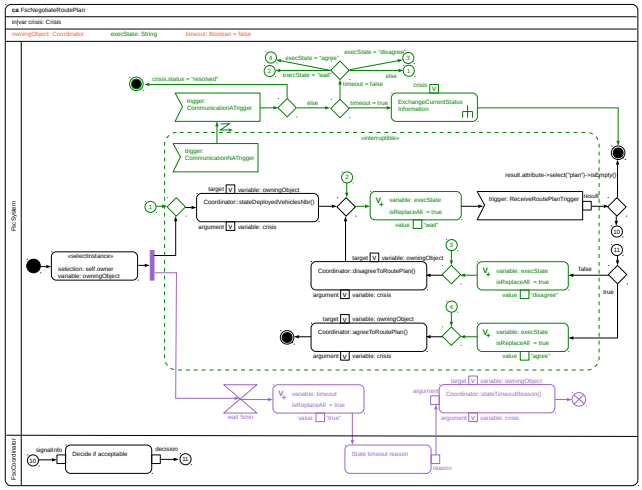
Questions The main goal of this assessment is a comparison of language characteristics in general and composition features in specific. Thus, we formulate our research question as follows.



(a) CMS main process



(b) PSC route negotiation process



(c) FSC route negotiation process

FIGURE 6.8.
Classically modeled processes of the bCMS system with color coded functions (green = communication availability, purple = negotiation timeout, blue = renegotiation, red = mixed) taken from [Mut12]

RQ1 What are the language's characteristics (see catalog in [GAA+13]) compared to other languages with particular focus on composition techniques?

Units of Analysis The ACML is assessed using the comparison criteria catalog [GAA+13] that was developed at the Bellairs Research Workshop in 2011[1] and 2012[2]. A questionnaire based on the comparison criteria catalog was filled out for several different modeling languages. The results are analyzed and compared.

Assessment Execution

The assessment lead to several tables and figures that allow the comparison of the ACML to other languages. Figure 6.9 shows a table that characterizes the ACML (Adapt Cases) concerning its composition operators.[3] This and similar tables have been filled out for the ACML and 14 other languages for the CMA'12 workshop.

Figure 6.10 shows the UML languages that are used by the ACML (Adapt Cases) and the other assessed modeling languages. The ACML uses UML use cases, components, activities, and classes as well as OCL and Ecore. The figure nicely shows how the ACML is related to other modeling languages and where intersections can be used to merge two or more approaches.

Assessment Results & Discussion

Detailed results and discussions for this assessment have been published in [MAA+12]. In short, the assessment allowed to group and relate the different assessed languages to each other. Furthermore, the modeling approaches have been contrasted with each other in terms of their software development phases and activities, paradigms and level of formality, and their use of composition rules and operators. As such, the assessment provides a first good comparison of different modeling approaches and allows to characterize the ACML in detail. The complete process of developing the comparison criteria and assessing the approaches took more than 2 years and greatly influenced the design of the ACML. For more information, please refer to the documents on the website at [Luc13].

[1] http://www.cs.mcgill.ca/~joerg/SEL/AOM_Bellairs_2011.html
[2] http://www.cs.mcgill.ca/~joerg/SEL/AOM_Bellairs_2012.html
[3] Please note that the *Context Model* in the table (Question 2.2.H) corresponds to the Adaptation View Model.

FIGURE 6.9. Assessment Form concerning Composition Operators

Adapt Case **2. Key Modeling Concepts** **2.2 Composability: composition rules and composition operators**		1) Adapt	2) ApplyAdaptation
B. What is the name of the composition?			
D. Is the composition a:	Composition Rule	x	
	Composition Operator		x
E. If the composition is a composition rule, what other compositions are used to realize the composition defined by the composition rule?		(2)	
H. State the signature of the composition…	**Adapt)** Adapt Case Model x Context Model → Context Model' **ApplyAdaptation)** Adapt Case Model x Context Model → Context Model'		
I. Does the result of the composition contain a modeling element that does not exist in the source models (i.e., does the composition add new model element(s) to the source models)?	Yes		
	No	x	x
J. Does the composition realize one or more composition rules?	Yes		x
	No	x	
K. What is the mechanism for identifying the inputs for the composition?	Explicit	x	x
	Pattern Matching		
	Binding		
L. What is the mechanism for applying the composition?	Explicit	x	x
	Implicit		
M. Is the composition:	Symmetric		
	Asymmetric	x	x
N. Is the composition:	Syntax-based		
	Semantics-based	x	x
O. Is the composition:	Deterministic	x	x
	Probabilistic		
	Fuzzy		
P. If the composition is a binary composition operator, what algebraic properties does the composition operator provide?	Commutativity		
	Associativity		
	Transitivity		
Q. If the composition specification is a composition operator, does the composition operator produce models that are closed under the operator?	Yes		x
	No		
R. Is the intent of the composition to address crosscutting concerns?	Yes	x	x
	No		
S. Is it necessary for a language of the modeling approach to support an explicit ordering of the composition?	Yes		
	No	x	x
T. Is the composition itself separated from the specification of first class entities?	Yes		
	No	x	x
U. How is a composed model intended to be presented to the modeler by a tool? The composed model is intended to be…	… shown with automatic layout.		
	… shown without automatic layout.		
	… shown by annotating the original model.		
	... not shown.	x	x

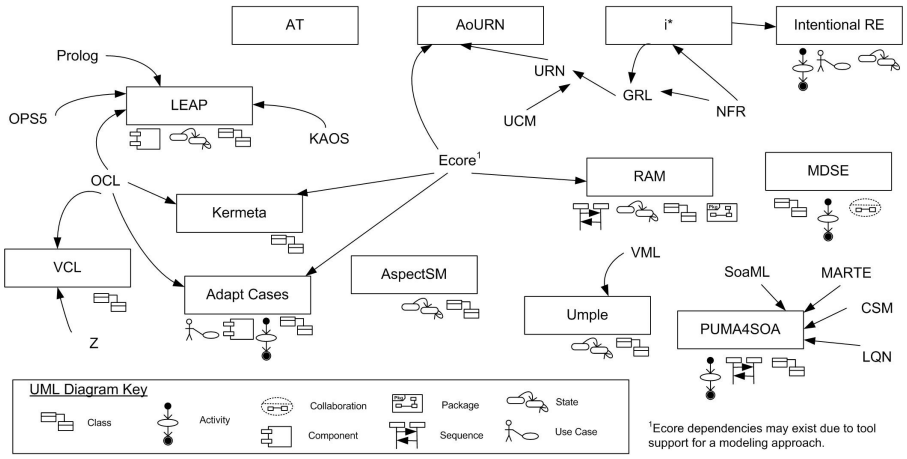


Figure 6.10. UML Languages used by the ACML

Threats to Validity 6.2

This section discusses the threats to the validity of the presented evaluations, i.e., influencing factors that may affect the validity of the evaluations' results. There are two different classification themes for threats to validity. The first scheme distinguishes between internal and external threats to validity [CSGS69]. Cook et al. [CC79] further extend the two by conclusion and construct validity. We will use the latter scheme. Before describing the four types of threats in detail, let's first consider the model in Figure 6.11 that describes the principles of experiments as presented in [WRH+00].

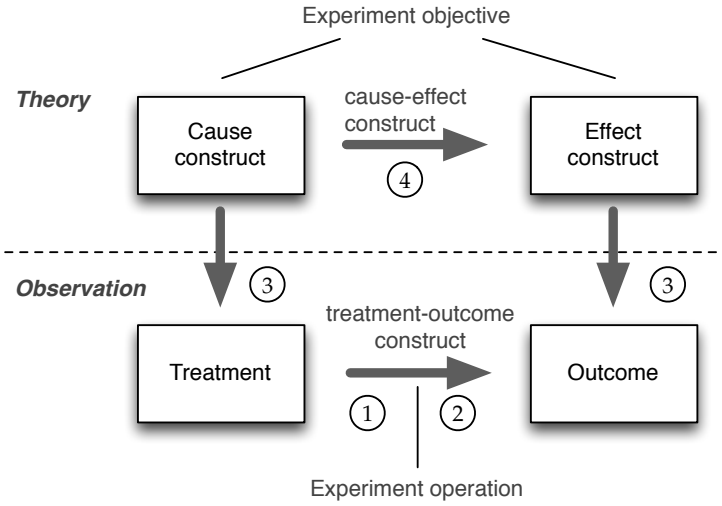


Figure 6.11. Principles of Experiments (from [WRH+00])

The figure's upper part describes the theory of an experiment, i.e., the idea of a cause-effect relationship. The theory about the cause-effect relationship is

represented by defining a hypothesis. The lower part describes the experiment that is performed to support the theory, i.e., to test the hypothesis. Therefore, one or more treatments that represent the cause are defined and executed, and the outcome is analyzed for its degree of matching the theoretic effect.

This construct includes several threats to the experiment's validity each of which can be assigned to one of the numbers that label the arrows. The numbers correspond to the four classes of validity that may be threatened as described in the following [WRH+00].

① **Conclusion Validity.** This validity concerns the relationship between treatment and outcome. In an experiment we have to check that that is a statistical relationship between the treatments that have be applied and the corresponding outcome. Examples include the statistical test chosen and the care taken in the measurement of an experiment.

② **Internal Validity.** The conclusion validity is given, i.e., there is a statistical relationship between treatment and outcome, the internal validity makes sure that the relationship is only dependent from variables that are under control. That is, if internal validity is given, there is not only a relationship between treatment and outcome in the experiment, but the treatment actually causes the outcome. An example is the compensation of subjects or the treatment of subjects if special events occur.

③ **Construct Validity.** This validity is concerned with the relation between the theory and the observation. It is given, if the treatment reflects the cause construct and if the outcome reflects the effect construct. For example, if as treatment for experience in a programming language the number of taken university courses is chosen, this might affect the construct validity.

④ **External Validity.** External validity is concerned with the generalization of experiment outside the scope of the study. External validity is always given if it is generalizable to the *target* scope, i.e., if the target group are students, then the external validity is perfectly given if the study subjects are students as well. If the target group are experienced professionals then using bachelor students as subjects might be a threat to external validity.

In the following, we will enumerate the most important threats to validity that were existent in our evaluations. Since the evaluation's descriptions given above are geared towards repetition, the following enumeration of threats can be considered as threats that have to be taken care of when repeating the evalu-

ations on a broader basis. The threats have been taken from [WRH+00] where several threats are listed to be used as checklist.

Conclusion Validity

Fishing Searching or *fishing* for particular results is a common threat to validity. In our evaluations, we address this threat by selecting criteria that are based on well-accepted research results and by involving third persons when interpreting the results (e.g., reviewers of conference/workshop submissions). Still, especially for attributes such as usability, fishing remains a risk that has to be taken care of.

Low Statistical Power The more power a statistical test has, the more valid are the patterns that can be found in gathered data. Low statistical power is a high risk in how the evaluations described above were performed since hardly any statistical test were applied. Instead, for data analysis mostly subjective judgments were used (cf. Table 6.1). We addressed this threat by providing a detailed scale and precise descriptions of when a particular measure shall be taken. However, especially for the technology-oriented experiment described in Section 6.1.4, further metrics have to be defined to measure and statistically compare the difference between the usage of ACML over plain UML.

Reliability of Measures Conclusion validity is depending on the reliability of measures. In our studies this is a threat of high risk since most measures were subjective rather than objective. This negatively affects comparability and thus repeatability.

Reliability of Treatment Implementations Another threat is the risk of having chosen the wrong evaluation criteria, especially when applying the first formative assessment. We addressed this threat by relying on the modeling dimensions that are well-known and accepted in the research community in self-adaptive systems. Still, there is the residual risk of having picked the wrong criteria.

INTERNAL VALIDITY

In our evaluations we did not plan to use control groups. Hence there are no threats of heterogeneity between study groups. On the other hand, we cannot determine if the treatment or any other factor caused the effect. Hence, when repeating the evaluations, it is advised to define control groups.

Instrumentation There is always a residual risk of having bad designed instruments such as questionnaires and the like. To address this threat, we had several review cycle for our instruments.

Other than that, there are no other general threats to internal validity. Of course, demoralization, rivalry, etc. has to be taken care of per experiment.

CONSTRUCT VALIDITY

Mono-operation Bias This threat concerns the amount of variables, cases, subjects, or treatments that affect construct validity. Of only single cases are studied, the cause construct might be underrepresented. This is an important existing threat in the past executions of the evaluations presented above since the number of example systems that were used for investigation (especially within the case studies) is rather small. However, in total we have four different systems that have been modeled using the ACML and more are about to follow.

Hypothesis Guessing The subjects chosen in the past executions of the evaluations described above usually knew about the tested hypothesis. This might affect the outcome either positively or negatively depending on their attitude to the hypothesis. In a repetition, it should be taken care that subjects do not know about the hypothesis and may not easily guess the hypothesis.

Experimenter Expectancies Of course, the experimenter himself might influence the experiments results, even unconsciously. In the past executions of the described evaluations, a single person advised the different evaluation approaches. However, this threat was addressed by not interfering too much with the ideas of the master students who prepared and executed the case studies. Further, the bCMS experiment had been supported by three different review-

ers and a large group of researchers during the CMA'12 workshop. Thus, this risk is supposed to be very small.

External Validity

Interaction of Selection and Treatment This threat concerns the fact that the subject groups may not be representative of the group we want to generalize to. In out evaluation that was targeted at laymen concerning the tested modeling language, most of the modelers already had knowledge in using the ACML. For most of the users, there had been an introduction to the ACML before performing the evaluations. We addressed this threat by performing the evaluations that required students to create models as early as possible and thus had a user group where only a few students already worked with the ACML while other solely listened to the talk about the ACML. Still, there is a small residual risk that the user group is not representative for laymen.

Interaction of Setting and Treatment This is the threat of having picked non-representative settings or material for experimentations. However, by aiming at more example systems from industry, we also address the threat of non-representativity of the chosen example systems. However, by now, the capabilities of generalization of the results is rather limited until more (large) systems have been modeled.

All in all, on the one hand, there are several threats to the validity of the past executions of the described evaluations that have to be addressed with care when repeating the experiments on a broader basis in future. On the other hand, many other threats that are listed in [WRH+00] are already addressed in the design of the presented evaluations.

Discussion & Future Work 6.3

Discussion The different evaluation approaches were used in a formative manner—that is during the language's design—to iteratively refine and enhance the language. As such, all evaluation greatly contributed to the language's design. Further, the evaluation approaches provided evidence for the

appropriateness of the language and allow the comparison of the language with other existing languages. Therefore, all evaluation approaches were designed to be repeatable. Of course, this is not always feasible but provides a basis for first objective and comparative evaluation results for the ACML.

Future Work The future work regarding the ACML's evaluation should be targeted towards a more extensive, thorough, and detailed evaluation that is based on user feedback. Therefore, the language should be used by different and large user groups with different background to obtain more detailed insights in the strengths and weaknesses of the ACML. In particular, the experiment construction has to be valid in terms of *preoperational explication of constructs*. That is, all constructs shown in Figure 6.2 have to be explicitly defined, the theory has to be clear, the treatments have to be sound, and the statistical test have to be chosen or constructed with care.

7

Tool Support

"I'm a great believer that any tool that enhances communication has profound effects in terms of how people can learn from each other, and how they can achieve the kind of freedoms that they're interested in."

– Bill Gates

7

The concepts presented in this thesis have been prototypically implemented in various different approaches. See Figure 7.1 for an overview. In Section 7.1, we show the graphical editors and tree editors for the ACML (cf. Section 3) and the BPAC approach (cf. Section 6.1.3). This includes editors for specifying the Adaptation View Model, the Adapt Case Model, business processes, service pools and Business Process Adapt Cases (BPAC). In Section 7.2, we present the tools that implement the QUAASY approach and the extended version that has been implemented for the BPAC approach. Finally, in Section 7.3, we conclude this chapter. All implemented prototypes can be downloaded from the website at [Luc13].

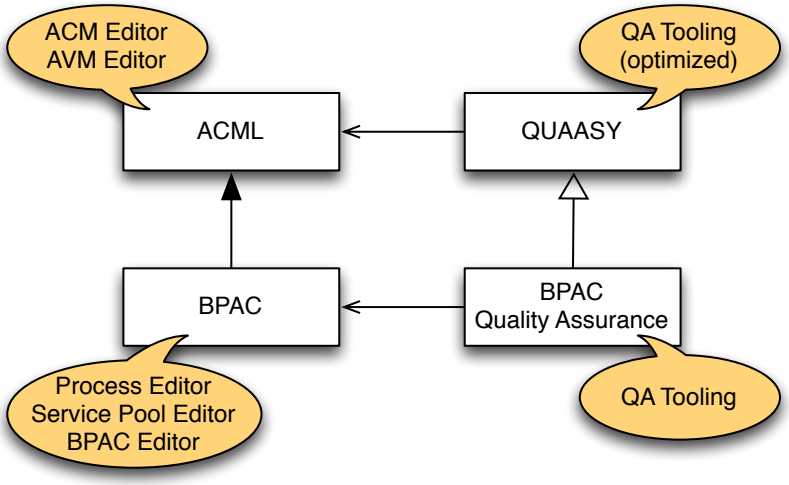


Figure 7.1. Prototypical Tool Implementations

Modeling of Self-Adaptive Systems 7.1

Since the complete tooling is based on the Eclipse Modeling Framework [BBM03], the generation of simple tree editors is straight-forward once a language's meta model has been defined using *ecore*. These editors can be further configured, equipped with validations, and icons.

A sample tree editor is shown in Figure 7.2. The editor's structure reflects the standard EMF editor structure. The left area contains a project explorer that contains all projects, models, and other files. The right area contains the model tree where new elements can be created using the context menu. The bottom area contains the properties view where the properties of model elements can be set. As such, all possible valid instances of the underlying ecore meta model can be created using this editor.

Of course, tree editors are visually not very appealing. Further, the relationships between elements are often hidden in the tree structure or even worse in the properties view. Thus, it is very helpful to provide graphical diagram editors as shown in the following.

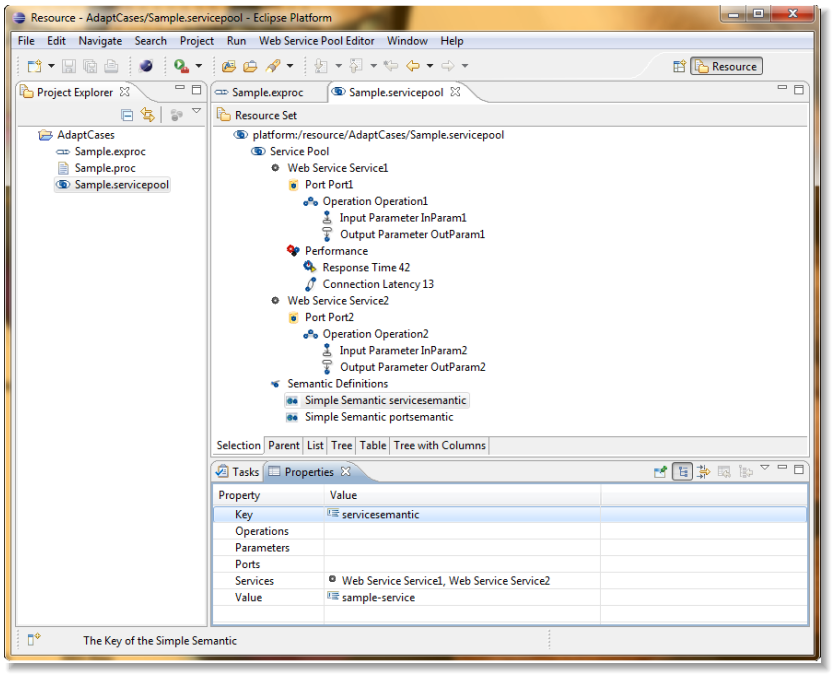


Figure 7.2. Generated and Extended EMF Tree Editor for Service Pools

Graphical editors have been created for the ACML and the BPAC extension. Both languages consist of several model kinds such as the Adapt Case Model and the Adaptation View Model in the ACML. For both of these two model

kinds, graphical diagram editors have been provided the first of which is depicted in Figure 7.3.

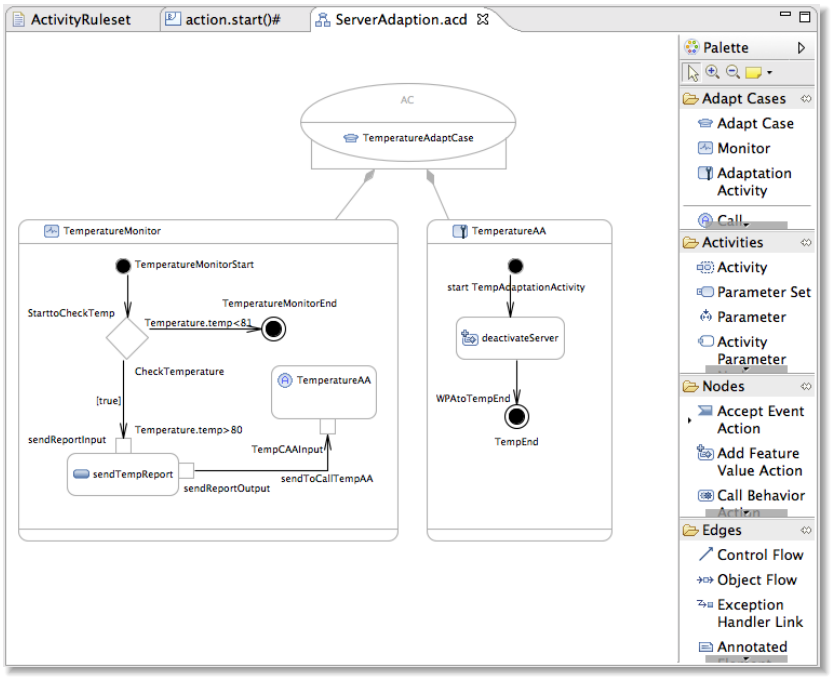


FIGURE 7.3. Graphical Adapt Case Model Editor

The figure only shows the diagram pane. Of course, the editor also provides a project explorer and a properties view as shown in Figure 7.2. In addition, graphical diagram editors usually have a palette that contains the model elements that can be instantiated by the use of drag and drop. The figure shows a single Adapt Case that is specified with a monitoring activity on the left and an adaptation activity on the right. One Adapt Case Model diagram may contain several Adapt Cases. Of course, Adapt Cases may be spread in several different diagrams. Elements from the Adaptation View Model can be referenced by using the properties view.

The Adaptation View Model is created using the corresponding diagram editor shown in Figure 7.4. The figure shows the Adaptation View Model that describes the rack server system that has been described in Section 6.1.1. Again, the palette shows the elements that may be instantiated in the diagram. Since the underlying meta models of the Adapt Case Model and the Adaptation View Model are linked to each other, elements that have been instantiated within the AVM diagram can be referenced in the ACM diagram.

For the domain-specific BPAC extension of the ACML, new graphical diagram editors were created based on the extended meta model. It would be possible to create a UML profile for the ACM diagram editor shown in Figure 7.3, however unfortunately, this features was not available when the BPAC editors had

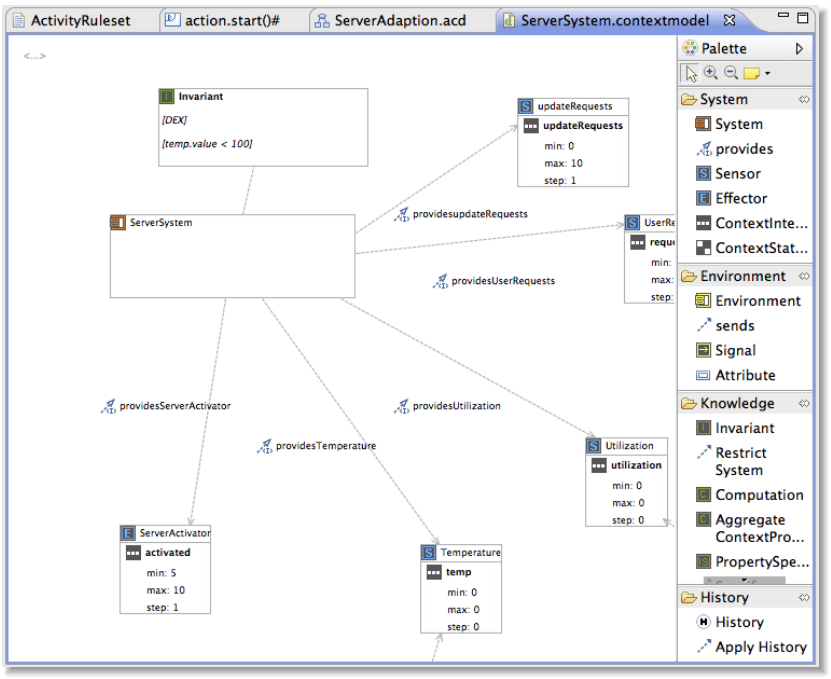


Figure 7.4.
Graphical Adaptation View Model Editor

been created. Thus, the diagram editors have been created from scratch using the Papyrus framework [Pap13]. Of course, on meta model level, these editors are still a valid ACML extension. Figure 7.5 shows the range of model kinds and corresponding editors that exist for the BPAC approach.

The process model shown in Figure 7.5 is an import of a Bonita Process Model created in Bonita, a BPMN-based modeling workbench [Bon13]. The web service pool model allows the definition of web services with operations containing input and output parameter. The corresponding editor has been shown in Figure 7.2 on Page 213. The process extension model allows to link process actions that are contained in the process model to web services that have been defined in the web service pool model. Again, the process extension model editor is a simple tree editor that has been customized concerning icons, etc. as shown in Figure 7.6.

Finally, the Business Process Adapt Case diagram editor that is shown in Figure 7.7 allows the graphical definition of BPACs that adapt the process model. In contrast to the original Adapt Case diagram editor, BPACs have a different adaptation activity that contains two partitions, the process layer partition and the service layer partition. The respective partitions may contain actions that either adapt the process definition (e.g., control flow, actions) or the service bindings (i.e., bindings of process actions to concrete web services). Further, the BPAC diagram editor contains the domain-specific model elements (e.g., actions) in its palette.

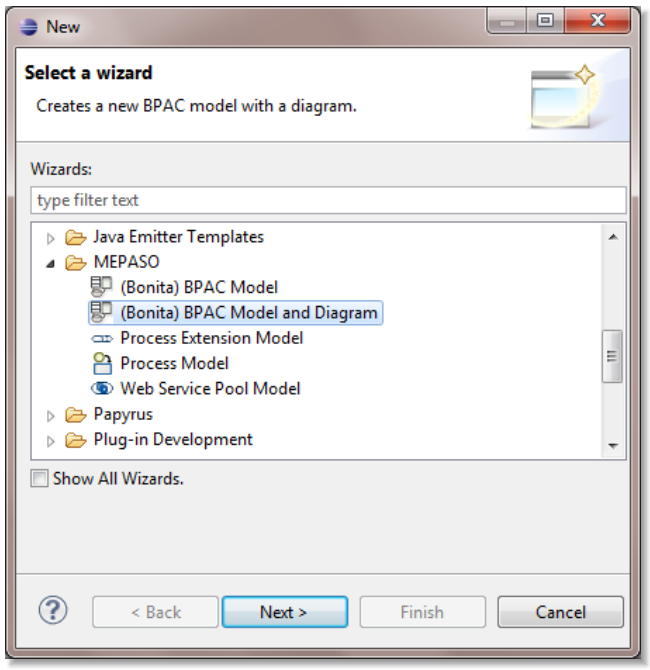


FIGURE 7.5.
Eclipse Wizard: BPAC Model Kinds

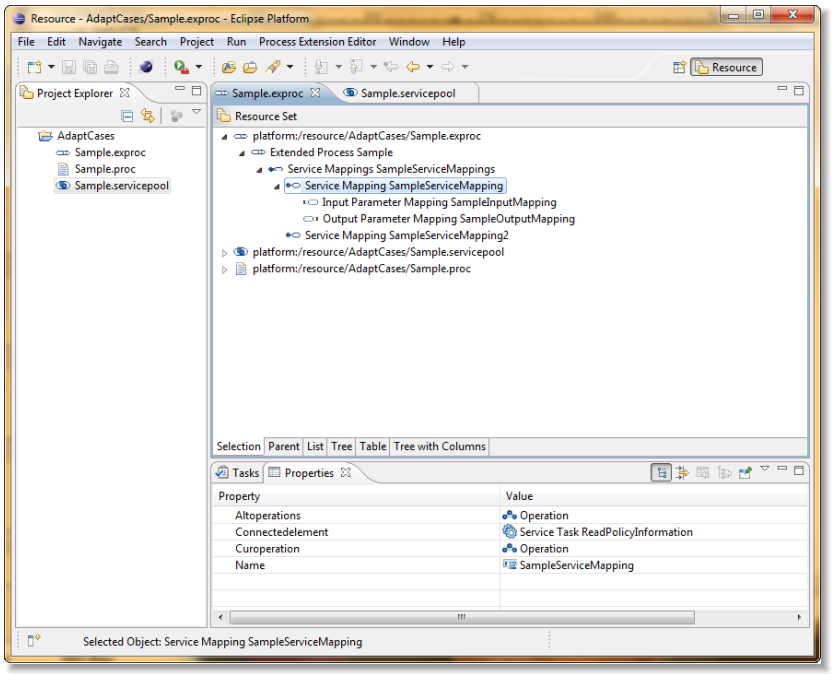


FIGURE 7.6.
EMF Tree Editor for Extended Process Definitions

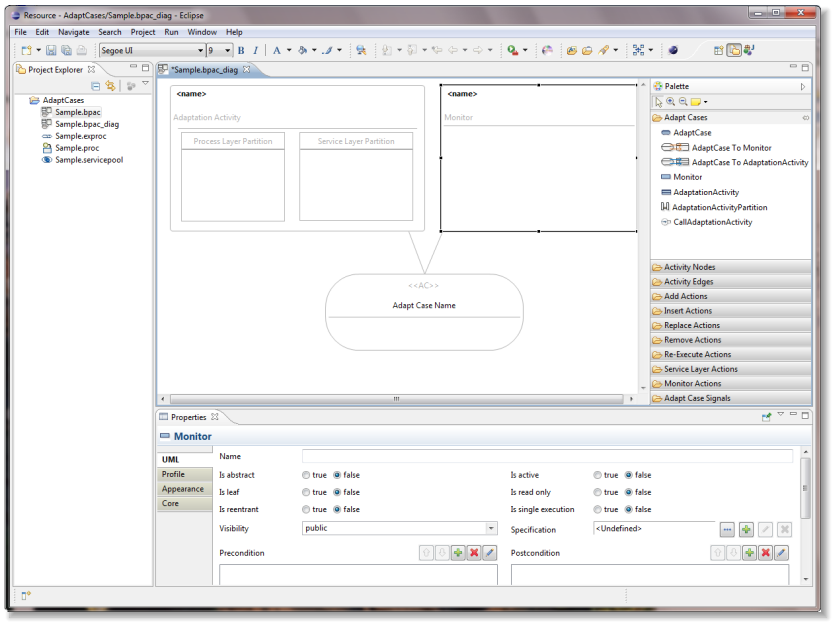


Figure 7.7. Graphical Business Process Adapt Case Editor

Quality Assurance for Self-Adaptive Systems 7.2

The quality assurance (QA) approaches for both the plain ACML and the BPAC extensions have been tool supported, too. While the BPAC quality assurance is completely supported graphically, by now the ACML quality assurance approach QUAASY uses the console for feedback provisioning. However, basically the techniques for feedback provision (i.e., the interpretation and representation of model checkers' counter examples) are identical, thus the BPAC QA techniques can be reused for QUAASY. Figure 7.8 shows how model checking is triggered for QUAASY and optimized QUAASY via the context menu.

Currently, the results shown on the console are as follows:

```
Counter examples found in setting: Boundary
Number of states: 61
Number of transtions: 95
Property: AFAG('system.perform()')
Counterexample: [s2, s3, s4, s5, s7, s11]
```

This is different from the BPAC QA approach. Again, the quality assurance is triggered using the context menu or a toolbar button. However, the feedback is interpreted before being represented to the user. That is, the counter example is translated back into the original model elements' names as shown in Figure 7.9. Two kinds of problems are shown. The first relates to a Business Process Adapt Case and states that the contained process containing the actions *ActionB* and *ActionA* contains a loop and therefore might not termi-

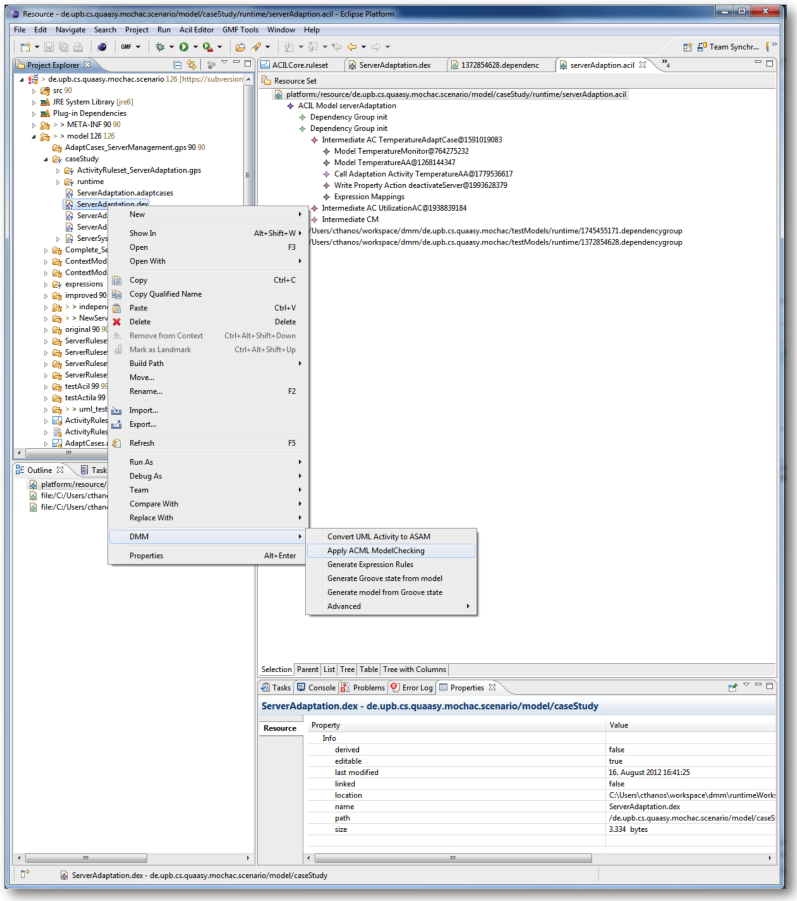


FIGURE 7.8.
ACML Model Checking Integration into Eclipse

nate. The second kind of problem relates to the business process that has been adapted. That is, after the adaptation has been applied, a loop has been detected and several elements cannot be reached any more. This error message indicates a design flaw in the BPACs.

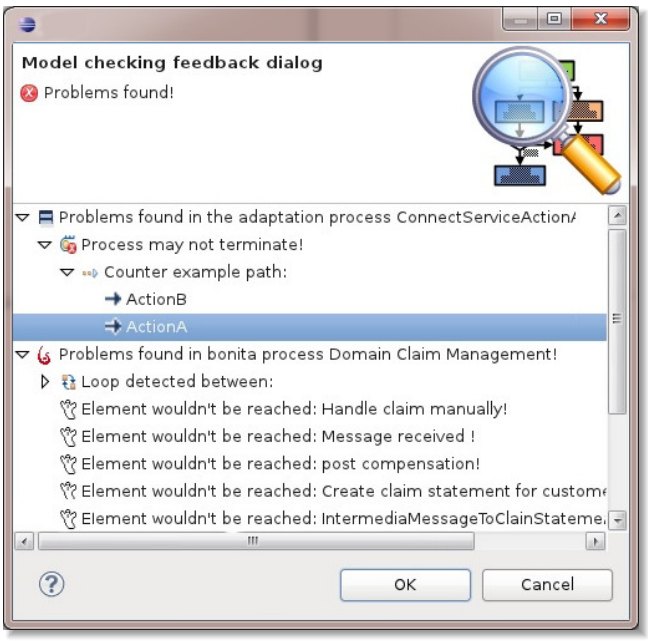


Figure 7.9. BPAC Quality Assurance Results

Conclusions 7.3

As briefly shown in this chapter, the ACML, QUAASY, and their extensions have been implemented prototypically to prove the concepts' validity. While the tools cover the languages' complete range they may not be convenient to use in productive environments. Thus, if the productive use of ACML and QUAASY is targeted in future, extensive efforts have to be spent into tool support development. However, for proving the concepts presented in this thesis, the described tools perfectly suffice.

8

Conclusions & Outlook

"Enjoying success requires the ability to adapt. Only by being open to change will you have a true opportunity to get the most from your talent."

– Nolan Ryan

8

FINALLY, in this chapter we conclude this thesis and discuss work that we think should be performed in future to advance the topic of engineering self-adaptive software systems. In Section 8.1 we discuss some general remarks on our research. Sections 8.2 and 8.3 discuss the two main contributions of the thesis, and finally, Section 8.4 points out future work.

8.1 GENERAL REMARKS

COMPLEXITY OF ENGINEERING SOFTWARE-INTENSIVE SYSTEMS INCREASES

The increasing complexity of software-intensive systems is to a great portion due to the increasing amount of uncertainty a system is expected to deal with. In software development, uncertainty is addressed by including alternative behavior into the software system that kicks in when the standard behavior does not suffice any more. However, with the increasing demands for resilient systems, these alternative behaviors are prevalent in these systems. Likewise increases the complexity of handling all different alternative behaviors at the same time. An approach to tackle these challenges is to separate the specification of these alternative behaviors into so-called self-adaptation logic that monitors and adapts the system's standard behavior if not sufficient any more.

SEPARATING OF SELF-ADAPTATION LOGIC AND DOMAIN LOGIC

Separating self-adaptation logic from domain logic has several benefits including the one of increased possibility for focus and analysis. Especially the analysis attracts more and more importance since a self-adaptive system's behavior seems to be intelligently autonomous and difficult to comprehend and re-enact. As a consequence, the demand for providing hard guarantees arises. However, prerequisite for modeling and analyzing self-adaptive software sys-

tems are languages, methods, and techniques that actually allow the separated specification and analysis of self-adaptation logic.

In this thesis, the proposed two main contributions are (1) a modeling language for self-adaptive software systems named *Adapt Case Modeling Language* (ACML) that allows the separated specification of self-adaptation logic, and (2) a quality assurance approach named *Quality Assurance for Adaptive Systems* (QUAASY) that enables the analysis of the modeled system for specific adaptation-related design flaws.

ADAPT CASE MODELING LANGUAGE

The **ACML** extends the UML to create a concern-specific modeling language. Extending the UML, we gain the benefit that our language integrates with most standard software engineering processes that are compatible with the UML-like object-oriented modeling principles. The ACML was designed to be a general-purpose language for self-adaptive software systems, however as shown in Section 6.1.3, the language can be extended with domain-specific means, easily. Finally, the ACML covers most modeling dimensions [ALMW09] and adaptation features that are known from literature.

QUALITY ASSURANCE FOR ADAPTIVE SYSTEMS

The quality assurance approach **QUAASY** reuses and extends existing techniques to model-check object-oriented design models. That is, the UML semantics proposed by the OMG are completely implemented and extended for self-adaptive software systems. Thus, our approach allows the analysis of self-adaptive software systems in an object-oriented paradigm using the well-accepted semantics provided by the UML specification [Obj11].

For both contributions we will discuss the main achievements in the following two sections.

Modeling Approach for Self-Adaptive Systems 8.2

The modeling language ACML is particularly interesting from the perspective of *a*) the *adaptation concepts* used and supported by the language, *b*) the paradigm *separation of concerns* that was used all over the language, and *c*) the modeling language implementation details that allow for easy integration and reuse.

Concepts The ACML widely uses concepts from the literature. For instance, the first class entities, the control loops, are shown to be accepted and, most importantly, very successful in [WIS13]. Further, as shown in Chapter 3 the language supports most of the modeling dimensions for self-adaptive software systems that have been proposed in [ALMW09]. Moreover, the language supports the various different levels of modeling languages such as adaptation of instances and type definitions (i.e., models). Thus, on the one hand, using the language it is possible to describe meta-adaptation, that is, adaptation rules that described the adaptation of other adaptation rules. On the other hand, allowing for adaptation on type level often leads to a cleaner design since alternative solutions do not have to be included in the system model, at all. The concepts, the language uses, have been evaluated and compared to the concepts of different other languages as shown in Chapter 6. That way, it was assured that the language corresponds to the current state of the art in modeling languages and, most importantly, addresses the users' needs. In particular, this was achieved by the use of proven principles and techniques such as *separation of concerns* and the *Unified Modeling Language* (UML).

Separation of Concerns According to Edsger W. Dijkstra in 1974 [Dij82], separation of concerns (SoC) allows "focusing one's attention upon some aspect". Since focusing the attention to the concern of self-adaptivity is desired to tame complexity, we heavily made use of this principle in the design of the language. Thus, not only is the adaptation specification separated from the remaining domain specification, but also the adaptation specification itself separates different sub concerns. As such, the adaptation rule definition (i.e., Adapt Cases) is decoupled from the adaptation interface definition (i.e., sensors and effectors within the Adaptation View Model). A specific challenge of applying SoC is to have the resulting separated models composable with each other to form a complete system model including the adaptation capabilities. Thus, composition or at least its specification is an important requirement for modern language design. As shown in Chapter 6, the composition of ACML models with the core application specification was payed special attention. Further, using the SoC principle allows the creation of models that are specifically prepared for focused analysis as shown in Chapter 4. Finally, the ACML shows to be a language that is focusing on a particular concern (i.e., self-adaptivity) but is generic regarding the domain it is used for. Thus, the ACML is concern-specific but domain-independent.

Modeling Language The ACML is an extension to the UML which does not break any features of the UML. It rather seamlessly integrates with the UML

allowing to reuse all tools and techniques that can be used for the UML. Especially, as a UML extension, the ACML allows the application of UML profiles. Thus, the ACML can easily be extended to a particular domain by the use of profiles. Since the ACML heavily uses UML activities, its semantics are clearly and unambiguously defined and correspond to the well-known token-offer semantics that is known from the UML. The ACML is targeted at component-based modeling as components are first class entities of the Adaptation View Model. However, since the Adaptation View Model is basically extending UML interfaces it can be used with non-component-based approaches, e.g., by the use with UML class diagrams. Thus, the ACML is prepared for a wide area of application.

QUALITY ASSURANCE FOR SELF-ADAPTIVE SYSTEMS 8.3

The most important achievements of the quality assurance approach QUAASY include the analysis' particular features, the tool-encapsulation, and the optimizations addressing the state space explosion problem.

Analysis QUAASY allows for the analysis of concern-specific quality properties. This is particularly interesting since many standard and well-known (quality) properties get new importance and challenge because of the introduction of the self-adaptation concern. In particular, this applies to global properties that can be checked for every self-adaptive system which is being modeled with the ACML and that come equipped with QUAASY. Further, the approach allows for the definition of application-specific properties that allow to ensure particular invariants in the light of self-adaptation. The use of formal techniques such as model checking allows the provisioning of hard guarantees while still usable by non-experts.

Tool Encapsulation A particular goal of software engineering research is to provide software engineers with tools that allow sophisticated analysis without knowledge of formal methods. As such, QUAASY performs the model checking hidden from the user by encapsulating the model checking execution within the ACML modeling workbench. The resulting analysis reports are translated back into the concrete ACML representation allowing non-experts to analyze the models and interpret the analysis results.

Optimization The major challenge in applying model checking is the state space explosion problem. This problem has been addressed within QUAASY in two different ways. First, an intermediate language optimized for model checking (small non-verbose semantics) has been employed. The ACML models are automatically translated into that intermediate language which is even possible off-line after saving a particular model. Second, a multi-staged model-checking approach has been implemented that allows to stop model checking early if an error has been found. Within this multi-stage approach, models are investigated on different levels of abstractions providing different levels of confidence regarding correctness.

8.4 FUTURE WORK

Although, the ACML & QUAASY approach has been defined extensively, there is still lots of future work that will further advance the approach. The four main challenges include further *engineering process integration*, additional *language features*, a thorough *evaluation*, and mature *tool support*.

Process Integration The approach needs a better integration with different requirements engineering approaches, e.g., those that rely on goal-based methods. Since requirements, and goals in particular, are not part of the UML, the technical integration can only be achieved via UML profiles. However, it remains to thoroughly investigate how requirements on goal level relate to self-adaptation, and for instance, whether self-adaptation should already be expressed and specified on requirements level. Another challenge regarding the process integration of the ACML is its integration with technical implementation approaches such as Rainbow [GCH+04] or StarMX [AST09]. It has to be investigated to what degree the ACML specifications can be translated to the respective technical representation automatically. Finally, the integration with non-UML-based languages such as Palladio [BKR09] has to be investigated since this would open new opportunities regarding different analysis targets (e.g., performance analysis).

Language Features The ACML is built to be generic and to support as most adaptation scenarios as possible. This is achieved by providing a set of low-level adaptation operations that allow every kind of change on model instances

and model types. While for the BPAC extension (see Section 6.1.3), a set of higher-level operations has been defined, this could be further advanced by identifying common adaptation pattern and schemes and defining the corresponding monitoring and adaptation actions therefore. Building on UML actions, the ACML is perfectly prepared to define high-level operations by combining low-level operations in activities.

Evaluation Although, as shown in Chapter 6, the ACML and QUAASY have been evaluated extensively using different techniques, it would be very beneficial to have a large-scale thorough and sophisticated evaluation, e.g., a controlled experiment with different large user groups and different case studies or examples to further reduce the threats to validity and gather more information for improving the approach.

Tool Support Finally, the best method is not worth it without good tool support. The tool support that was created for the ACML and QUAASY is a prototypical prove of concept implementation that, unfortunately, is not suited for productive use. Thus, future work should include the maturization of tool support including graphical feedback support, a graphical trace language for activities (application-specific properties), and a graphical state definition for invariants over system structure (application-specific properties) driven at high usability. For best results concerning interoperability with other approaches and tools (e.g., the full UML), the tool support should be fully included into the Eclipse Papyrus Environment [Pap13] or the like.

List of Figures

List of Tables

List of Definitions

Bibliography

[ABB+13] J. Andersson, L. Baresi, N. Bencomo, R. Lemos, A. Gorla, P. Inverardi, and T. Vogel. Software Engineering Processes for Self-Adaptive Systems. In R. Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 51–75. Springer, Berlin/Heidelberg, 2013. (Cited on page 173.)

[ALMW09] J. Andersson, R. Lemos, S. Malek, and D. Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2009. (Cited on pages 25, 181, 182, 223 and 224.)

[ASSV07] R. Adler, I. Schaefer, T. Schuele, and E. Vecchié. From Model-Based Design to Formal Verification of Adaptive Embedded Systems. In *Proceedings of the 9th International Conference on Formal Methods and Software Engineering (ICFEM '07)*, Lecture Notes in Computer Science, pages 76–95, Berlin/Heidelberg, 2007. Springer. (Cited on pages 116, 117 and 118.)

[AST09] R. Asadollahi, M. Salehie, and L. Tahvildari. StarMX: A Framework for Developing Self-Managing Java-based Systems. In *Proceedings of the 4th International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '09)*, pages 58–67, Los Alamitos, CA, USA, 2009. IEEE Computer Society. (Cited on pages 8, 59 and 226.)

[Bas07] V. R. Basili. The Role of Controlled Experiments in Software Engineering Research. In *Proceedings of the International Conference on Empirical Software Engineering Issues: Critical Assessment and Future Directions*, Lecture Notes in Computer Science, pages 33–37, Berlin/Heidelberg, 2007. Springer. (Cited on page 179.)

[BBM03] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003. (Cited on pages 44, 76 and 213.)

[BCGR09] M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. Definition of the System Model. In *UML 2 Semantics and Applications*, pages 61–93. John Wiley & Sons, Inc., 2009. (Cited on page 52.)

[Bec11] M. Becker. Context Model for Adaptive Systems. Master's thesis, Research Group of Databases and Information Systems, University of Paderborn, Germany, 2011. (Cited on page 81.)

[Bis11] A. Biser. Evaluation of Adapt Cases. Master's thesis, Research Group of Databases and Information Systems, University of Paderborn, Germany, 2011. (Cited on pages 179, 180, 184, 185 and 187.)

[BK11] B. Bartels and M. Kleine. A CSP-based Framework for the Specification, Verification, and implementation of Adaptive Systems. In *Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS '11)*, pages 158–167, New York, NY, USA, 2011. ACM. (Cited on pages 116, 117 and 118.)

[BKR09] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, January 2009. (Cited on page 226.)

[BLB12] M. Becker, M. Luckey, and S. Becker. Model-driven Performance Engineering of Self-Adaptive Systems: A Survey. In *Proceedings of the 8th International Conference on Quality of Software Architecture (QoSA '12)*, New York, NY, USA, 2012. ACM. (Cited on page 15.)

[BLB13] M. Becker, M. Luckey, and S. Becker. Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time. In *Proceedings of the 9th International Conference on the Quality of Software Architectures (QoSA '13)*, New York, NY, USA, 2013. ACM. (Cited on pages 15, 117, 158 and 169.)

[BMSG+09] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, Berlin/Heidelberg, 2009. (Cited on page 6.)

[Bon13] BonitaSoft. Bonita Open Solution. http://www.bonitasoft.com, 2013. (Cited on page 215.)

[Boo94] G. Booch. *Object-oriented Analysis and Design with Applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994. (Cited on page 36.)

[Bos02] C. Boston. The Concept of Formative Assessment. ERIC Digest. October 2002. (Cited on page 179.)

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. (Cited on page 36.)

[BSE11] N. Bandener, C. Soltenborn, and G. Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2011. (Cited on page 138.)

[Bus09] Business Process Management Initiative. Business Process Modeling Notation (BPMN) Version 2.0. Technical report, Object Management Group (OMG), January 2009. (Cited on page 189.)

[CBE+10] F. Christ, J.-C. Bals, G. Engels, C. Gerth, and M. Luckey. A Generic Meta-Model-based Approach for Specifying Framework Functionality and Usage. In J. Vitek, editor, *Proceedings of the 48th International Conference on Objects, Models, Components and Patterns (TOOLS'10)*, volume 6141 of *Lecture Notes in Computer Science*, pages 21–40, Berlin/Heidelberg, June 2010. Springer. (Cited on page 15.)

[CC79] T. D. Cook and D. T. Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin New York, 1979. (Cited on page 203.)

[CCG+12] A. Capozucca, B. H. Cheng, G. Georg, N. Guelfi, P. Istoan, and G. Mussbacher. bCMS - Requirements Definition. http://www.cs.colostate.edu/remodd/v1/content/bcms-requirements-definition-0, 2012. [Online; accessed 28-February-2013]. (Cited on pages 7, 21, 194 and 196.)

[CDE+03] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet,

J. Meseguer, and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, *Proceedings of the Conference on Rewriting Techniques and Applications (RTA '03)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer, June 2003. (Cited on page 62.)

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 1 edition, June 2000. (Cited on page 25.)

[CG12] S.-W. Cheng and D. Garlan. Stitch: A Language for Architecture-Based Self-Adaptation. *Journal of Systems and Software*, 85(12):2860–2875, December 2012. (Cited on pages 74, 75 and 76.)

[CGS05] S.-W. Cheng, D. Garlan, and B. Schmerl. Making Self-Adaptation an Engineering Reality. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. Moorsel, and M. Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2005. (Cited on pages 6 and 25.)

[CKTW08] M. V. Cengarle, A. Knapp, A. Tarlecki, and M. Wirsing. A Heterogeneous Approach to UML Semantics. In P. Degano, R. Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models*, pages 383–402. Springer, Berlin/Heidelberg, 2008. (Cited on page 52.)

[CLG+09] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 1–26, Berlin/Heidelberg, 2009. Springer. (Cited on pages 61, 75 and 106.)

[Coc00] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000. (Cited on pages 9, 40 and 81.)

[CSBW09] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A

Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, Lecture Notes in Computer Science, pages 468–483, Berlin/Heidelberg, 2009. Springer. (Cited on pages 8, 59, 162 and 163.)

[CSGS69] D. Campbell, J. Stanley, D. Gampbell, and J. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Rand McNally Chicago, 1969. (Cited on page 203.)

[Dav93] A. M. Davis. *Software Requirements: Objects, Functions and States*. Prentice Hall PTR, New Jersey, 2nd edition, 1993. (Cited on page 11.)

[DDF+06] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A Survey of Autonomic Communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, December 2006. (Cited on page 48.)

[Dij82] E. W. Dijkstra. EWD 447: On the Role of Scientific Thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982. (Cited on pages 8 and 224.)

[DMSFR10] G. Di Marzo Serugendo, J. Fitzgerald, and A. Romanovsky. MetaSelf: an Architecture and a Development Method for Dependable Self-* Systems. In *Proceedings of the Symposium on Applied Computing (SAC '10)*, pages 457–461, New York, NY, USA, 2010. ACM. (Cited on page 172.)

[DNGM+08] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications. *Automated Software Engineering*, 15(3-4):313–341, December 2008. (Cited on pages 25 and 27.)

[DW07] T. De Wolf. *Analysing and Engineering Self-Organising Emergent Applications*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, May 2007. Berbers, Yolande and Holvoet, Tom (supervisors). (Cited on page 172.)

[EG00] G. Engels and L. Groenewegen. Object-Oriented Modeling: A Roadmap. In *Proceedings of the Conference on The Future of Software*

Engineering, ICSE '00, pages 103–116, New York, NY, USA, 2000. ACM. (Cited on page 34.)

[EHHS00] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceedings of the 3rd International Conference on The Unified Modeling Language: Advancing the Standard (UML '00)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000. (Cited on page 119.)

[Eme08] E. A. Emerson. The Beginning of Model Checking: A Personal Perspective. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, pages 27–45. Springer, Berlin/Heidelberg, 2008. (Cited on page 48.)

[ES10] G. Engels and S. Sauer. A Meta-Method for Defining Software Engineering Methods. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-driven Engineering*, pages 411–440. Springer, Berlin/Heidelberg, 2010. (Cited on page 33.)

[ESW07] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of UML Activities Using Dynamic Meta Modeling. In *Proceedings of the 8th Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS '06)*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007. (Cited on pages 53, 86, 119, 121, 123 and 143.)

[FBGL+13] M. Fazal-Baqaie, B. Güldali, M. Luckey, S. Sauer, and M. Spijkerman. Maßgeschneidert und werkzeugunterstützt Entwickeln angepasster Requirements Engineering-Methoden. *OBJEKTspektrum (Online Themenspecials)*, (RE/2013):1–5, June 2013. (Cited on page 15.)

[FBLE13] M. Fazal-Baqaie, M. Luckey, and G. Engels. Assembly-based Method Engineering with Method Patterns. In M. Kuhrmann, D. M. Fernàndez, O. Linsen, and A. Knapp, editors, *Proceedings of the Workshop on Modellierung von Vorgehensmodellen - Paradigmen, Sprachen, Tools (MvV '13)*, pages 435–444. GI, Köllen Druck+Verlag GmbH, Bonn, 2013. (Cited on page 15.)

[FS09] F. Fleurey and A. Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of

Dynamic Adaptive Systems. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, volume 5795 of *Lecture Notes in Computer Science*, pages 606–621. Springer, 2009. (Cited on pages 59, 74, 75, 76, 116, 117 and 118.)

[GAA+13] G. Georg, S. Ali, D. Amyot, B. H. Cheng, B. Combemale, R. France, J. Kienzle, J. Klein, P. Lahire, M. Luckey, et al. Modeling Approach Comparison Criteria for the CMA@RE Workshop at RE 2013, 2013. (Cited on pages 15 and 201.)

[GCGC08] M.-P. Gleizes, V. Camps, J.-P. Georgé, and D. Capera. Engineering Systems Which Generate Emergent Functionalities. In D. Weyns, S. Brueckner, and Y. Demazeau, editors, *Engineering Environment-Mediated Multi-Agent Systems*, volume 5049 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2008. (Cited on page 172.)

[GCH+04] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004. (Cited on pages 8, 59, 78 and 226.)

[GdlIW13] D. Gil de la Iglesia and D. Weyns. Guaranteeing robustness in a mobile learning application using formally verified MAPE loops. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '13)*, pages 83–92, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on pages 116, 117 and 118.)

[Gie07] H. Giese. Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems. In F. Kordon and J. Sztipanovits, editors, *Reliable Systems on Unreliable Networked Platforms*, volume 4322 of *Lecture Notes in Computer Science*, pages 258–280. Springer, 2007. (Cited on pages 74, 76 and 77.)

[GKLE10] C. Gerth, J. Küster, M. Luckey, and G. Engels. Precise Detection of Conflicting Change Operations using Process Model Terms. In N. R. D.C. Petriu and Ø. Haugen, editors, *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, volume 6395 of *Lecture Notes in Computer Science*, pages 93–107, Berlin/Heidelberg, October 2010. Springer. ACM Distinguished Paper Award MODELS 2010.

(Cited on page 15.)

[GKLE11] C. Gerth, J. Küster, M. Luckey, and G. Engels. Detection and Resolution of Conflicting Change Operations in Version Management of Process Models. *Software and Systems Modeling*, pages 1–19, December 2011. (Cited on page 15.)

[GL12] C. Gerth and M. Luckey. Towards Rich Change Management for Business Process Models. In U. Kelter, editor, *Proceedings of the Workshop on Comparison and Versioning of Software Models (CVSM'12)*, volume 32 of *Softwaretechnik-Trends*, pages 32–34. FG Softwaretechnik, Gesellschaft für Informatik e.v. (GI), November 2012. (Cited on page 15.)

[GLE12] S. Geisen, M. Luckey, and G. Engels. Ein Ansatz zur dynamischen Qualitätsmessung, -bewertung und Anpassung von Software Engineering Methoden. In *Proceedings of 19th Workshop on Qualitätsmanagement und Vorgehensmodelle (GI-WIVM '12)*, pages 111–120. Shaker, September 2012. (Cited on page 15.)

[GLKE10] C. Gerth, M. Luckey, J. Küster, and G. Engels. Detection of Semantically Equivalent Fragments for Business Process Model Change Management. In *Proceedings of the 7th International Conference on Services Computing (SCC '10)*, pages 57–64. IEEE Computer Society, 2010. Best Student Paper of SCC 2010. (Cited on page 15.)

[GLKE11] C. Gerth, M. Luckey, J. Küster, and G. Engels. Precise Mappings between Business Process Models in Versioning Scenarios. In *Proceedings of the 8th International Conference on Services Computing (SCC '11)*, pages 218–225. IEEE Computer Society, 2011. (Cited on page 15.)

[GSB+08] H. Goldsby, P. Sawyer, N. Bencomo, B. Cheng, and D. Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'08)*, pages 36–45. IEEE Computer Society, 2008. (Cited on pages 8 and 59.)

[Hau05] J. H. Hausmann. *Dynamic Meta Modeling*. PhD thesis, University of Paderborn, 2005. (Cited on pages 13, 45, 52, 75 and 230.)

[HGB10] R. Hebig, H. Giese, and B. Becker. Making Control Loops Explicit

when Architecting Self-Adaptive Systems. In *Proceeding of the 2nd International Workshop on Self-Organizing Architectures (SOAR '10)*, pages 21–28, New York, NY, USA, 2010. ACM. (Cited on pages 8, 59, 74, 76 and 78.)

[HMK09] J. Hielscher, A. Metzger, and R. Kazhamiakin. Taxonomy of Adaptation Principles and Mechanisms. `http://www.s-cube-network.eu/results/deliverables/wp-jra-1.2/CD-JRA-1.2.2_Taxonomy_of%20Adaptation_Principles_and_Mechanisms.pdf/view`, 2009. (Cited on page 25.)

[Hop71] J. E. Hopcroft. An n log n Algorithm for Minimizing States in a Finite Automation. In *Theory of Machines and Computations*. Academic Press, 1971. (Cited on page 144.)

[JÖ2] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language (UML '02)*, pages 412–425, London, UK, UK, 2002. Springer. (Cited on pages 47 and 60.)

[Jac02] D. Jackson. Alloy: a Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, April 2002. (Cited on page 75.)

[JB06] F. Jouault and J. Bézivin. KM3: a DSL for metamodel specification. In *Proceedings of the 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, pages 171–185, Berlin/Heidelberg, 2006. Springer. (Cited on page 44.)

[KC03] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003. (Cited on pages 6 and 60.)

[KM07] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. In *Proceedings of the Conference on Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE Computer Society, 2007. (Cited on page 60.)

[KNR05] M. Kuhrmann, D. Niebuhr, and A. Rausch. Application of the V-Modell XT – Report from a Pilot Project. In M. Li, B. W. Boehm, and L. J. Osterweil, editors, *ISPW*, volume 3840 of *Lecture Notes in Computer Science*, pages 463–473. Springer, 2005. (Cited on

page 33.)

[Kru03] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003. (Cited on pages 33 and 166.)

[LBFW10] M. Luckey, A. Baumann, D. M. Fern'andez, and S. Wagner. Reusing Security Requirements using an Extended Quality Model. In *Proceedings of the 6th Workshop on Software Engineering for Secure Systems (SESS '10)*, May 2010. (Cited on page 15.)

[LE13] M. Luckey and G. Engels. High-Quality Specification of Self-Adaptive Software Systems. In *Proceeding of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '13)*, New York, NY, USA, May 2013. ACM. (Cited on pages 15, 74 and 116.)

[LEE12] M. Luckey, M. Erwig, and G. Engels. Systematic Evolution of Model-Based Spreadsheet Applications. *Journal of Visual Languages and Computing*, 23(5):267–286, Oct 2012. (Cited on page 15.)

[LGSE11] M. Luckey, C. Gerth, C. Soltenborn, and G. Engels. QUAASY - QUality Assurance of Adaptive SYstems. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC '11)*. ACM, June 2011. (Cited on page 15.)

[LM12] M. Luckey and F. Mutz. Modeling with Adapt Cases. In *Repository for Model-Driven Development (ReMoDD)*. University of Paderborn, 2012. http://www.cs.colostate.edu/remodd/v1/content/modeling-adapt-cases. (Cited on pages 15 and 196.)

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proceedings of the Workshop on Intelligent Signal Processing*, May 2001. (Cited on page 44.)

[LNGE11] M. Luckey, B. Nagel, C. Gerth, and G. Engels. Adapt Cases: Extending Use Cases for Adaptive Systems. In *Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS '11)*, pages 30–39, New York, NY, USA, May 2011. ACM. (Cited on pages 15, 181, 182 and 183.)

[LTGE12] M. Luckey, C. Thanos, C. Gerth, and G. Engels. Multi-Staged Quality Assurance for Self-Adaptive Systems. In *Proceedings of 1st International Workshop on EVALUATION for SELF-ADAPTIVE and SELF-ORGANIZING SYSTEMS (Eval4SASO '12)*, 2012. (Cited on page 15.)

[Luc13] M. Luckey. Webpage: ACML & QUAASY. `http://triviality.de/acml`, July 2013. (Cited on pages 17, 189, 191, 196, 199, 201, 212 and 257.)

[MAA+12] G. Mussbacher, O. Alam, M. Alhaj, S. Ali, N. Amàlio, B. Barn, R. Bræk, T. Clark, B. Combemale, L. M. Cysneiros, U. Fatima, R. France, G. Georg, J. Horkoff, J. Kienzle, J. C. Leite, T. C. Lethbridge, M. Luckey, A. Moreira, F. Mutz, A. P. A. Oliveira, D. C. Petriu, M. Schöttle, L. Troup, and V. M. B. Werneck. Assessing Composition in Modeling Approaches. In *Proceedings of the 2nd Workshop on Comparing Modeling Approaches (CMA '12)*, pages 1:1–1:26, New York, NY, USA, 2012. ACM. (Cited on pages 15 and 201.)

[Moo09] D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009. (Cited on pages 181, 182, 183 and 185.)

[MPP08] M. Morandini, L. Penserini, and A. Perini. Towards Goal-Oriented Development of Self-Adaptive Systems. In *Proceedings of the 3rd International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '08)*, pages 9–16, New York, NY, USA, 2008. ACM. (Cited on page 8.)

[Mur04] R. Murch. *Autonomic Computing*. IBM Press, 2004. (Cited on pages 25 and 76.)

[Mut12] F. Mutz. Modeling Structural and Behavioral Adaptation of Software Systems. Master's thesis, Research Group of Databases and Information Systems, University of Paderborn, Germany, 2012. (Cited on pages 81, 198, 199, 200, 272, 276 and 281.)

[NSS+11] F. Nafz, H. Seebach, J.-P. Steghöfer, G. Anders, and W. Reif. Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach. In C. Mueller-Schloer, H. Schmeck, and T. Ungerer, editors, *Organic Computing – A*

Paradigm Shift for Complex Systems, volume 1 of *Autonomic Systems*, pages 79–93. Springer Basel, 2011. (Cited on pages 116, 117 and 118.)

[Obj00] Object Management Group (OMG). Unified Modeling Language (UML), Infrastructure (Version 1.3). Technical report, Object Management Group, Framingham, MA, 2000. (Cited on page 36.)

[Obj03] Object Management Group (OMG). Model-Driven Architecture (MDA). Technical report, Object Management Group, Framingham, MA, May 2003. (Cited on pages 7, 9, 58, 59, 77 and 230.)

[Obj06] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification Version 2.0. Technical report, Object Management Group, Framingham, MA, 2006. (Cited on pages 44 and 46.)

[Obj08] Object Management Group (OMG). Software & Systems Process Engineering Metamodel Specification (SPEM). Technical report, Object Management Group, Framingham, MA, April 2008. (Cited on pages 16 and 165.)

[Obj10a] Object Management Group (OMG). Systems Modeling Language (SysML), Version 1.2. Technical report, Object Management Group, Framingham, MA, June 2010. (Cited on page 60.)

[Obj10b] Object Management Group (OMG). Unified Modeling Language (UML), Superstructure (Version 2.3). Technical report, Object Management Group, Framingham, MA, 2010. (Cited on pages 12, 74, 79 and 123.)

[Obj11] Object Management Group (OMG). Unified Modeling Language (UML), Infrastructure (Version 2.4.1). Technical report, Object Management Group, Framingham, MA, 2011. (Cited on pages 36, 42, 47 and 223.)

[Obj12] Object Management Group (OMG). Service Oriented Architecture Modeling Language (SoaML), Version 1.0.1. Technical report, Object Management Group, Framingham, MA, May 2012. (Cited on page 47.)

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society. (Cited

on page 25.)

[Pap13] Papyrus. http://www.papyrusuml.org, June 2013. (Cited on pages 215 and 227.)

[Par94] D. L. Parnas. Software Aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. (Cited on page 5.)

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2003. (Cited on pages 51, 53, 119 and 128.)

[Ren04] A. Rensink. State Space Abstraction Using Shape Graphs. In *Proceedings of the 3rd International Workshop on Automatic Verification of Infinite-State Systems (AVIS '04)*, pages 226–241. Elsevier, 2004. (Cited on page 156.)

[Res12] Research Group MePaso. Modeling and Execution of Process-driven Adaptive Service Orchestrations. http://is.uni-paderborn.de/?id=14596, 2012. (Cited on pages 164 and 189.)

[RG02] M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 42–68. Springer, 2002. (Cited on page 75.)

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997. (Cited on page 51.)

[RRM+12] J. Rybka, E. Rybka, F. Mutz, T. M. Babu, N. N. Pathak, T. Miklossy, and S. Jojiju. Description of the Car Window Insurance Case Study. Deliverable of the Project Group MEPASO at the University of Paderborn, 2012. (Cited on pages 190, 191, 192 and 232.)

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959. (Cited on page 144.)

[Rup07] C. Rupp. *Requirements-Engineering und -Management: Professionelle, Iterative Anforderungsanalyse für die Praxis*. Hanser, 2007. (Cited on page 40.)

[Sch97] K. Schwaber. SCRUM Development Process. In J. Sutherland, C. Casanave, J. Miller, P. Patel, and G. Hollowell, editors, *Business Object Design and Implementation*, pages 117–134. Springer London, 1997. (Cited on page 33.)

[SE09] C. Soltenborn and G. Engels. Towards Test-Driven Semantics Specification. In B. S. A. Schürr, editor, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, volume 5795 of *Lecture Notes in Computer Science*, pages 378–392, Berlin/Heidelberg, 2009. Springer. (Cited on page 53.)

[SGP12] G. Salvaneschi, C. Ghezzi, and M. Pradella. ContextErlang: Introducing Context-Oriented Programming in the Actor Model. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development (AOSD '12)*, pages 191–202, New York, NY, USA, 2012. ACM. (Cited on page 59.)

[Sha80] M. Shaw. The Impact of Abstraction Concerns on Modern Programming Languages. *Proceedings of the IEEE*, 68(9):1119 – 1130, sept. 1980. (Cited on page 36.)

[SNSR10] H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif. A Software Engineering Guideline for Self-Organizing Resource-Flow Systems. In *Proceedings of the 4th International Conference on Self-Adaptive and Self-Organizing Systems (SASO '10)*, pages 194–203, 2010. (Cited on page 173.)

[Sol13] C. Soltenborn. *Quality Assurance with Dynamic Meta Modeling*. PhD thesis, University of Paderborn, 2013. (Cited on pages 52 and 133.)

[SSLRM11] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness Requirements for Adaptive Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '11)*, pages 60–69, New York, NY, USA, 2011. ACM. (Cited on page 59.)

[ST09] M. Salehie and L. Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and*

Adaptive Systems, 4(2):1–42, 2009. (Cited on pages 6, 25 and 27.)

[Tha12] C. Thanos. Efficient Quality Assurance for Self-Adaptive Software Systems. Master's thesis, Research Group of Databases and Information Systems, University of Paderborn, Germany, 2012. (Cited on page 139.)

[VG12] T. Vogel and H. Giese. A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '12)*, pages 129–138. IEEE Computer Society, 6 2012. (Cited on pages 74, 75 and 76.)

[vL09] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, March 2009. (Cited on page 162.)

[WIMA12] D. Weyns, M. Iftikhar, S. Malek, and J. Andersson. Claims and Supporting Evidence for Self-Adaptive Systems: A Literature Study. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '12)*, pages 89 –98, June 2012. (Cited on pages 5 and 6.)

[WIS13] D. Weyns, M. U. Iftikhar, and J. Söderlund. Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '13)*, pages 3–12, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on pages 187, 194 and 224.)

[WMA10] D. Weyns, S. Malek, and J. Andersson. FORMS: a FOrmal Reference Model for Self-adaptation. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC '10)*, pages 205–214, New York, NY, USA, 2010. ACM. (Cited on page 25.)

[WRH+00] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. (Cited on pages 179, 203, 204, 205, 207 and 232.)

[WSB+09] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Proceedings of the 17th International Requirements Engineering Conference (RE '09)*, pages 79–88. IEEE

Computer Society, 2009. (Cited on pages 59, 162, 163 and 168.)

[Yin91] R. K. Yin. Advancing Rigorous Methodologies: A Review of "Towards Rigor in Reviews of Multivocal Literatures... ". *Review of Educational Research*, 61(3):pp. 299–305, 1991. (Cited on page 180.)

[ZC06] J. Zhang and B. H. C. Cheng. Model-Based Development of Dynamically Adaptive Software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 371–380, New York, NY, USA, 2006. ACM. (Cited on pages 74, 75, 76, 116, 117 and 118.)

[ZR11] E. Zambon and A. Rensink. Using Graph Transformations and Graph Abstractions for Software Verification. *Electronic Communications of the EASST*, 38, August 2011. (Cited on page 156.)

(Total references: 122)

Appendices

A

Meta Model Definitions of the ACML

The following sections give a detailed description of the meta model that describes the ACML. Even more detailed information can be obtained from the ACML specifications on the website at [Luc13].

A.1 STRUCTURAL MODELING WITH ACML

The structural modeling part of the ACML allows to define an adaptation-specific view on the system. It therefore allows the decoration of parts of the system model using specialized components and attached interfaces. This decorating model is coined the *Adaptation View Model* (AVM).

Figures A.1 to A.26 describe the core concepts of the AVM. The figures show a meta model which extends classes from the UML meta model. The italic written names in the upper right of a class depict the super classes that are not shown with specialization arrows in the same diagram. Other than that all following meta models are given in standard class diagram notation as known from the UML specification.

All elements that are used from the UML meta model are specialized (subtyped) with a name starting with *ACML*, e.g., ACMLComponent specializes the UML Component. All *ACMLElements* inherit from *VersionedElement* which allows the creation of histories as described in Section A.1.3.

A.1.1 CORE ELEMENTS

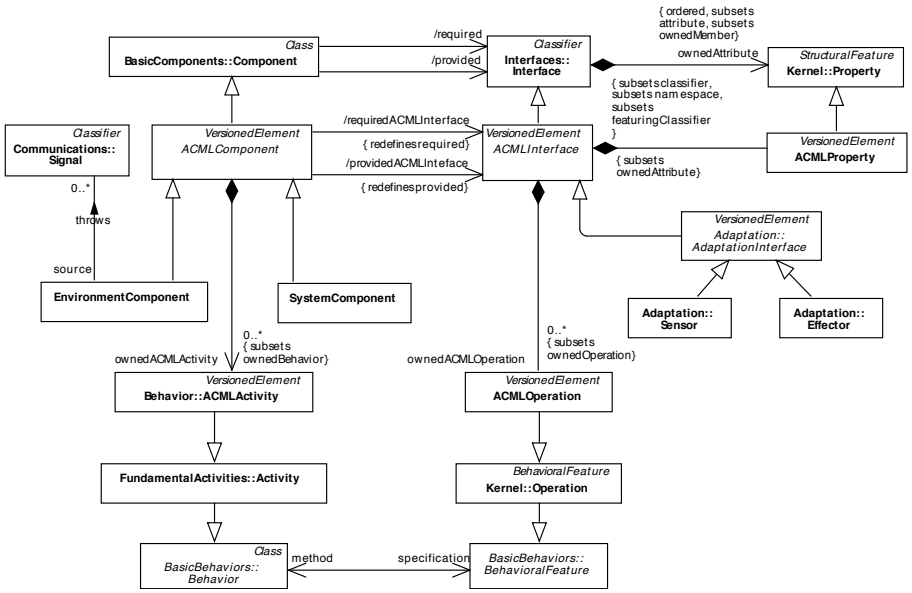


FIGURE A.1.
Meta Model: Basic Adaptation View

ACMLComponents and ACMLInterfaces. The meta model shown in Fig-

ure A.1 depicts the core elements of an Adaptation View Model, the ACMLComponents. ACMLComponents define their behavior in terms of required and provided ACMLInterfaces which in turn may have ACMLOperations and ACMLProperties. An ACMLComponent is an abstract class which is specialized to EnvironmentComponent and SystemComponent. A SystemComponent describes any logical entity of the system. As known from the UML, a component is a modular unit with well-defined interfaces that is replaceable within its environment but does not provide a concrete implementation. A system may either be described with the use of several SystemComponents that are connected to each other via interfaces, or the system may be described by a single SystemComponent (e.g., named "*system*") that only exposes several interfaces that are needed for adaptation. An EnvironmentComponent may additionally throw signals that in turn trigger the system's adaptation (cf. Section A.2).

Since ACMLComponents subtype UML classes, they may have attributes and operations, and may participate in associations and generalizations. Further, as known from UML components, the ACMLComponents' behavior may be realized by a set of realizing classifiers.

An important aspect of components is their ability to be reused in different system designs. A component is an autonomous unit within a system and is only accessible via its provided interfaces, while all other internals are hidden. ACMLComponents expose required or provided ACMLInterfaces. Provided interfaces may either be realized directly by the components or by one of its realizing classifiers. ACMLInterface is an abstract class which is specialized to two non-abstract adaptation interfaces: Sensor and Effector. Both Sensor and Effector may expose ACMLProperties and ACMLOperations. The main purpose of Sensors is to gain information about the system and its environment. They do not change (i.e. adapt) the system. Contrarily, the Effectors' main purpose is to change (i.e. adapt) the system. They are not used to gain information about the system and its environment.

ACMLComponents may have attached behaviors for interfaces or the components themselves. This behavior describes the component's external behavior more precisely. Behaviors such as protocol state machines usually make the sequence of operation calls at interfaces explicit while behaviors such as activities usually describe the orchestration of the component's composites to make its internal behavior explicit. For self-adaptive systems, this behavioral component specification enables the adaptation of the system's behavior by changing these very activities.

ACMLProperties specialize the UML Property. They are described in detail below. ACMLOperations specialize the UML Operation which is a UML BehavioralFeature. The concrete behavior of BehavioralFeatures is defined using UML Behaviors, which may, e.g. be UML Activities. Thus, ACMLActivities are used to define the concrete behavior of ACMLOperations.

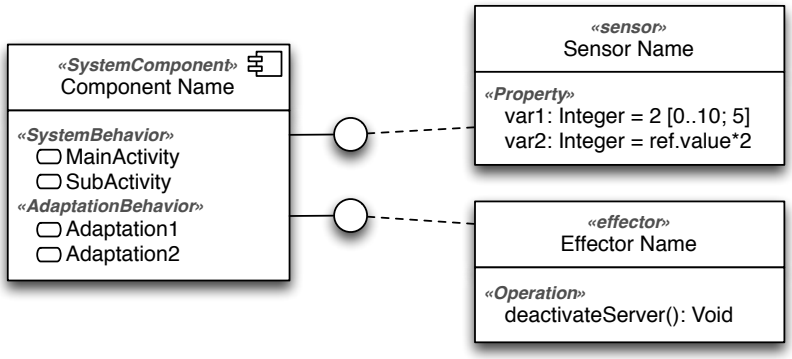


FIGURE A.2. SystemComponent with AdaptationInterfaces (Sensors, Effectors) and Properties

The concrete syntax is aligned with the UML as shown in Figure A.2. The system component exposes two provided adaptation interfaces, a sensor and an effector. Further, the component has two defined system behaviors and two adaptation behaviors, the latter of which may define adaptations of the former.

ACMLAssociations. Figure A.3 shows the definition of ACMLAssociations. They are contained in ACMLComponents and may associate several subcomponents to each other. They specialize standard UML Associations but also inherit from VersionedElement.

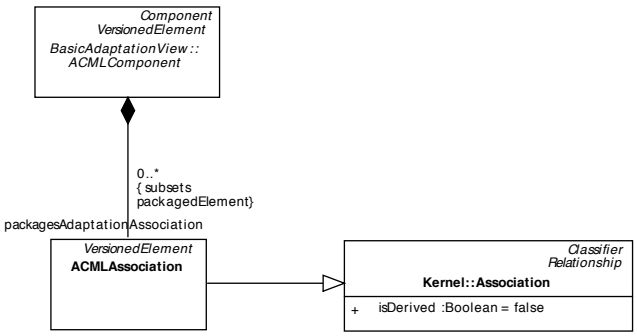


FIGURE A.3. Meta Model: ACML Associations

Associations are denoted as known from the UML with simple lines and optional name, roles, and cardinalities.

ACMLActivities. As shown in Figure A.4 the abstract class ACMLActivity is UML Behavior and refined to AdaptationBehavior and SystemBehavior. SystemBehavior is used to define the concrete behavior of ACMLOperations and

ACMLComponents. AdaptationBehavior is used to describe the behavior that actually modifies the system's structure or behavior. AdaptationBehavior can be exposed using effector interfaces. ACMLActivities contain nodes and edges just like standard UML Activities. However to be historizable, edges are refined to ACMLActivityEdge which may be an ACMLObjectFlow or an ACMLControlFlow. The UML semantics remain unchanged. AdaptationRegion specializes UML StructuredActivityNode and is used to group those elements that are meant to be adaptable. This modeling element is especially important if modeling with multiple users since it allows the adaptation modeler to emphasize the adaptive part of the system.

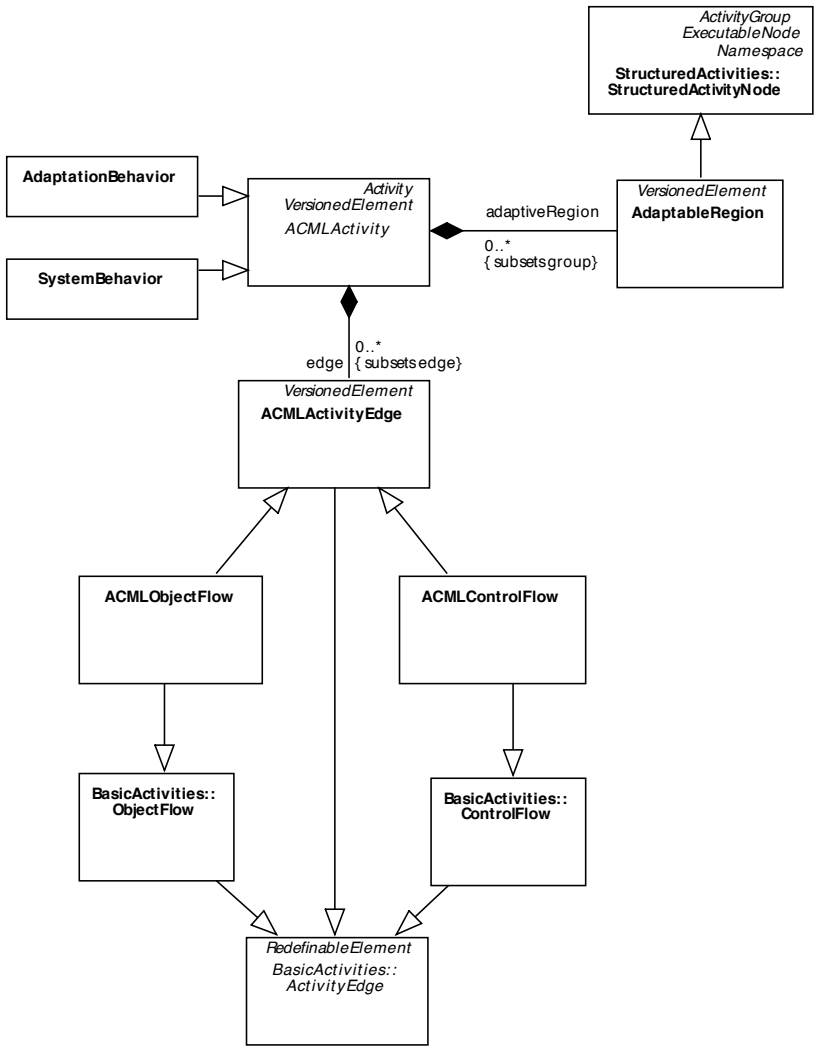


Figure A.4.
Meta Model: ACML Behavior

The concrete syntax of ACMLActivities is aligned to UML activities. Figure A.5 shows SystemBehavior and on the left and AdaptationBehavior on the right. Activities may be represented in diagram shape as shown at the top, and in

activity shape as shown at the bottom of the figure. In addition to a name, each ACMLActivity is equipped with a tag *sys* or *adapt* that allows to distinguish both activity types.

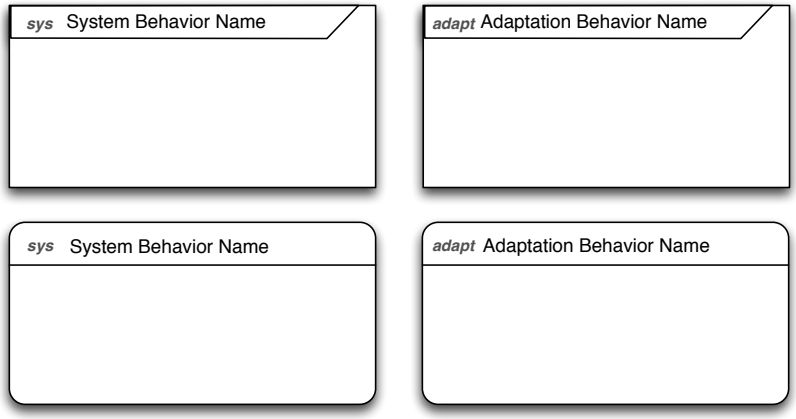


FIGURE A.5.
Concrete Syntax: ACML Behavior

ACMLProperties. The ACML distinguishes two different kinds of properties: interval properties and state properties. ACMLIntervalProperties are shown in Figure A.6. They define some characteristic values being a minValue, a maxValue, a stepSize, and a defaultValue. These values depict the modeler's estimation of the property's range. The values are vitally important for quality assurance, since the system and the environment are simulated according to these values, that is, the property's value is simulated in the range from `minValue` to `maxValue` with a step size of `stepSize` with an initial value of `defaultValue`. Instead of these characteristic values, an ACMLIntervalProperty may be defined using an IntervalPropertySpecification, too. An IntervalPropertySpecification is a UML Expression that may use the value of other ACMLProperties to compute a particular value.

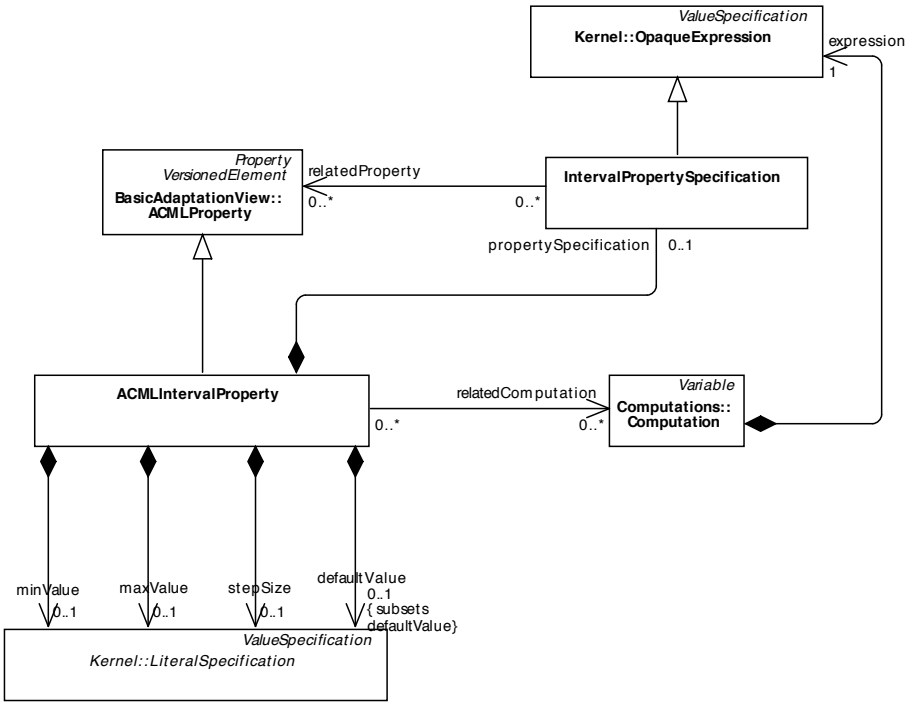


Figure A.6. Meta Model: ACML Interval Properties

The concrete syntax of IntervalProperties is shown in Figure A.7. A property has a name following by a colon and the property's type. Following an equal sign, the default value is given. The brackets include the range and the step size.

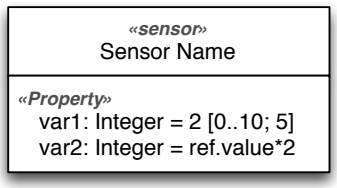


Figure A.7. Concrete Syntax: ACML Interval Property

The second kind of ACMLProperties is the ACMLStateProperty as shown in Figure A.8. The ACMLStateProperty defines a set of state literals the property may adopt. State literals (StatePropertyLiteral) may correspond to specific UML states that are contained within a respective UML StateMachine. Thus, the modeler may either define an ACMLStateProperty to be an unordered set of state literals or the modeler may define the state transitions using a UML StateMachine.

FIGURE A.8.
Meta Model: ACML State Properties

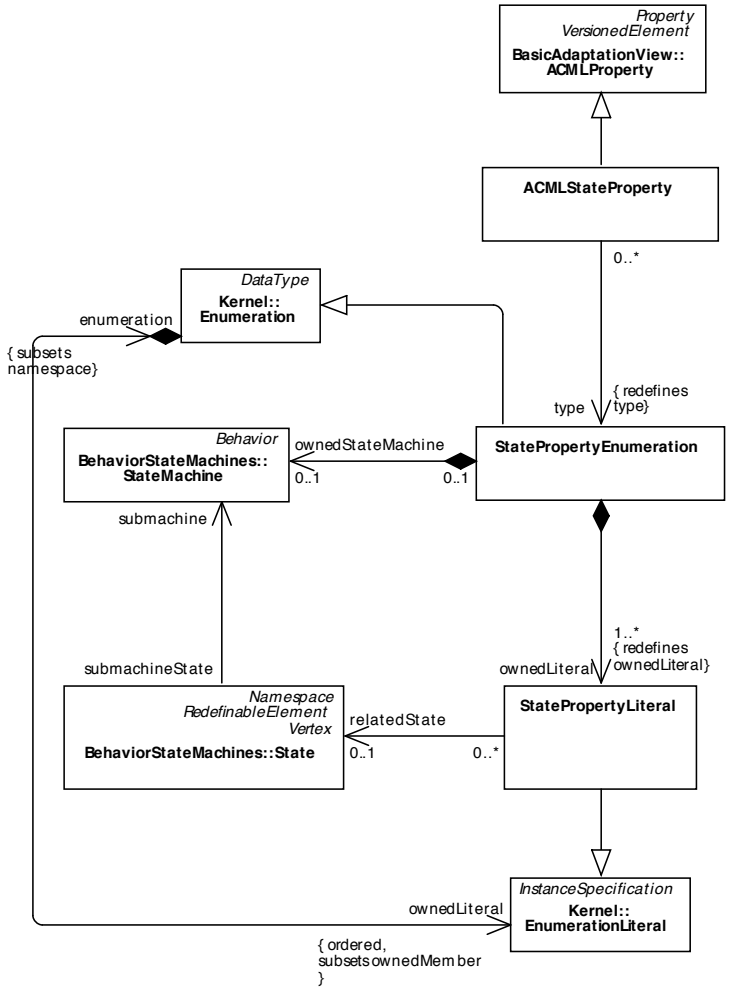


Figure A.9 shows the concrete syntax of ACMLStateProperties. The property has a name followed by a colon and a set of state names. Following an equal sign, the initial state is defined. Optionally, the name of a state machine may be given in brackets.

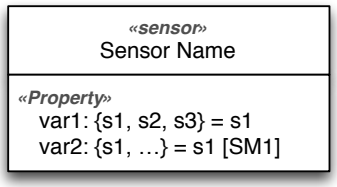


Figure A.9.
Concrete Syntax:
ACML State Property

ACMLPropertyHistory. It is possible to preserve all values of ACMLProperties. Therefore, as shown in Figure A.10, a History is assigned to an ACMLProperty that contains a set of timed HistoryElements. These HistoryElements may either be UML InstanceValues for ACMLIntervalProperties or StatePropertyLiterals for ACMLStateProperties.

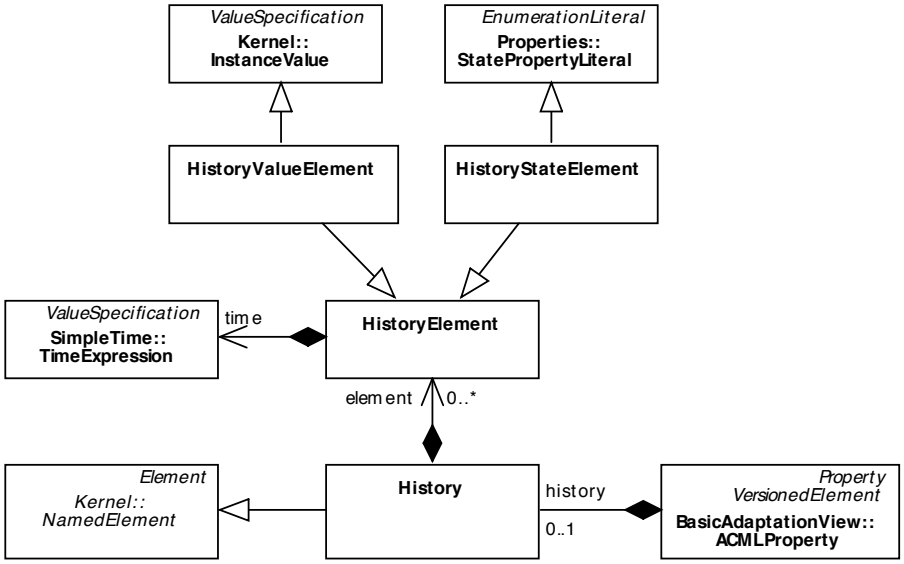


Figure A.10.
Meta Model: Property
Histories

In concrete syntax, a property history is defined by using a small clock icon next to the property's definition as shown in Figure A.11.

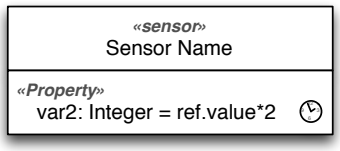


Figure A.11.
Concrete Syntax:
ACML Property
History

A.1.2 INSTANCE SPECIFICATIONS

For several reasons it can be useful to describe instance specifications of a Adaptation View Model. An instance specification allows the modeling of concrete instances (e.g. of components or properties) and the specification of concrete property values. In the UML, instance specifications are primarily used for class diagrams and modeled with so-called object diagrams. Objects are UML InstanceSpecifications for UML Classes. An InstanceSpecification refers to a particular UML Classifier and defines Slots for each Property the Classifier defines. A Slot defines a concrete value for a Property. In the ACML, InstanceSpecifications are used broader, i.e. they are not only used for structural diagram elements such as classes and components, but also for behavioral diagram elements such as activities.

Figure A.12 shows the concept of InstanceSpecifications used for structural elements of the ACML. Since this concept is already extensively used by the UML, the ACML basically specializes the corresponding UML elements. The ACML-ClassifierInstanceSpecification redefines the classifier association to point to VersionedElements. That is, in the ACML only VersionedElements may be instantiated.

FIGURE A.12.
Meta Model: ACML Instances

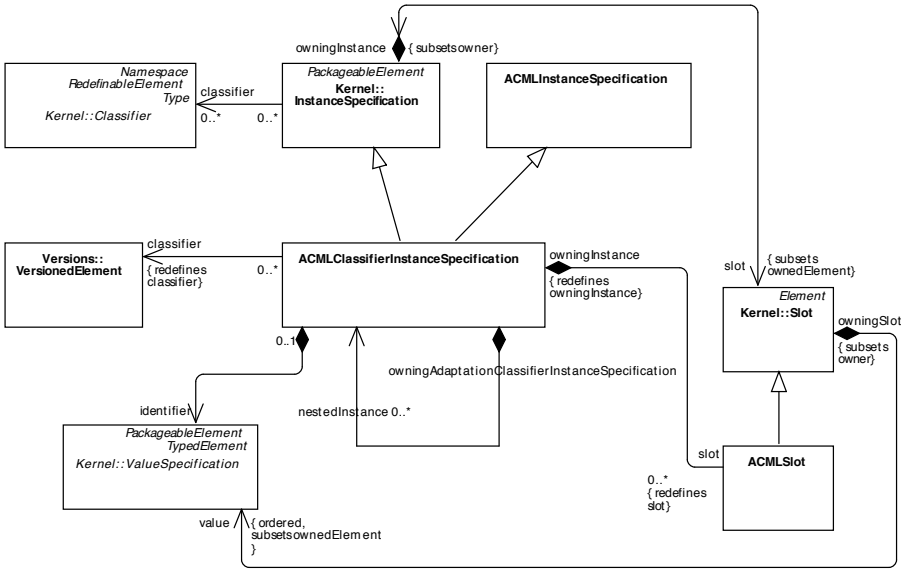


Figure A.13 shows the use of InstanceSpecifications for behavioral diagrams within the ACML. An AdaptationActivityInstanceSpecification refers to an ACMLActivity instead of a UML Classifier and contains a TimeExpression that defines the Activity's startTime. Further, an AdaptationActivityInstanceSpecification contains an ordered set of ExecutionHistoryEntries. These entries con-

tain the value for every parameter, variable, and object nodes, the Activity holds. Finally, the ExecutionHistoryEntry contains a set of TokenNodes that refer to the active ActivityNodes. A TokenNode defines the number of tokens that reside on the particular ActivityNode, as well as a derived variable indicating whether the particular ActivityNode as executing. The latter value depends on the activity semantics that has been defined by the UML.

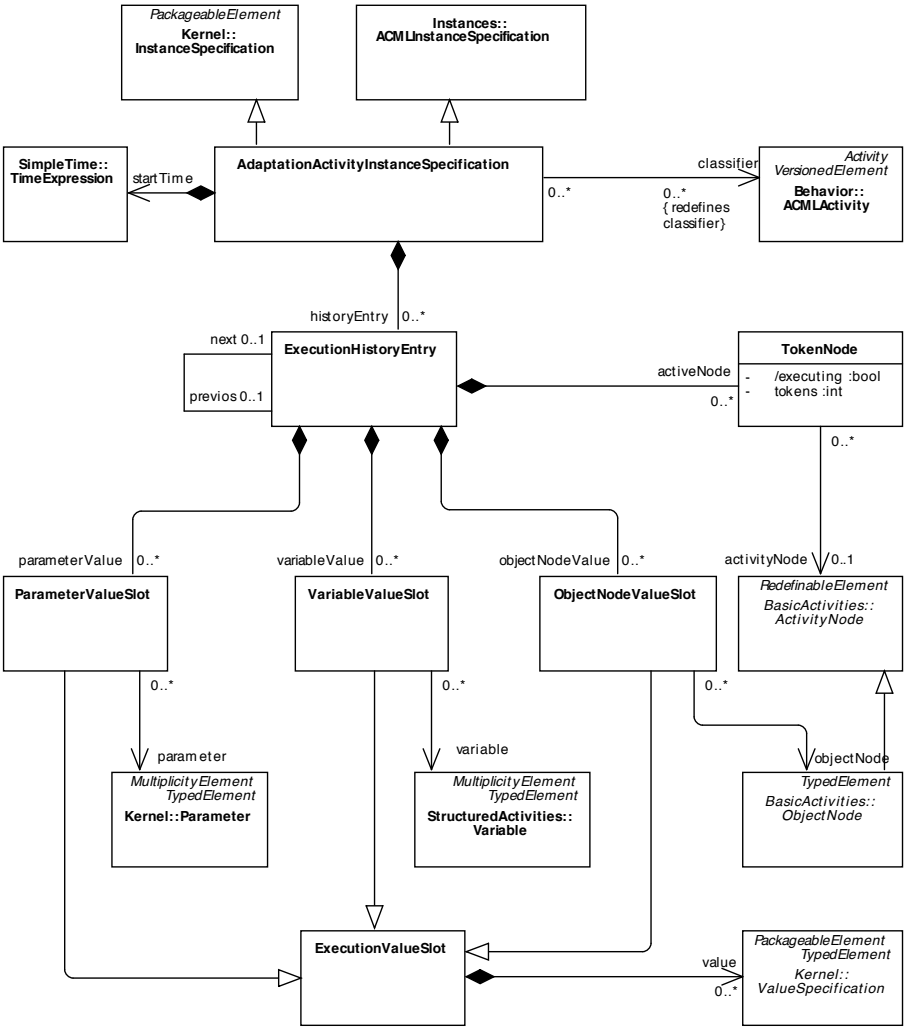


Figure A.13.
Meta Model: ACML Behavior Instances

The concrete syntax of ACMLInstanceSpecifications uses the notation of UML object diagrams.

A.1.3 KNOWLEDGE

Knowledge is specific information that is used during adaptation of the system. It contains history logs[1] of system states, computations/aggregations and policies, and constraints all of which are described in detail below. To organize the knowledge model, the concept of *namespaces* has been used. The UML implements namespaces using packages which may contain packageable elements. These elements can be accessed globally by concatenating the package's name, "::", and the packaged element's name. To leverage this concept in the ACML, knowledge is grouped using a hierarchy of packages, as shown in Figure A.14. It reflects the three main kinds of knowledge: constraints, computations & policies, and histories.

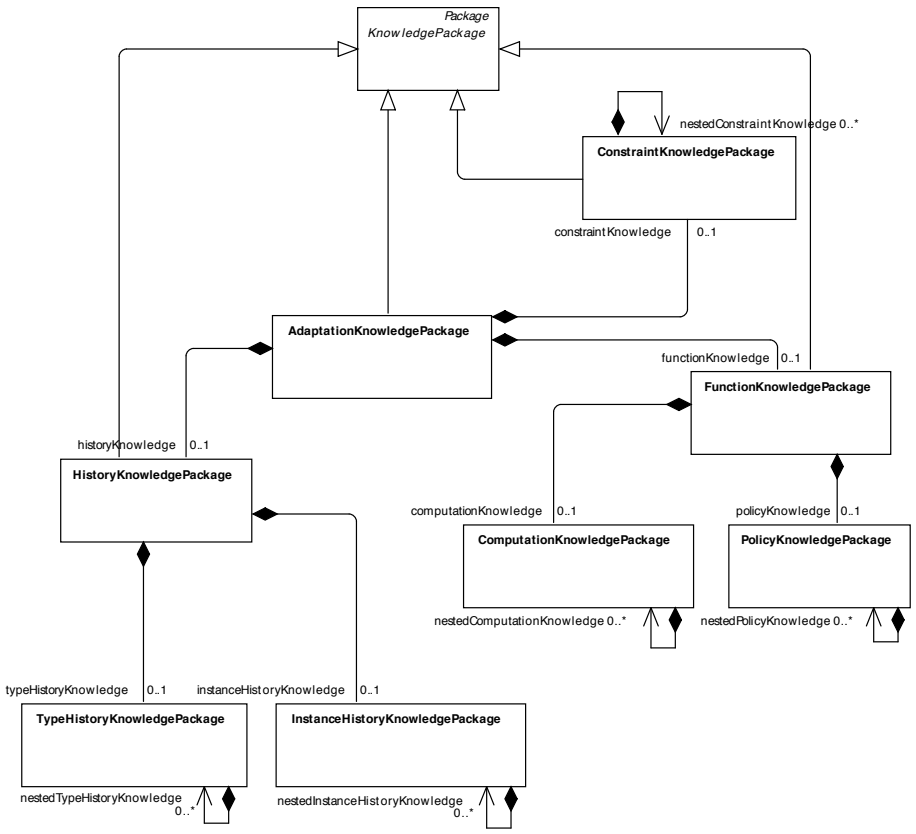


FIGURE A.14.
Meta Model:
Knowledge Packages

Every package serving as container for adaptation knowledge shown in the figure inherits from KnowledgePackage, a UML package. The inheritance edges from the leave packages are omitted in the figure for reasons of readability. The root knowledge package is the AdaptationKnowledgePackage. All kinds

[1] Not to confuse with adaptation histories which log the changes that have been caused by adaptation. System histories log the changes that have been caused by normal system progress.

of knowledge can be grouped into the root package using a tree-like structure. The cardinalities ensure that the first two levels of the knowledge tree are fixed. Semantic constraints assure that the different knowledge packages contain only packages of the respective kind.

The concrete syntax is taken from UML packages. Packages may be decorated with their type in guillemots. The package name should reflect their kind. Figure A.15 shows the UML package notation on the right side and a tree-like notation on the left.

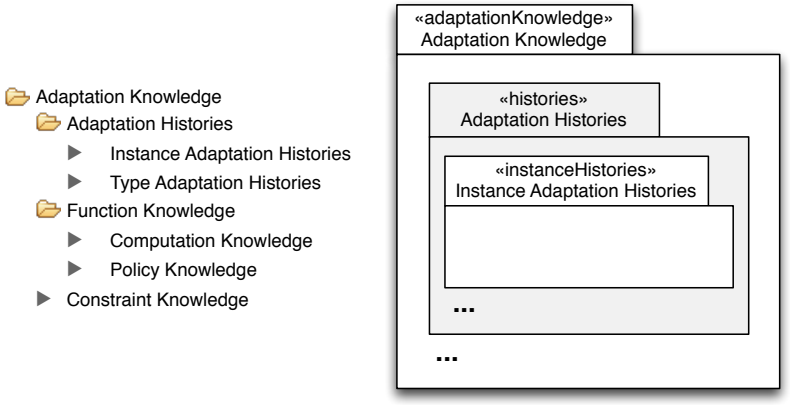


Figure A.15.
Concrete Syntax:
Knowledge Packages

ACML Instance and Type Adaptation History. An adaptation history is used to log the changes, which have been performed by some adaptation action. Both, type and instance changes can be logged within a history. Therefore, a type or instance history definition is created that refers to a particular type. If that type is changed, a new entry is set into its type history. The type itself does not know its history. Instead, the history refers unidirectional to a particular type. Instance histories log every change of an instance, e.g. if instance values are changed.

Whenever an adaptation is performed, the history of the adapted elements are expanded by the old version. Thereby, adaptation can be reverted at all time (backwards adaptation), or respectively, adaptation transactions can be specified.

Figure A.16 shows the InstanceHistorySet that is contained in the InstanceHistoryKnowledgePackage. At design-time, an InstanceHistorySet is specified for each VersionedElement that shall be logged. For each instance of the VersionedElement, the InstanceHistorySet contains an InstanceHistory that points at the current ACMLInstanceSpecification, and further, contains a set of InstanceHistoryEntries each of which points at an *old* ACMLInstanceSpecification and at a *containerId*, the container's id, the instance resides in. Whenever the container of an instance is deleted, the corresponding InstanceHistory's attribute *containerDeleted* is set to *true*. This indicates that an adaptation of this instance cannot be reverted since its container is missing. The *containerId* allows to find the old instance's original container. The InstanceHistoryEntry further has an attribute *instanceDeleted* that is set to *true* if an instance has been deleted. If a deletion is reverted, another InstanceHistoryEntry is added with the attribute *instanceDeleted* set to *false*. Thus, deleted entries might end up between two non-deleted entries.

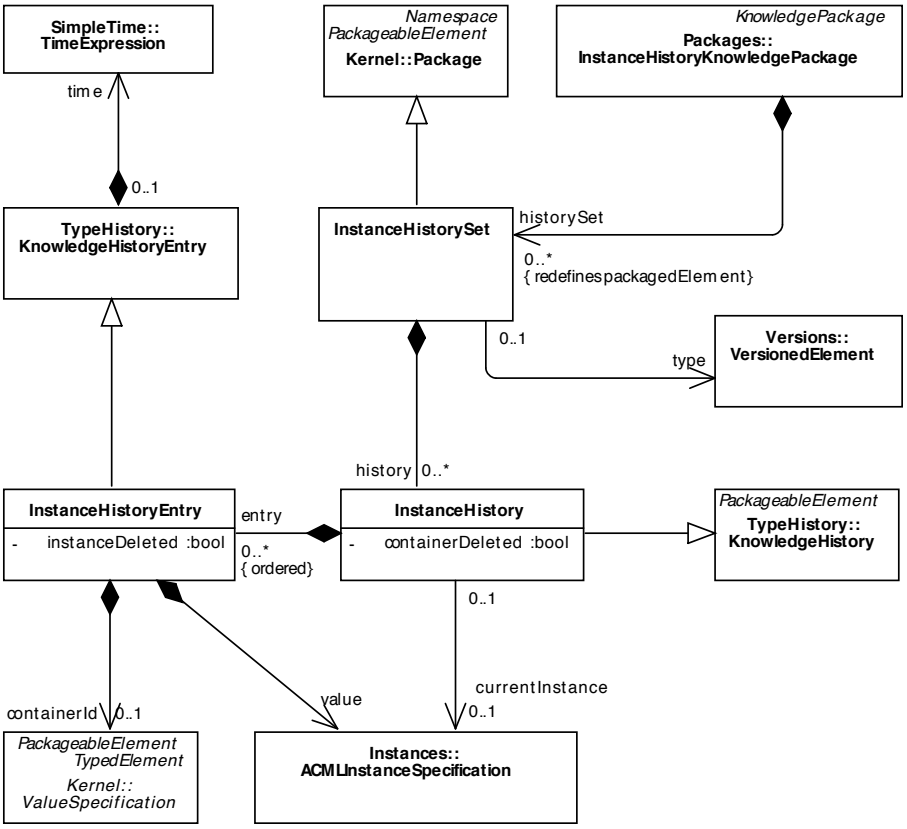


Figure A.16.
Meta Model: Instance History

Figure A.17 shows the meta model excerpt for type histories. A TypeHistoryKnowledgePackage contains a set of TypeHistories. A TypeHistory refers to a specific VersionedElement that shall be tracked over time. Before a VersionedElement's type is changed, the VersionedElement is copied and a new TypeHistoryEntry is created pointing at this old version. Each TypeHistoryEntry is a KnowledgeHistoryEntry and thus contains a TimeExpression that specifies its creation time.

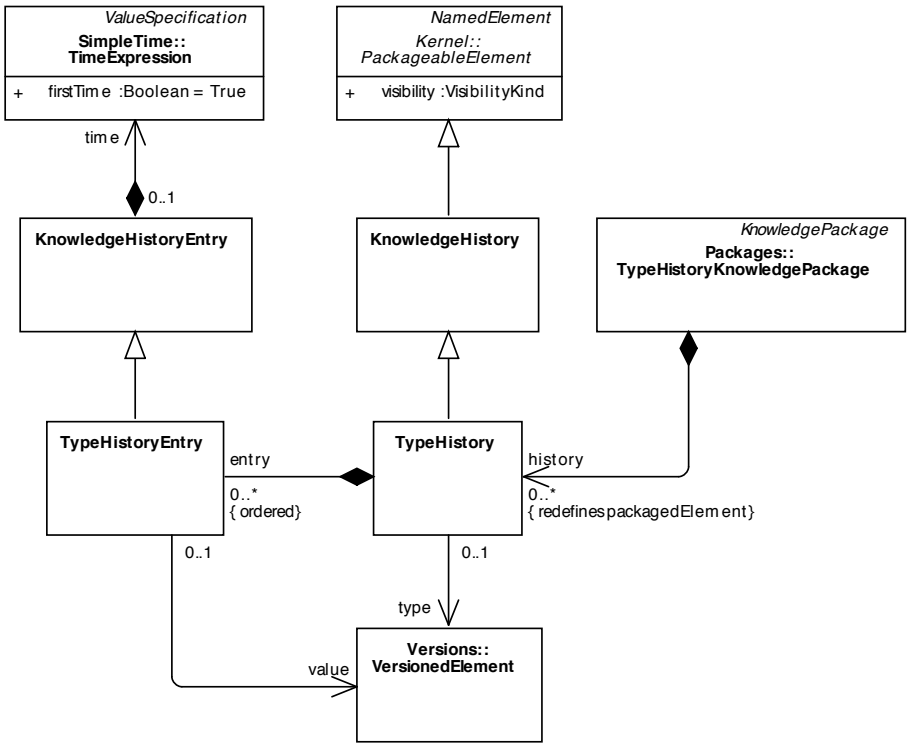


FIGURE A.17. Meta Model: Type History

In concrete syntax, type and instance history definitions are shown in the HistoryKnowledgePackages. They are listed with their name and the VersionedElement's name and are decorated by an icon indicating whether a type or instance history shall be created. Figure A.18 shows the concrete syntax.

The concrete semantics of type and instance histories are detailed in [Mut12]. There, special attention has been paid to deleting used elements, creating deep copies on update, etc.

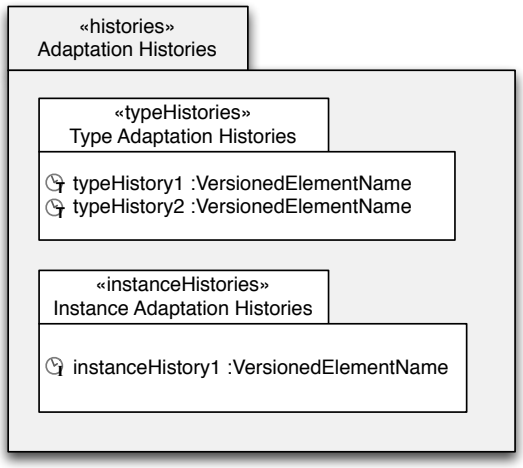


Figure A.18.
Concrete Syntax of Adaptation Histories

ACMLPolicies. Policies are simply ACMLActivities that have been extracted from concrete Operations definitions or Adapt Case definitions to allow for reuse. As such, they may describe business rules, high-level adaptation operations, complex computation behavior, or selectors for instances and types. They can be used from within the whole Adaptation View Model as well as from within the Adapt Case Model (see Section A.2). Policies are described in the meta-model excerpt shown in Figure A.19.

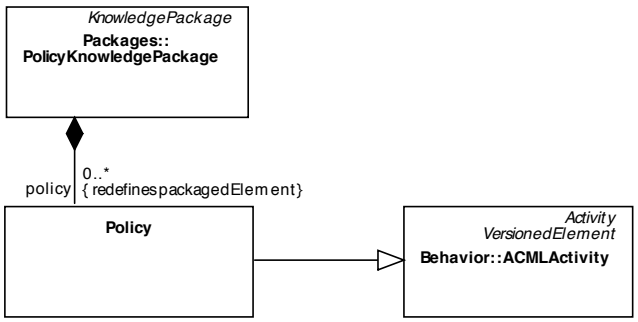


Figure A.19.
Meta Model: Policies Knowledge

The concrete syntax of policies is taken from UML activities with an extra tag *policy* left from the policy's name as shown in Figure A.20. Every policy is listed in the PolicyKnowledgePackage as shown on the figure's right side.

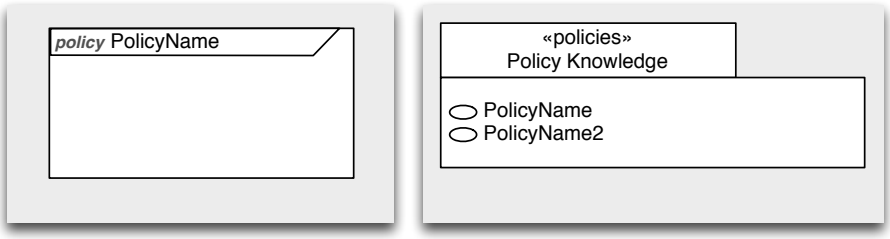


Figure A.20.
Concrete Syntax of Policy Knowledge

ACMLConstraints. The ConstraintKnowledgePackage contains a set of UML Constraints, i.e. invariants that have to hold throughout the complete system lifecycle. Among others, these constraints are checked during quality assurance of the models.

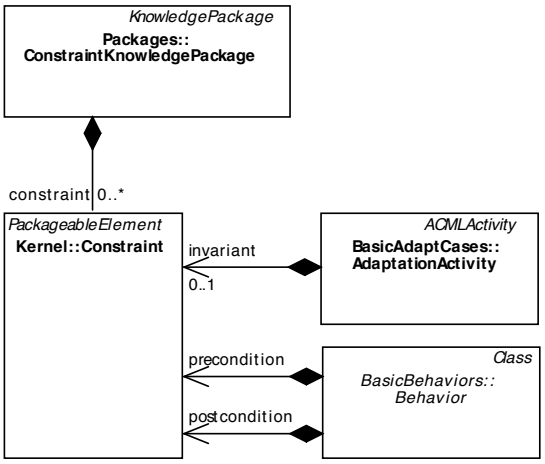


FIGURE A.21. Meta Model: Constraints Knowledge

Constraints are expressions used to determine the validity of models in the presence if adaptation. Constraints may be pre- and postconditions or invariants. These constraints can be specified for Adapt Cases, and the Adaptation Activity. In invariant that is defined for an Adaptation Activity states that the corresponding condition may never be violated during adaptation. Besides invariants, pre-, and postconditions, constraints can also be included in the ConstraintKnowledgePackage. These constraints behave like invariants as well and may never be violated. Constraints are checked for validity during quality assurance of the self-adaptive system model.

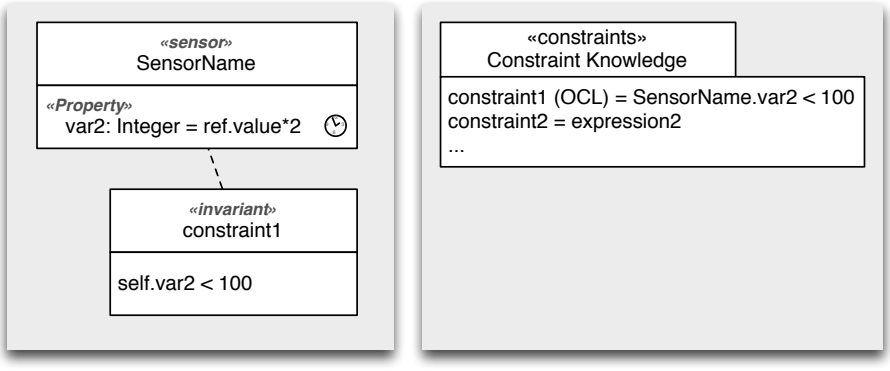


FIGURE A.22. Sensor with attached Invariant Constraint

The concrete syntax of constraints is illustrated in Figure A.22. As shown on the figure's left side, constraints can be visualized within the Adaptation View Model. In that case, the constraint's context may be replaced by `self`. All constraints are collected within the constraint knowledge package that can be vi-

sualized as depicted on the right side of Figure A.22. Here, the constraint's language may be given in brackets.

ACMLComputations. Finally, Figure A.23 shows the meta model excerpt that describes the concept of Computations. A Computation is a Variable and thus can be accessed easily within its scope. Essentially, the Computation is a UML OpaqueExpression that uses referenced Classifiers to compute a specific value that is of interest for adaptation. For instance, Computations are used to create aggregations of several Sensor values. Just like ACML Properties, a Computation can be assigned a History such whenever the Computation's value changes, it is stored within a HistoryElement (see Figure A.10). Computations are always contained in a ComputationKnowledgePackage. The evaluation of computation is lazy, i.e. whenever the computation is accessed, the value is recomputed.

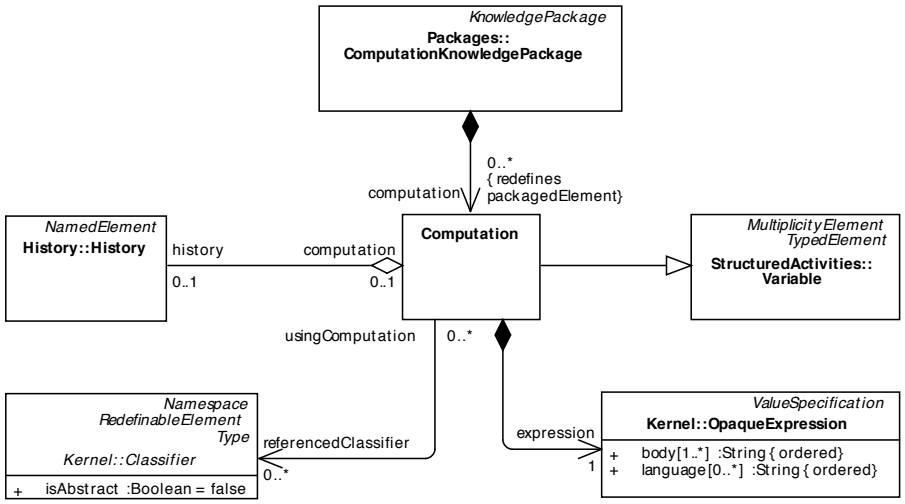


Figure A.23. Meta Model: Computation Knowledge

The concrete syntax of computations is depicted in Figure A.24. The figure's left side shows a computation within the Adaptation View Model. The computation references two UML classifiers, i.e. sensors and computes a new value that is named *computation1* and has type *Int*. The figure's right side shows the computation contained in the computation knowledge package in UML package notation. The second computation defines a history log, that is, whenever the computation's value changes, the old value is preserved and can be used, e.g., in monitoring definitions to initiate pro-active adaptation.

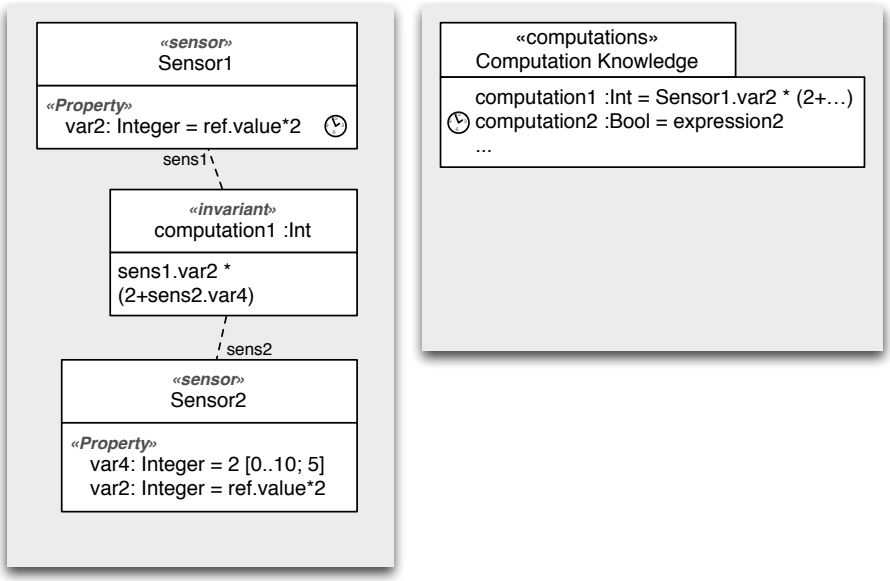


FIGURE A.24. Computation referencing two Classifiers (Sensors)

A.1.4 VERSIONS & VARIABLES

To be able to create an adaptation history, every historized element must be versioned. A VersionedElement has a version which is given by the current timestamp. Using timestamps, different versions are easily sortable (e.g., to create an overall adaptation history) and it is easy to define the most recent version over the complete model simply being the current timestamp. Figure A.25 shows all elements that inherit from VersionedElement. Further, to be able to determine the newest version of an element on type-level, and have current and old versions separated from each other, a concept of legacy containment references has been introduced. That is, each *owned...* reference is accompanied by a *legacy...* reference that contains all elements that once were owned. Details on reasons and implementation are given in [Mut12].

VariablePins as shown in Figure A.26 are a simplification of InputPins and OutputPins which are defined by the UML for use in activity diagrams with object flow. The problem with the original object flow is that several object flow edges have to be included in more complex activity diagrams which makes them hard to read and understand. The idea of VariableInputPins and VariableOutputPins is that they transform the original object flow into variables that can be accessed easily within the activity. Therefore, VariablePins inherit from Variable while a Variable can either be a concrete instance value of a type (TypeVariable). Both Variables and TypeVariables can be used as parameters for activities.

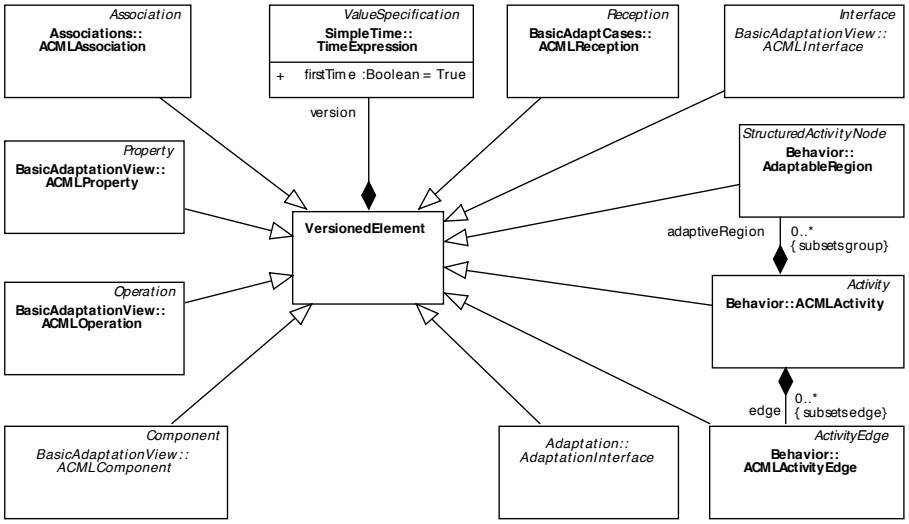


Figure A.25.
Meta Model: Versioned Elements

In concrete syntax, variable pins are represented as normal pins with a v inside the pin (i.e., v).

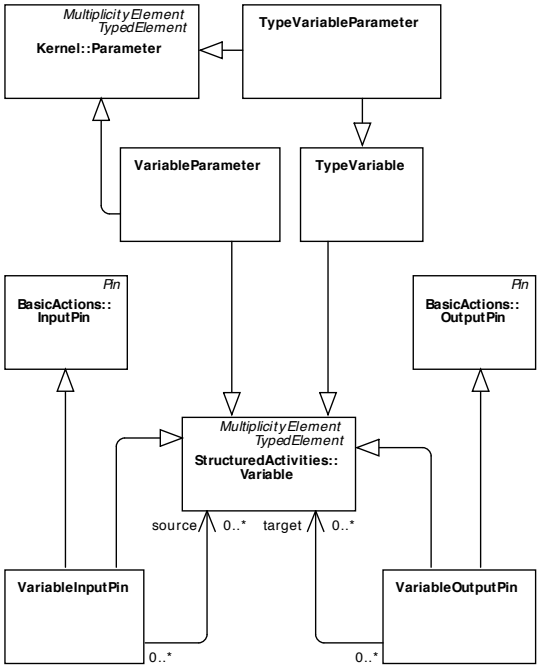


Figure A.26.
Meta Model: Variable Pins

A.2 BEHAVIORAL MODELING WITH ACML

In the last section, the Adaptation View Model (AVM) has been described. The AVM allows the specification of the system's adaptation-relevant structure, e.g., relevant system and environment components and their exposed adaptation interfaces, i.e. sensors and effectors. This section elaborates on the Adapt Case Model (ACM), which allows to specify the concrete adaptation behavior in terms of event-condition-action rules. These rules are described using extended UML Activities.

Adapt Cases. Figure A.27 shows the meta model excerpt that describes the basic concepts of Adapt Cases. An AdaptCase inherits from UML UseCases and UML Classes. The former inheritance allows AdaptCases to be displayed in use case diagrams and to be related to other use cases, e.g., with an «*adapt*» relation, i.e. an AdaptCase adapts the behavior another use case exhibits. Being a class, an AdaptCase may receive signals which have been thrown e.g. by environment components.

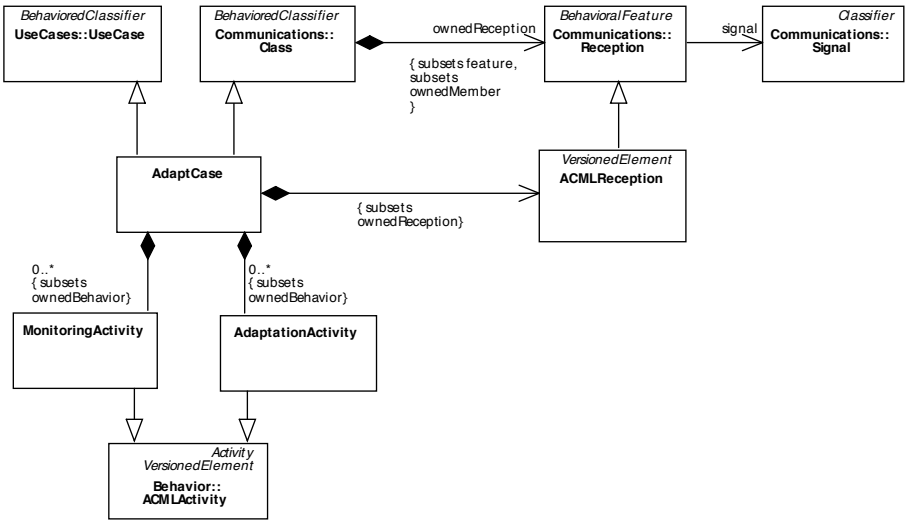


FIGURE A.27. Meta Model: Basic Adapt Cases

Figure A.28 shows an AdaptCase in the right system that may receive a signal named *Signal1* and additionally has a condition (i.e. pre or post condition) defined. The AdaptCase defines an adaptation of the use case shown in the left system.

Since an AdaptCase is a BehavioredClassifier, it may own UML Behavior, e.g. activities. An Adapt Case contains the specialized behaviors MonitoringActivity and AdaptationActivity, both inheriting from ACMLActivity. The Mon-

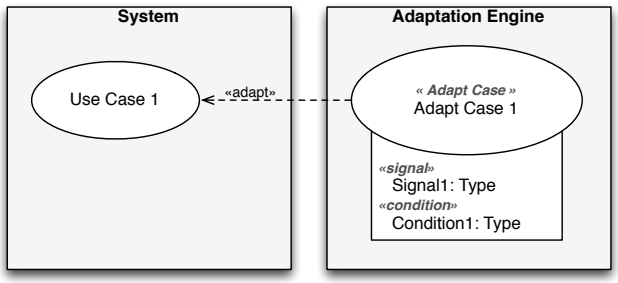


Figure A.28.
An AdaptCase with Signals and Conditions

itoringActivity is meant to monitor sensors and analyze the gathered data to eventually trigger an AdaptationActivity. The AdaptationActivity plans the adaptation, e.g., with the help of sensors and executes the adaptation using effectors.

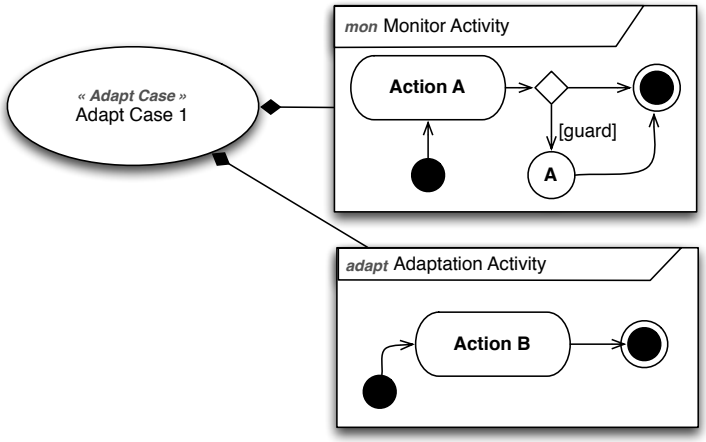


Figure A.29.
An AdaptCase with attached Monitoring and Adaptation Activities

An Adapt Case with attached Monitoring and Adaptation Activity is shown in Figure A.29. After the monitor has performed *Action A* it checks the guard. If the guard is true, the Adaptation Activity is triggered, otherwise the monitor just exits. Triggering the Adaptation Activity is done using a Call Adaptation Activity which is described in the following.

Call Adaptation Activity. To trigger a particular AdaptationActivity, a MonitoringActivity may contain a CallAdaptationActivityAction that is shown in Figure A.30. The CallAdaptationActivityAction inherits from the UML CallBehaviorAction and points to the specific AdaptationActivity that shall be started. If the are multiple AdaptationActivities available, the concrete syntax carries the name of the target activity.

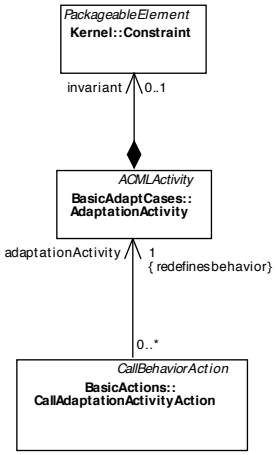


FIGURE A.30.
Meta Model:
Adaptation Activity

Figure A.30 further shows AdaptationActivities to have an additional invariant Constraint. This invariant has to hold throughout the complete execution of the AdaptationActivity. The UML already defines pre and post conditions for behavior. Constraints my be visualized in a separate compartment of the Adaptation Activity as shown in Figure A.31.

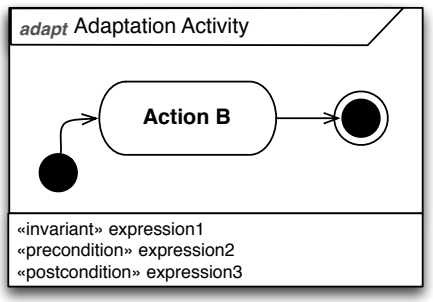


FIGURE A.31.
An Adaptation
Activity exposing
some Constraints

Basic Adaptation Actions. Besides the CallAdaptationActivityAction, the ACML defines several other new actions, the structure of which is shown in Figure A.32. The figure shows the four main categories of actions that are meant to be mainly used in effector definitions: StructureInstanceAdaptationAction to adapt an instance's structure, StructureTypeAdaptationAction, to adapt the structure on type level, ActivityInstanceAdaptation to adapt an activity on instance level (e.g., state changes), and ActivityTypeAdaptationAction to adapt an activity on type level (e.g., reordering or removing particular actions).

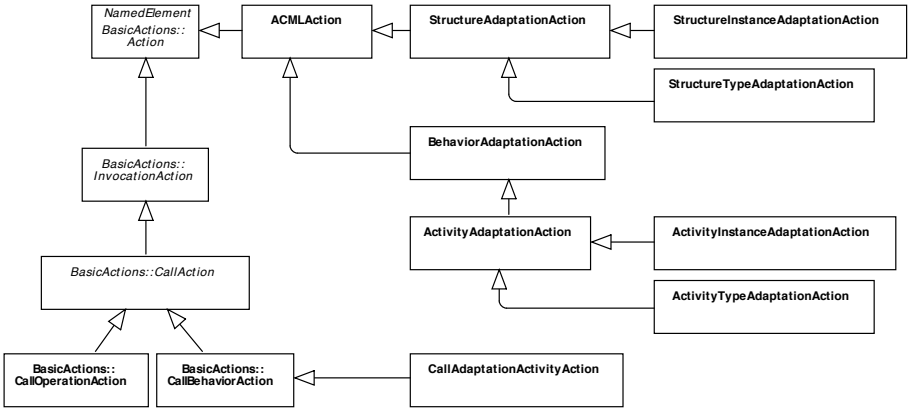


Figure A.32.
Meta Model: Basic Adaptation Actions

To adapt instance or types, they have to be addressable. Figure A.26 showed the existence of Variables and TypeVariables that may hold concrete instances and types. Figure A.33 shows the corresponding SelectionActions that select an instance or type and save them into variables or type variables. The selection is performed using a selection criteria, a concrete UML ValueSpecification.

Concrete actions and their concrete syntax have been defined in [Mut12].

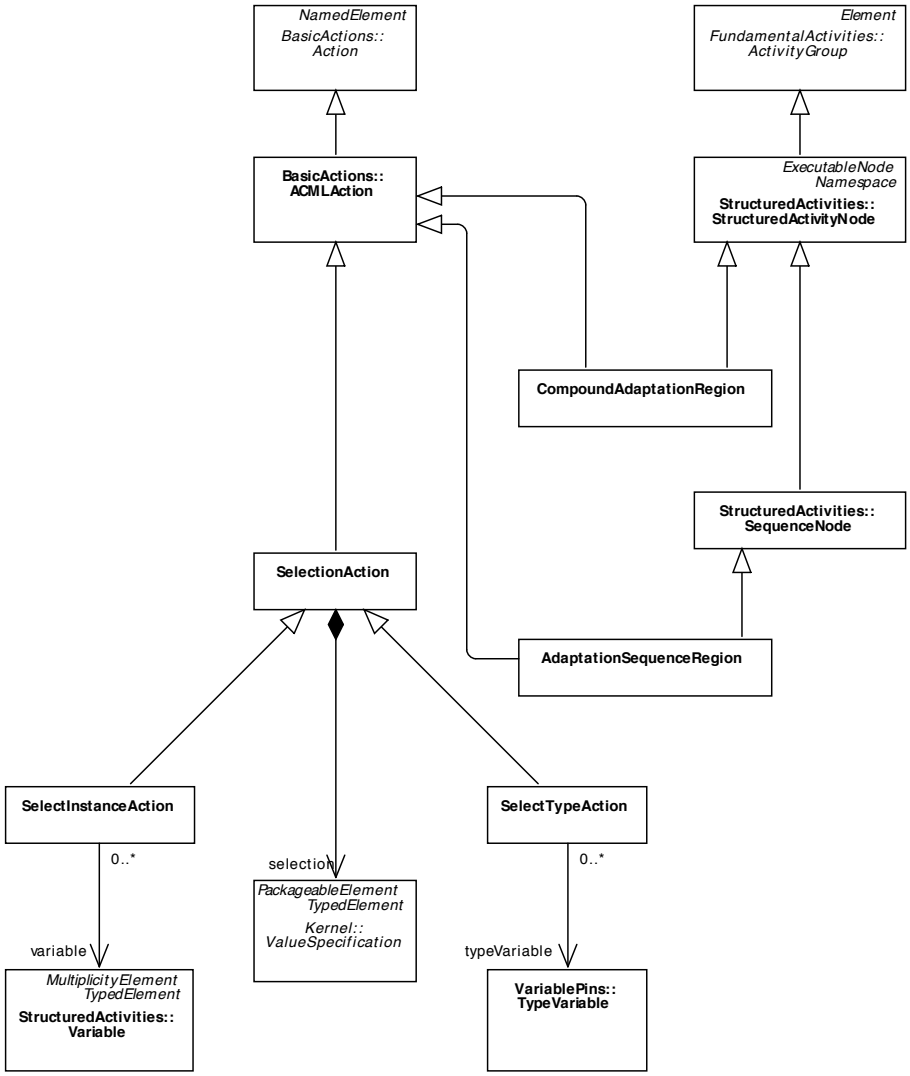


FIGURE A.33.
Meta Model: Adapt Case Helper Actions