MODELING RUNTIME ADAPTABILITY

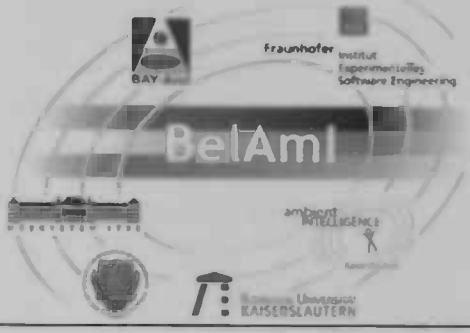
Diploma thesis

Ewoud Werkman

February, 2007

Supervisors:
Martin Becker, Fraunhofer IESE, Kaiserslautern
Paris Avgeriou, University of Groningen, Groningen

RuG



IESE

Fraunhofer Institut
Experimentelles
Software Engineering

TABLE OF CONTENTS

PREFACE

Writing this diploma thesis is the last step in the process of finishing my computer science study. During that last step, many decisions have to be taken, varying from "What will be the subject?" and "Where do I start?" to "How do I validate my research?". Each of these decisions must be well-considered, while every choice has a certain impact on the final result of the thesis. Therefore, deciding is difficult when the impact is unclear.

The word 'decision' comes from the Greek 'κριοις', which can be literally translated to the word 'crisis' in English. It seems that the ancient Greek already found out that deciding felt like having a crisis. The same holds for me: a crisis was looming when the impact of a decision was unclear. To assist me in these decisions, several people guided me and clarified the impact of the decisions when they were unclear to me.

I want to thank Martin Becker, my supervisor at the Fraunhofer Institute of Experimental Software Engineering (IESE) in Kaiserslautern, Germany. The amount of knowledge I gained under his supervision during my internship and thesis research on Software Engineering and Architecture in general, and Software Product Lines in specific, has been invaluable.

Furthermore, my thanks go out to my colleagues at the Product Line Architectures department at IESE and the researchers at the Software Engineering and Architecture (SEARCH) department at the University of Groningen, the Netherlands, with Paris Avgeriou, my supervisor in the Netherlands, in particular.

This work has been carried out in the BelAmI (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) project, funded by the German Federal Ministry of Education and Research (BMBF), Fraunhofer-Gesellschaft, and the Ministry for Science, Education, Research, and Culture (MWWFK) of Rhineland-Palatinate in Germany.

Groningen, February 17, 2007

Ewoud Werkman

1 INTRODUCTION

This chapter introduces the Ambient Intelligent domain, a domain that has recently attracted the attention of researchers and practitioners around the world. As it is a new domain, several problems still need to be solved. This thesis focuses on *runtime adaptability*, one of the requirements of an Ambient Intelligent application.

The next section describes the context in which the research is carried out and why this research is necessary. The subsequent sections analyze the exact nature of runtime adaptability and propose a solution for handling runtime adaptability in software applications. Furthermore, the research goals are enumerated and, at the end of this chapter, an overview of the rest of this thesis is given.

1.1 RESEARCH CONTEXT

Improved health-care and living conditions have increased life expectations all around the globe. Although living longer is a very good result of scientific development, it has also its effects on our society. Today, demographic studies show that in Western nations the distribution of age has changed in such a way that the group consisting of elderly people grows bigger and will continue to grow in the future. Figure 1-1 depicts this trend for Germany's distribution of age for the years 1950, 2001, and 2050 (predicted, other western countries show a similar trend).

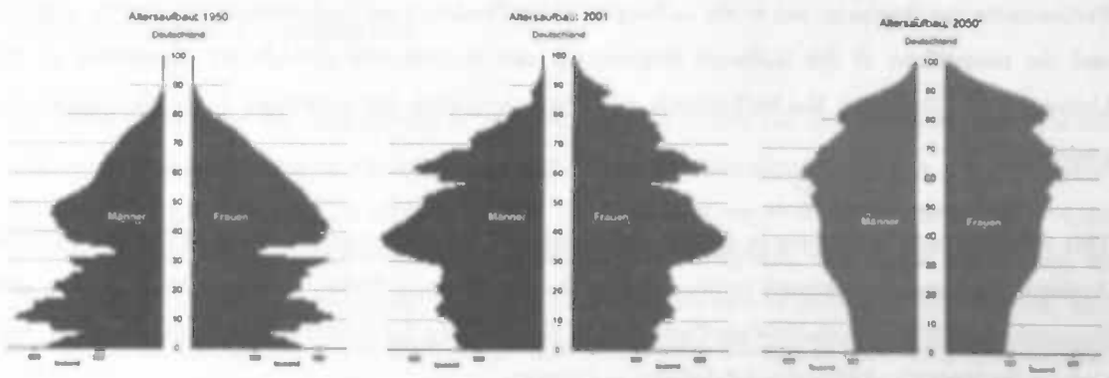


Figure 1-1: Age distribution in Germany (= predicted, source: Statisisches Bundesamt, BMGS, DIW).*

Age affects the capabilities of elderly people. In the old days elderly people stayed with their children to be supported when they grew old. Nowadays we value (among others) individuality more and want to stay independent as long as possible. When staying independent is not possible anymore, due to e.g. physical or mental health problems, nursing homes are there to take care of those elderly and provide a convenient place to live. Although nursing homes are convenient, they take away a lot of our independence. Additionally, they cost money that has to be provided by the working part of the population. Since that part of the population is not growing, we will encounter a point at which the cost for supporting the elderly cannot be afforded by the working population anymore. Therefore, there is a

need to reduce the load on nursing homes and have a cost-effective way to let people stay longer independently in their own house.

Governments have identified these problems as well and made money available for addressing them. The German Federal Ministry of Education and Research (BMBF), Fraunhofer-Gesellschaft, and the Ministry for Science, Education, Research, and Culture (MWWFK) of Rhineland-Palatinate are funding a project called BelAmI (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) [12]. The project is carried out by the University of Budapest and the Bay Zoltán Foundation in Hungary, and the University of Kaiserslautern and Fraunhofer IESE in Germany. One of the project foci is to support elderly people to stay longer at home and as a result alleviate the pressure on nursing homes in the future.
Several interviews with domain experts in nursing homes showed that many elderly people need to leave their own house, because they do not drink enough, eat bad food or do not feel confident enough to do their daily routine by themselves. Providing a computer system that is able to assist with drinking, eating and is able to increase confidence by detecting possible emergencies of elderly people, will extend the stay of those elderly people at their own house and makes a step towards lowering the pressure on the nursing homes and its costs.

Gordon Moore, co-founder of chip company Intel, predicted in 1965 that in the following ten years the computational power of microprocessors roughly doubles every 18 months [23]. Astonishingly, this prediction known as 'Moore's law', still applies. Besides the possibility to compute faster than ever, the increase in computational power of microprocessors in the last decennia has some interesting side effects. First, mass-production of microprocessors has decreased the costs considerably, and second, the sizes of microprocessors have decreased too. Third, the energy consumption of microprocessors has been reduced. Together, these effects offer the possibility to create powerful microprocessors that are small, cheap, and able to run for a long time on small batteries. This makes it feasible to embed them in all kinds of objects in our daily life.

Creating environments equipped with embedded computers is called 'pervasive' or 'ubiquitous' computing [46]. Ambient Intelligence [1, 21, 32], for short AmI, is seen as the paradigm that enables the creation of a new generation of systems that assist people in their everyday lives. Highly portable, numerous, cheap and embedded devices, distributed transparently all around us create an environment with great potential. Equipped with flexible, intelligent and anticipative software, it allows us to create applications that are *adaptive* to the user and its environment in order to support the user as good as possible. Instead of a computer-central environment, an ambient intelligence environment puts the user central, and makes his wishes and needs a key priority. Consequently, human-machine interaction is an important enabler for this paradigm and requires the use of multi-modal user-interfaces, such as speech recognition, gesture classification and situation assessment. Current microprocessor technology is able to acquire and process the information produced by these interaction mechanisms, which allows elderly people – who have not grown up with computers and its (difficult) handling – to use such a system.

Combining current microprocessor technology and the Ambient Intelligent paradigm together is a promising approach to handle the upcoming problems of the distribution of age in our nations.

In order to make the Ambient Intelligent paradigm reality, analyzing its devices and applications is needed. The AmI devices share one or more of the following properties that influence the complexity of the AmI system:

Embedded:	requires small devices, and are therefore less powerful than a normal desktop or server system;
Transparent:	requires small devices, which are easy to integrate into everyday objects in such a way that the user is not aware of them;
Distributed:	requires networks to connect distributed devices;
Numerous:	requires cheap devices;
Portable:	requires low-power consumption and wireless network connectivity. Additionally their weight should be low;
Cheap:	requires mass-produced microprocessor technology.

Because of this combination of properties (that comprise the term *Ambient*), each AmI device tries to focus on a small part of the functionality that needs to be addressed in the domain. Therefore, different and optimized hardware solutions are used to implement that part of the functionality and consequently create a heterogeneous environment. In an AmI environment, all these heterogeneous devices need to be integrated, which is a complex task.

The domain model for the AmI domain is depicted below in Figure 1-2. It clarifies the concepts and relations among these concepts which are used in the examples throughout this thesis. The AmI system shown in the figure integrates all kinds of Devices. These devices can be divided in sensor devices and actuator devices. Sensor devices measure the environment of the system, such as the temperature, the location of the elderly in the house, etc. The Actuator devices are able to change the environment, such as lowering the temperature of the room, or giving a dehydration warning to the elderly. The Actuator can also change the system itself, when it needs to improve its own functionality (e.g. using a different actuator to remind the elderly of drinking). The Reasoning maps the sensor data on actuator actions in an intelligent way. The environment defines everything measurable around the AmI system, such as the activities of the elderly, temperature of the room, etc.

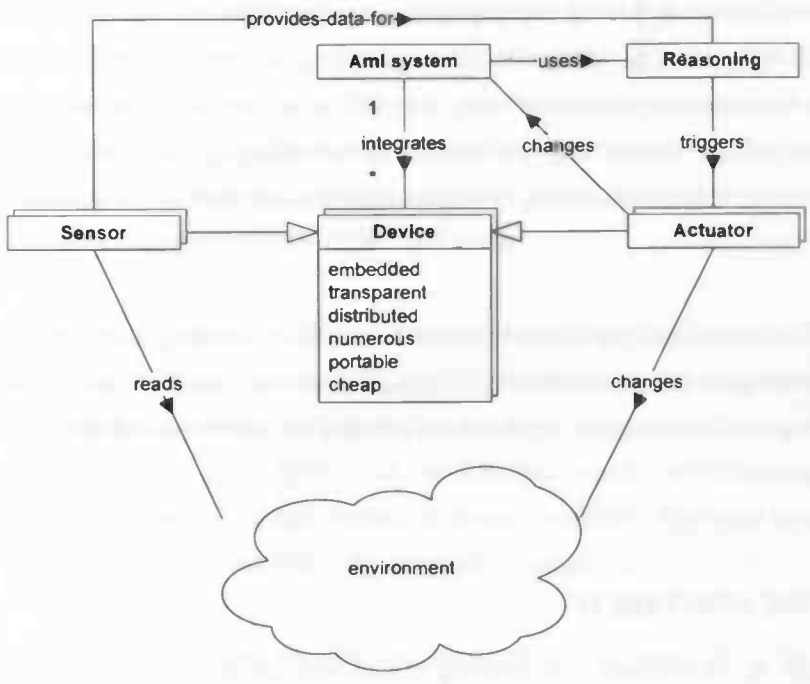


Figure 1-2: Domain model for the Ambient Intelligence domain.

Due to the nature of the devices and the associated requirements, an AmI system is constrained by its resources. For example, the available network bandwidth, processing power, battery power, network connectivity or portability, to name a few. Furthermore, AmI systems are constraint by their quality attributes, such as dependability. When an AmI system assists elderly people, wrong advice by the system (due to wrong reasoning) is unacceptable, as people's lives may be at stake. Therefore, the quality requirements of an AmI system are very strong. To address these requirements, the software of the AmI system requires several key properties (among others):

Anticipative: ability to detect changes and act upon these changes (i.e. the detection of a different situation);

Intelligent: ability to act upon changes in such a way that it changes the system as good as possible (that is why it is called Ambient *Intelligent*).

Dependent: ability to function properly in any situation (e.g. if one of the AmI devices breaks down the rest of the system should still function.);

These properties allow the application to be adapted to a behavior that fulfills its requirements in the changed situation as good as possible. To alter the behavior of the application, the application itself has to be adapted. Accordingly, the software of the AmI system needs to be:

Flexible: ability to adapt the application to new requirements induced by the current situation of the system;

Extensible: ability to add and integrate new (heterogeneous) devices;

Important here to notice is that all the key application properties for an AmI system require the application to be "adaptable" or "changeable". Since the changeability of an application normally refers to the easiness to change the application code, *adaptability* is used to express the ease with which a system or parts of the system may be adapted to the changing requirements [42]. Therefore, adaptability is a key enabler for ambient intelligence applications such as the assisting system for the elderly.

There is some confusion about the difference between adaptive systems (systems that adapt *themselves* to a particular deployment environment) and adaptable systems (systems that can be adapted to a particular deployment environment). Appendix A: explains the confusion and describes the definition used in this thesis.

1.2 RUNTIME ADAPTABILITY

In order to fulfill the requirement to exhibit application behavior that works as good as possible in a changing environment, the application has to be adaptable at runtime, i.e. after the application has been designed, deployed and started. This means that the application has to be designed for supporting runtime adaptability.

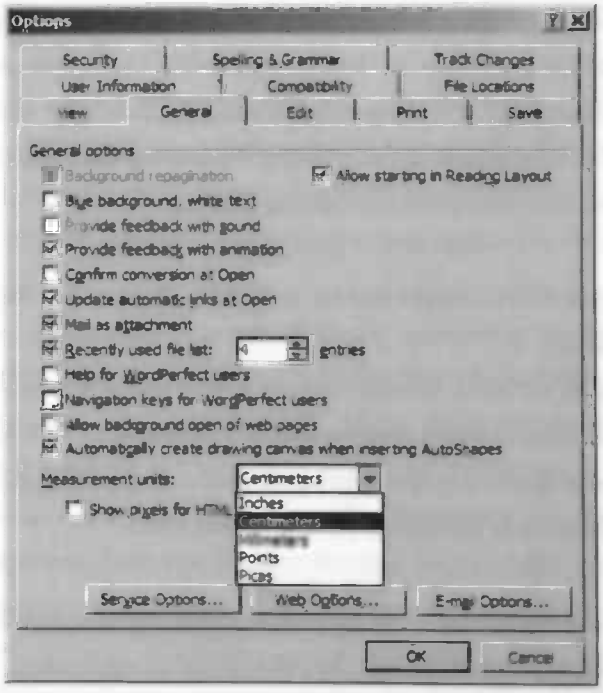


Figure 1-3: Runtime software adaptation in Microsoft Word

Runtime adaptability refers to the easiness in which an application can be changed, after it has been developed, deployed and started. Runtime adaptability is not new. Specific mechanisms, such as parameterization or selection, allow the behavior of an application to be adapted at runtime and are used in most software nowadays. In the case of selection, an option dialog is shown that allows the user

to set certain options of the software at runtime, for example. Figure 1-3 shows such an option dialog in Microsoft Word. An explanatory text is attached to the option that describes what the option does, and, in many cases, a tool tip is shown when the mouse is held stationary over the option to explain the option more thoroughly. In several applications, a help button is presented to find out all the details of an option. All these sources of information allow the user to reason about these options and carefully select the right option to improve the utility of the application.

The difference with AmI applications is that the adaptation is not made by the user of the application, but by the system engineer or the application itself. In the latter case, these applications are called (self-) *adaptive*, which means that they are capable to adapt their own behavior in response to changes in its operating environment [35]. Since such applications cannot understand option dialogs, explanatory texts, and tool tips or help files written for human readability, they need other ways to find out what can be adapted, and what the effect of that specific adaptation is.

IBM and others identified the need for adaptive applications, too, since the costs and time for administration and troubleshooting of complex applications increased. IBM's autonomic computing initiative [28] tries to address these problems and identifies four types of self-management behavior, based on the autonomous behavior of the human body:

- Self-configuring: Increased responsiveness; Adapt to dynamically changing environments.
- Self-healing: Business resilience; Discover, act and diagnose to prevent disruptions.
- Self-optimizing: Operational efficiency; Tune resources and balance workloads to maximize use of IT resources.
- Self-protecting: Secure information and resources; Anticipate, detect, identify and protect against attacks.

An Ambient Intelligent system requires all these four types of self-management. Therefore, these four types of self-management cover the architectural drivers for runtime adaptability.

Although self-management systems – such as AmI systems – need to be adaptive, this thesis will not provide a complete solution for it. Adaptivity consists roughly of the following three steps:

- Sense: detect changes in the system (by using Sensors in Figure 1-2);
- Reason: analyze changes and search for proper actions to handle the change (by using Reasoning in Figure 1-2);
- Act: issue actions to adapt the system (by using the Actuators in Figure 1-2).

This thesis addresses a part of the last point: in order to adapt the system at all, you need to know what the system can adapt, how it can be adapted, and what effect it will have on the system (cf. Section 2.5). Without this semantic base for adaptability at runtime, the rest of the steps for adaptivity are not possible. Therefore, this thesis will focus on a solution to describe and use runtime adaptability in a

software system. The question of *who* will adapt, i.e. the system itself in case of self-management or a person, is not relevant in this thesis, as both require the system to be runtime adaptable.

Approaches such as Chameleon [43] and Koala [45] support runtime adaptivity (see Chapter 3 on Related work). They have specific functionality for detecting that an adaptation is necessary and use predefined configurations (i.e. created by the developer at development time) to switch to different behavior at runtime. The effects of this adaptation are implicitly defined in the configuration and the application itself has no knowledge on what effects a configuration change will have at runtime. The adaptation behavior of these approaches is therefore fixed at runtime and consequently not sufficient for the dynamic and evolving environment of AmI systems (see Section 2.1). AmI systems need to determine the right configuration at runtime to fulfill the requirements for an anticipatory and flexible system. To this end, an AmI system has to be provided with adaptation knowledge that forms the semantic base for the adaptation decisions at runtime.

1.2.1 ENGINEERING RUNTIME ADAPTABILITY

The principle of separation of concerns, identified by E.W. Dijkstra for the programming discipline in 1976 [20], is one of the most important principles of engineering. As we cannot address many concerns (functionality, quality, costs, etc.) at once, this separation allows us to focus on a single concern (e.g. adaptability) at a time, allowing us to explicitly design for such a concern.

In order to create a sound solution for adaptability, it is important to separate the functional behavior (i.e. the actual function of the application) and adaptation behavior (i.e. the quality to support change) of the application [35]. Otherwise, the increased complexity induced by incorporating adaptability in an application will be very difficult to handle or will result in poor application design.

This requires a software engineering approach. Software engineering "is the engineering discipline which is concerned with all aspects of software production that uses software engineering methods to provide a structured approach using system models, design advice and process guidance to create a solution for the problem at hand" [41]. A key aspect of software engineering is the design of a software architecture. The architecture of a software system is "the structure or structures of the system, which comprise software components (software building blocks), the externally visible properties of those components and the relationships among them" [6]. Software architecture allows for early assessment of and design for quality attributes of a system such as adaptability, since it is the first step after the specification of the requirements in a software engineering process [14]. This early assessment is necessary since the impact of the software architecture on the final implementation and quality of the software is considerable.

A process for engineering adaptability will help us to separate the different concerns of the software application and provide the ability to focus on the adaptability aspect only. Such a process is necessary,

due to the complexity of incorporating adaptability in a software application. The process for engineering adaptability will be model-based. Models provide a very adequate abstraction of a system and uses specific guidelines to structure the system. Furthermore, when models replace the otherwise ad-hoc implementation of adaptability, the quality of the resulting application improves, as models can be analyzed, validated, and simulated before the system is put to use. This quality improvement is required for self-management systems.

As modeling only describes the system on an abstract level, the process for adaptability describes which steps need to be taken to transform this abstract model to code that can be incorporated in the software application. This engineering methodology is called Model-Driven Engineering (MDE) and is better known as the Model-Driven Architecture (MDA) [33] initiative by the Object Management Group (OMG) that provides guidelines to support the model-driven engineering of software. MDA is accompanied by several standards, including UML, the Universal Modeling Language developed by OMG [34], and will be used throughout this thesis for modeling. Consequently, this thesis will use a model-based process to describe and use runtime adaptability.

1.2.2 SIMILARITIES WITH VARIABILITY MODELING

The example shown in Section 1.2, which changes Microsoft Word's behavior at runtime using an option dialog, uses a variability mechanism called 'selection'. Variability mechanisms are thoroughly studied in the area of software product lines (SPL) [14, 15, 31]. An SPL is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [40]. These common assets must be applicable in different contexts for different applications created from the product line infrastructure, otherwise reuse of those assets – one of the main goals of SPLs – is not economically feasible.

Therefore, common assets often possess variability. Variability refers to the points where the asset can be changed, the so-called variation points [31]. In the case of the example for selection, every variation point has a set of alternatives, called variants, which can be chosen from. For the 'measurements unit' in Figure 1-3 the user can select from the variants 'centimeters', 'millimeters', 'inches', 'points' or 'picas'. (The SPL concepts and process is elaborated in Section 2.2)

There is an interesting parallel between variability modeling and runtime adaptability modeling [8]. In general, variability allows the developer to reuse and adapt the asset to fit in a specific context. Although this context is considered static when developing an application from an SPL point of view, the concept that the asset can be changed to match a certain context is similar to what is needed for an AmI application when adapting to a different situation at runtime. Thus, the variability of an asset can be exploited at runtime to provide for the adaptability needed for self-management systems.

To visualize this similarity, Figure 1-4 shows two 'design pyramids', one for traditional software development (a) and one for the development of a runtime adaptable system (b). A design pyramid shows the complete design space of possible design decisions during different phases of the software development. The inception of an application starts with an idea. At that point, there are an uncountable number of ways to come to an implementation, both realizable (Realizable designs) and unrealizable (the complete design space includes unrealizable designs, too).

When the requirements for a product are being defined, design decisions are made. These design decisions add detail to the design of the application, but also constrain the solution space of the final implementation. This happens for every design decision at every level of abstraction. An important difference between design decisions on different levels of abstraction is that decisions taken on higher levels have more impact (as they constrain the solution space more) than those on lower levels. As a consequence, decisions taken at higher level of abstraction are much harder to revoke (as the whole process must be restarted to that point, which is very costly [13]) than decisions on lower levels.

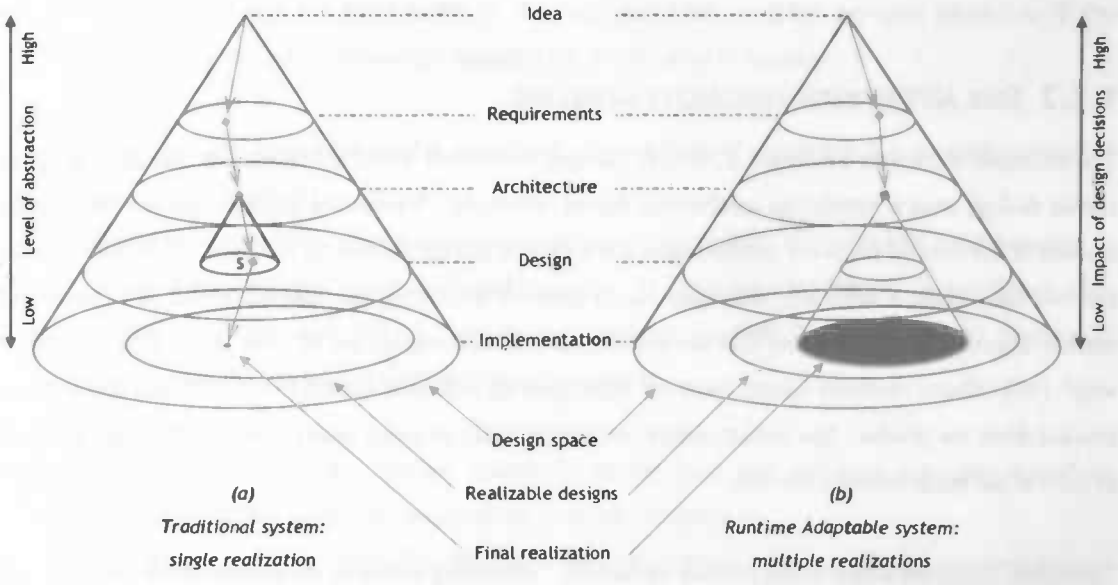


Figure 1-4: Design pyramids for traditional software engineering (a) and the engineering of runtime adaptable software (b).

For the decisions on the architectural level, only design solutions contained in the area denoted by S are possible in Figure 1-4 (a). As more architectural decisions are made, space S narrows down to a single design solution, depicted by the square dot. When the last design decision in the whole process has been made, a single realization is acquired: the final result of all the design decisions.

Software Product Lines allow for the specification of variability. On the architectural level, variation points can be specified that allow variation in the lower levels of abstraction. Consequently, every design in space S in Figure 1-4 (a) is part of the product line, as it describes the possible design solutions. When a product is derived, the open variation points are resolved (by making decisions),

based on the configuration of a specific product, and a single solution is acquired. When different decisions are made during derivation, a different product is acquired. This way, multiple products can be made from a single architecture.

Runtime adaptable systems as regarded in this thesis, use the same process as SPLs. This process is depicted in Figure 1-4 (b). The difference is that during product derivation, not all variation points are resolved, but some are deliberately kept open. The final solution will still have open variation points at the implementation level, and the decisions for resolving these open variation points are delayed to the runtime. The kept variability is exploited at runtime to change the application by switching the implementation variants of a variation point. This means that the final realization covers multiple realizations that are all contained within a single implementation, as depicted by a big solution space of the final realization, shown in Figure 1-4 (b).

In SPLs the decision of which variant to use in which context is made by the system engineer during the design and implementation of the application. In the decision model of the asset, each variation point, its variants, and the consequences of a decision when choosing a certain variant, is documented in an informal form [31] (i.e. not interpretable by computers). This is not usable for runtime adaptability, since such an informal document cannot be used at runtime. A formal specification is needed for that. Furthermore, the consequences or impact of a decision is often described in a functional way, i.e. the impact on the structure of the application. Runtime adaptability also needs the non-functional impact of decisions (i.e. the influence on quality attributes of the system). Since decisions have to be taken during the execution of the application, the knowledge expressed in the decision model must be made available in a computer-interpretable format at runtime and extended with the influence on the quality attributes and resources of the system. In this way, adaptability can be expressed by using concepts and models from the variability management of SPLs.

1.3 CONTRIBUTIONS AND RESEARCH GOALS

The development of runtime adaptable systems, such as AmI systems and self-management systems, is very complex. To reduce this complexity, this thesis' goal is to describe a model that contains all the necessary information needed to specify runtime adaptability. Such a model will reduce the complexity of adaptation and increase its predictability, because a model allows for the analysis, validation and simulation of a runtime adaptation before it is carried out. As this model is only a specification, the thesis also describes a process that shows how this model can be transformed to create runtime adaptable systems. Due to the similarities with Software Product Lines, the process will be an extension of the SPL process, albeit with a focus on the adaptability at runtime.

Problem statement:

Design a model-based solution for the engineering of runtime adaptability.

The following questions need to be answered to come to a solution:

Research questions:

1. *What is runtime adaptability?* (Chapter 2)
   As the term runtime adaptability comprises different types of adaptation behavior, this question will scope the term to the runtime adaptability that is required in the research context described at the beginning of this chapter and consequently regarded in this thesis. Furthermore, the information that is required to define an adaptation is analyzed. This information answers questions like 'what is adapted?', 'who adapts?' and 'what is the result of this adaptation?'.

2. *What other model-based approaches are available that can be used to engineer runtime adaptability?* (Chapter 3)
   The quest for approaches that consider adaptability in software products is not new and this question analyzes what models are already developed in this area of research. Several approaches focus on adaptability at development time, and the analysis identifies if the concepts of these approaches are reusable for adaptability at runtime.

3. *What are the requirements for specifying a model-based approach for runtime adaptable systems?* (Chapter 4)
   Based on the analysis of the approaches, this question defines the steps that are necessary to create a model-based runtime adaptable system. It describes the entities and relations of a model-based approach and analyzes the requirements for both the model and the process to incorporate the model in a runtime adaptable system.

4. *How does a model for runtime adaptable system look like and how can this model be incorporated into a software engineering process.* (Chapter 5)
   The answer to this question describes how the requirements of the model are converted into a model for runtime adaptability. Additionally, the process to incorporate the model in an application is described.

5. *Does the model-based solution satisfy the requirements?* (Chapter 6)
   This question answers if the requirements are met for the model and process defined by the previous question.

1.4 OVERVIEW

The remainder of this thesis is organized as follows. Chapter 2 introduces the concepts and scenarios for runtime adaptability that are used throughout this thesis. It will also introduce a running example

that is used to clarify the statements made in this thesis. Furthermore, it will divide runtime adaptation into different types to scope the runtime adaptability described in this thesis. Additionally, the knowledge necessary to describe runtime adaptability is analyzed in this chapter. Chapter 3 focuses on variability modeling from SPLs and adaptable systems by looking at related work. It analyzes eight approaches that all describes parts of the model that is needed for runtime adaptability. Chapter 4 will use this analysis and the scenarios introduced in Chapter 2 as input for defining the requirements of a model-based approach for runtime adaptability. Chapter 5 shows how a model for runtime adaptability looks like and how this model can be incorporated in the software engineering process. Chapter 6 validates the model theoretically using ATAM and shows that the model can be used to describe the adaptability of an application. A prototype tool that leverages the creation of runtime adaptable systems is developed to show that the model can be used in practice. Chapter 7 concludes this thesis by presenting the validation results and answers the research questions from the previous section. Chapter 8 discusses the whole approach and points to future directions of research on runtime adaptability.

2 ADAPTATION SCENARIOS AND CONCEPTS

This chapter introduces two scenarios and a running example that are used throughout this thesis to clarify the statements made, identify the requirements for runtime adaptability, and to validate the research later in this thesis. Furthermore, it scopes the research to a specific kind of runtime adaptation, online-determined reconfiguration, and shows why this is different from other kinds of adaptation behavior. The chapter ends with an analysis of the modeling of adaptation.

Chapter 1 identified the need for runtime adaptability in self-management applications in general and in the AmI domain – the domain from which this work emerged – in specific. Furthermore, it identified that not only modeling of adaptation is required to reduce the complexity involved in runtime adaptability, but that the process of incorporating such a model into an application is also of great importance.

As this work is based on the similarities with SPLs, the process that is used in SPLs to derive a product will be taken as model for the process of creating a runtime adaptable system. As it is important to know how this process works, the next section will describe this process in the context of an Ambient Intelligent Home Care System.

2.1 TYPES OF ADAPTATION BEHAVIOR

The term 'runtime adaptability' covers wide area of adaptation behaviors. To scope this thesis, the different adaptation behaviors are subdivided in Figure 2-1. This subdivision is taken from Trapp [43].

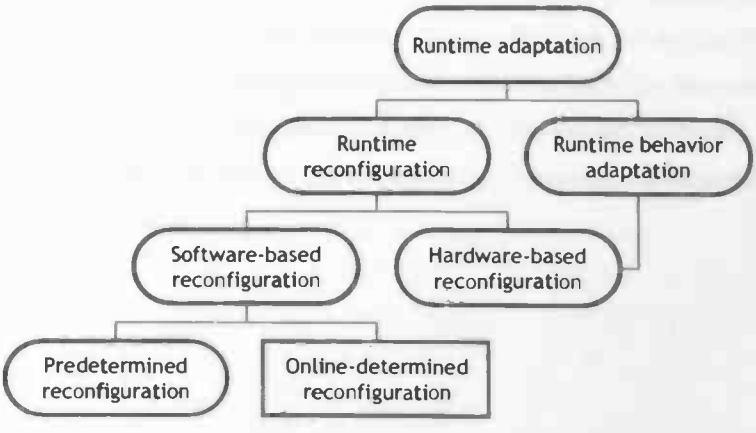


Figure 2-1: Different types of adaptation behaviors (taken from [43]).

Runtime adaptation comprises Runtime Reconfiguration and Runtime Behavior Adaptation. Runtime behavior adaptation allows the system to change its behavior completely, especially to unforeseen situations. This allows for enormous flexibility. Such systems use e.g. neural networks and genetic

algorithms to find the correct behavior. As this kind of adaptation behavior is hard to predict, such behavior cannot be used in AmI-based applications, currently.

Runtime reconfiguration on the other hand defines constraints for the behavioral possibilities at runtime. Runtime reconfiguration can also be subdivided in two different behavioral variants: Software-based and Hardware-based reconfiguration. Hardware-based reconfiguration systems are based on e.g. FPGA (Field-programmable Gate Arrays) and can reroute the programmable circuitry of the system. As the extend of reconfiguration can differ, hardware-based reconfiguration also falls in the category of runtime behavior adaptation, since it is possible to completely reprogram the hardware, giving it a complete different behavior.

Software-based reconfiguration can be subdivided in predetermined reconfiguration and online-determined reconfiguration. Predetermined reconfiguration focuses on adaptation behavior that is predetermined during the development of the application. Predetermined reconfiguration is based on the definition of configurations. Each of these application configurations can be tested, validated and simulated on forehand to make sure that the application's (adaptation) behavior is predictable when the application is deployed. Domains such as the automotive domain require this predictability, as selecting a wrong configuration might affect peoples lives.

Online-determined reconfiguration is the focus of this thesis. This type of adaptation behavior allows for the online determination of the correct configuration. Consequently, the developer is alleviated from the task to specify all the configurations at development time. Furthermore, online-determined reconfiguration behavior allows the application to be extensible, as the application can repeat the determination of the correct configuration after the extension. This allows for more flexibility than predetermined reconfiguration.
Two different subjects can do this online determination:

1. An operator of the system (e.g. the system engineer).
2. The system itself (creating an adaptive application).

Both subjects require additional knowledge at runtime that describes the system, in order to make well-considered adaptations. The modeling of this knowledge is subject in this thesis. Furthermore, when this thesis refers to runtime adaptability, online-determined reconfiguration is meant.

2.2 SOFTWARE PRODUCT LINES

Ambient Intelligent Home Care Systems [9], developed in the BelAmI-project, are systems that assist elderly and handicapped people to live longer in their own house. Those systems integrate several heterogeneous subsystems that address different goals for the monitoring and assistance of the elderly and handicapped people. The combination of these subsystems makes it not only possible to fuse information from different subsystem's sensors into information with higher accuracy and certainty,

but also use different actuators for achieving a certain goal. This allows for better assistance and a longer stay at home.

The demands for assistance differ among the assisted persons and change during the use of the system, as the capabilities and situations of the assisted persons change. To be able to address all possible ways of assistance, all of these heterogeneous subsystems – either needed or unneeded – should be included in the resulting system. However, this approach strives against the goal to keep the costs low and address the problems concerning the high costs of nursing homes. Consequently, there is a need to tailor the assistance system for the assisted person and only install the components that are useful for his situation.

Variability modeling techniques in SPL approaches allow for the identification of the common and variant parts in the Home Care System. Such techniques lower the complexity of composing a usable system, which increases the overall quality and applicability of the system. Creating a software product line for the assistance system allows the selection of exactly those components that address the assistance demands of the assisted person.

2.2.1 DOMAIN CONCEPTS

Figure 2-2 depicts the domain model for a SPL, with a focus on the variant parts. The Product line is defined by the Requirements engineer, and consists of multiple products. The requirements engineer also defines the requirements for each of the products of the product line. Each Product has Variation points that realize the variability of the product. Each variation point has Variants, which describe the possible variations for that particular variation point. These variations can be values for parameters, alternative implementations for specific functionality, etc. Each variant describes specific functionality, which have consequently different quality. The functionality of a variant, implemented by the Software developer, influences the behavior of the product. When a different variant has been chosen for a variation point, the behavior of the product will change. This also holds for the utility of the product. When a different variant has been chosen, the quality of that variant influences the utility of the product. The Software architect defines where the product has its variation points and what variants are available for that variation point.

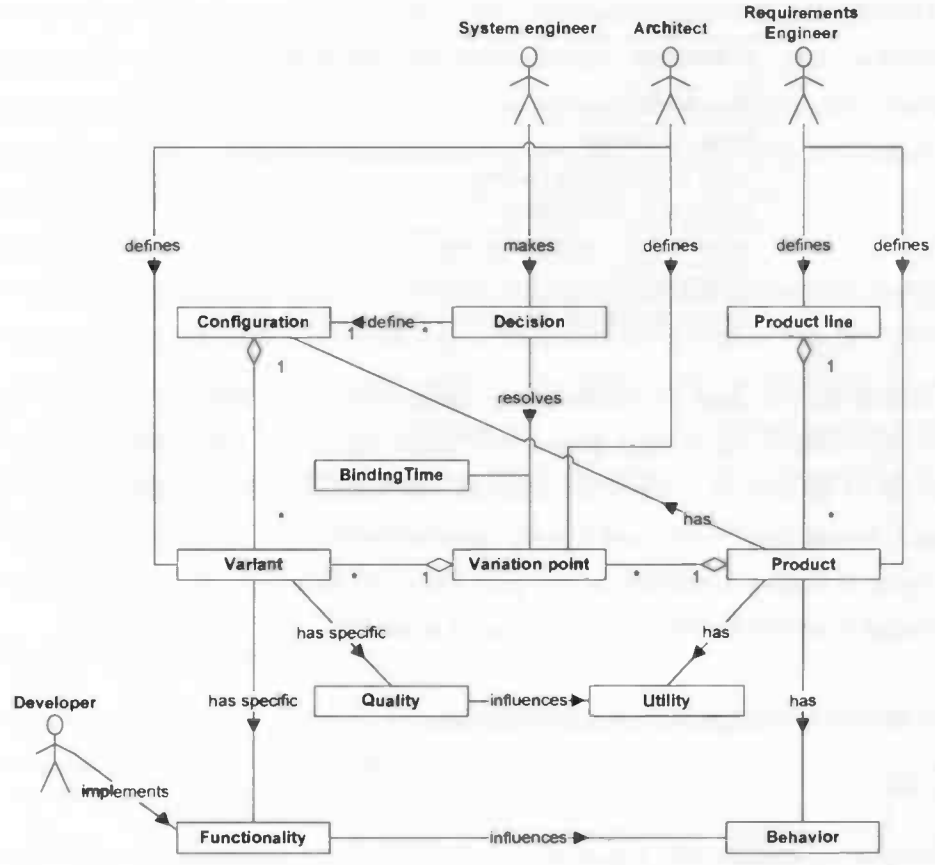


Figure 2-2: Domain model for the variability in Software Product Lines

The System engineer makes decisions for resolving all open variation points. These decisions are documented in a Configuration that describes which variants should be chosen for a particular variation point in a product. The time at which a variation point is resolved is called the binding time. Possible binding times are e.g. development time, compile time, link time, or runtime. When all variation points are closed, the product is ready for shipping to the customer.

2.2.2 EXTENDING THE SPL APPROACH TO THE RUNTIME

An SPL approach for the AmI Home Care System addresses only the requirement that the demands for assistance differ among the assisted persons when an assistance system is installed. To address the change during the use of the system too, it is required that the tailored system possesses more functionality than initially required when assessing the assisted person's situation. This functionality can be enabled when new requirements for assistance emerge while the system is running. Table 2-1 shows the decomposition of the assistance system's functional components.

The first five subsystems address specific assistance functionality. For example, the 'Dehydration assistance' subsystem focuses on measurement of the amount of liquid that has been drunk on a day. The 'Robotic tray carriage system' consists of a robot that is able to navigate through the house and carries things for the assisted person, e.g., when he or she cannot walk easily anymore. The last two subsystems in the list above specify how the system interacts with the assisted person.

Table 2-1: Functional components of an Aml Home Care System

Home Care System's functional components
Dehydration assistance
Food assistance
Light & Electricity control
Emergency Recognition
Robotic tray carriage system
Communication system: {Visual, Speech}
Speech language: {English, French, German, Dutch}

Not all assisted persons need the 'Robotic tray carriage system', which is a relatively expensive subsystem. Furthermore, the assisted person only needs support for one 'Speech language'; other languages can be selected for use in other countries. The 'Communication system' on the other hand will normally include both 'Visual' and 'Speech' communication, since it is often seen that the assisted person's vision or hearing deteriorates as they grow older. The inclusion of both subsystems allows the system to adapt to support the assisted person as good as possible.

Figure 2-3 shows this configuration process graphically:

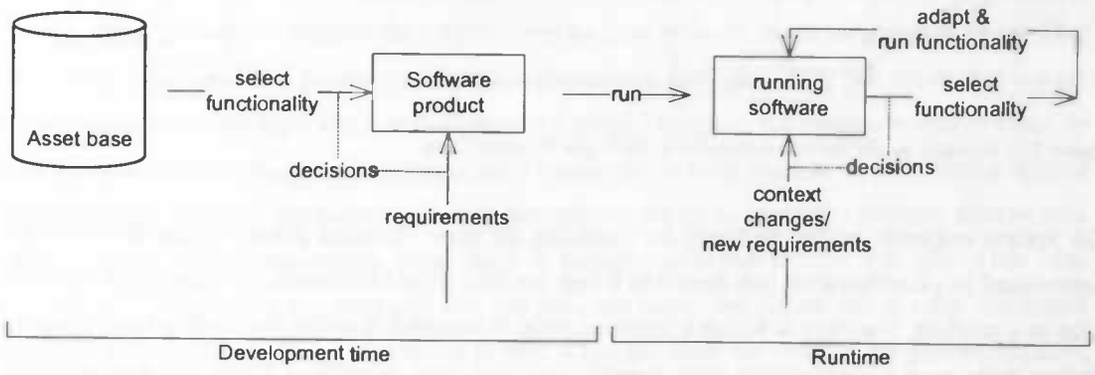


Figure 2-3: Configuration of a software product at development time (SPL approach) and at runtime.

When the process is divided into the activities that take place at development time and at runtime, several similarities can be identified. The software product is composed of functionality provided by the generic Asset base for the product line at development time. This asset base contains assets that can be reused in different products by the exploitation of their variability. Based on the requirements of the software product (in this case the demands for assistance), decisions are made to select the correct functionality from this asset base.

When the system is running, the same process is executed when new requirements arrive while the context of the software application changes or a new variant is added to a variation point. Once more decisions are taken to select the right functionality to adapt the software to meet these new requirements. This functionality must already be incorporated in the product to allow for this adaptation.

As the decisions to include specific functionality in this scenario are based on assessments of people from the healthcare domain (e.g. caregivers) at development time, the decisions after the deployment of the system might be made by different people or the system itself. The rationale for these decisions, i.e. what can be adapted, what impact will it have on the system itself and on the assisted person, must be available at runtime too.

An architectural model that describes the rationale for these decisions and the decisions itself leverages the development of these applications, if and only if this model is available at both development time and runtime. The architectural model must specify where functionality can be added, removed and changed, and for each addition, removal or change, the impact on the system and its requirements should be documented. This documentation process should take place at development time in order to be available at runtime.

2.3 ADAPTATION SCENARIOS

The adaptation process of an adaptable system can be subdivided into two consecutive phases: adaptation at development time and adaptation at runtime.

Adaptation at development time is the focus of a Software Product Line as it allows the products that comprise the product line to be adapted to distinguishable products usable in different contexts. Adaptation at runtime is the focus of runtime adaptability, as it allows the product to be adapted to fit in a different context at runtime while it is running.

In this section, two scenarios are introduced that focus each on one of these phases. The first scenario focuses on the configuration of a product during the development phase; the second scenario focuses on the adaptation at runtime. After these scenarios, the iCup is described, a portable intelligent cup that is used in the dehydration subsystem. This iCup will be used throughout this thesis as a running example.

2.3.1 ADAPTATION AT DEVELOPMENT TIME

Bob is a system engineer at a software company that produces assistance systems for the elderly. As the software system targets different assistance demands they have set up a product line to tailor the system for a specific person.

Today, Bob has received a new request for an assistance system for Mrs. Alice, because she has fallen from the stairs a week ago. He loads the configuration tool on his desktop PC and opens a new assistance system's configuration. This configuration describes which assistance subsystems can be included in the final product. According to the requirements he got from the doctor of Mrs. Alice, Bob selects the right subsystems in the configuration tool for the assistance system. Unfortunately, not all

decisions in the configuration tool can be made, as the doctor was not able to answer all the questions for Mrs. Alice's recovery.

Luckily, the configuration tool allows the decisions to be taken by the system itself after the system has been deployed. Bob selects this option for these decisions. Now, Bob stores the configuration and orders the tool to compose the assistance system's software. A few minutes later, this process is ready and the technicians can install the system for Mrs. Alice.

2.3.2 ADAPTATION AT RUNTIME

The fall from the stairs made Mrs. Alice feel quite insecure and confused. To support her in her recovery at home, the doctor requested besides the dehydration assistance subsystem and food assistance subsystem also an emergency recognition subsystem. The emergency recognition subsystem keeps track of Mrs. Alice's vital functions and alarms the doctor if they deviate from her normal routine. All assistance subsystems share the same wireless access point to communicate with the software that reasons about Mrs. Alice's health.

Unfortunately, Mrs. Alice tumbles again, is unable to move and call for help. As the system detects a deviation from her normal routine, the system decides that it needs more information about Mrs. Alice's vital functions. The current bandwidth consumption however, does not allow this action, because it is saturated by other wireless devices in the room. Therefore, the assistance system looks up which unimportant subsystems use communication bandwidth. It finds out that the dehydration system is not important right now and reconfigures the dehydration system to cache the drinking activity data of Mrs. Alice to use less bandwidth. Now, there is enough bandwidth to look into Mrs. Alice vital functions. As her heart rate has increased and she does not move, the system decides that the doctor needs to be called for help. The doctor drives to Mrs. Alice and finds her lying on the ground. Luckily, she didn't break any bones and the doctor gives her something for the pain. She will recover completely in a short time.

2.4 A RUNNING EXAMPLE: THE ICUP

In Section 2.2.2 the AmI Home Care System was briefly introduced. One of the subsystems is the dehydration assistance subsystem, which measures the drinking activity of the assisted person. These measurements are collected by means of intelligent cups: iCups. An iCup is a cup equipped with an embedded computer, called a Particle computer [19]. A Particle is a small (size of 3 penlights), battery-powered, embedded node that communicates wirelessly with an access point that is connected to the local area network (LAN). On the LAN, an application is installed that processes the data send by the Particles and is e.g. able to send a reminder to the assisted person by speech synthesis. The maximum data rate of the access point is 48kbit/s (on the application layer) and this bandwidth has to be shared by all Particles in range of the access point.

The iCup is composed of several components:

- Aperiodic sending. This component allows the Particle to send the data aperiodic, with the goal to reduce the energy consumption used in transmissions.
- Reminder. This component reminds the assisted person by using the built-in piezo beeper, when he or she should drink more to prevent dehydration. As this beeper is not very loud, the iCup normally sends the data to the application on the LAN. This application can then remind the assisted person far better by using speakers and speech synthesis.
- Energy saving. This component tries to reduce the energy consumption of the battery-powered Particle.
- Acknowledgment. This component enables reliable communication between the access point and the Particle.
- Sensor. This component does the actual measuring of fluid drunk by the assisted person.
- OTAP. OTAP stands for Over The Air Programming. This component allows remote reprogramming of the iCup using the wireless link.

All these components influence the way the iCup measures and sends data to the application on the LAN. Furthermore, each of the components can be switched on or off, which allows the iCup to use its resources (bandwidth, energy, processing power, etc.) as intelligently as possible.

2.5 MODELING ADAPTATION

This section analyses what knowledge must be available at runtime to describe online-determined adaptation. The next subsection decomposes adaptation into different concerns that need to be addressed for the modeling of adaptation. The subsequent subsections describe the adaptation process and mechanisms that are required for actually realizing an adaptation. As online-determined adaptation requires knowledge to determine online what impact an adaptation may have, the chapter ends with an analysis of that concern.

2.5.1 DECOMPOSITION OF ADAPTATION

Applications that provide online-determined reconfiguration need specific information for its ability to adapt and what influence that specific adaptation will have on the application. Therefore, the concept of adaptation is analyzed more thoroughly. Figure 2-4 separates adaptation in seven concerns. This figure is based on the analysis by Becker et al. in [11].

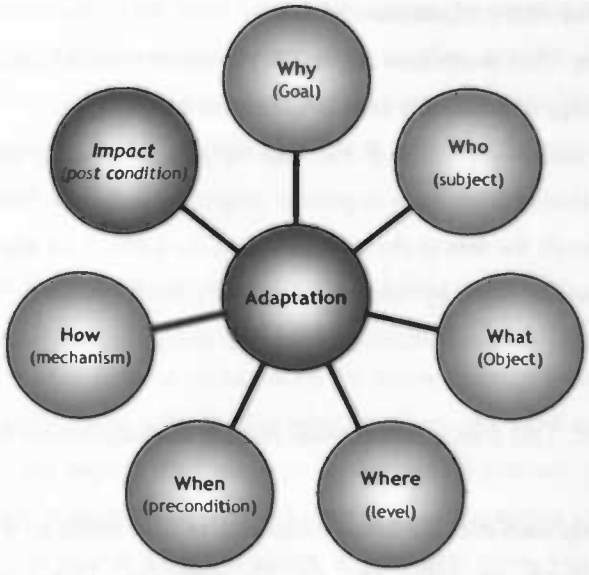


Figure 2-4: Information requirements for adaptation (based on [11]).

Why: addresses the goal(s) for adaptation. If there is no goal or driver for adaptation, the rest of the questions are irrelevant. Goals can be e.g. self-management or flexibility at runtime.

Who: defines who the subject of the adaptation is. This thesis has already distinguished between two subjects: the user of the software (e.g. system engineer or home user) and the software itself (making the software autonomous).

What: defines what object will be adapted. In general, the object will always be the behavior of the software. Specifically, the object will be the resources or the quality attributes of the software.

Where: addresses on which level the adaptation takes place. This can be e.g. on an architectural level, service level, component level, or attribute level.

When: defines the precondition for adaptation. Which requirements should be fulfilled in order to make an adaptation at all. Becker et al. distinguish in [11] between the different steps in the development process of an application, such as design, implementation, compilation, deployment or execution. Since this thesis addresses runtime adaptability, it will focus on the runtime preconditions.

How: defines which mechanism is used for the adaptation. E.g., select one or more alternatives or set a certain parameter.

Impact: defines the post condition of the adaptation, which means that the effect of an adaptation is defined. Impact has a different color and has been printed in italics in Figure 2-4, since knowing the post condition of an adaptation is not necessary for carrying out an adaptation. As the scenarios require the prediction of the effects of an adaptation, this information is in those cases necessary and is therefore considered as an important concern for (runtime) adaptation.

Not all of the seven concerns are necessary to define *adaptability*. The following diagram groups the different concerns together:

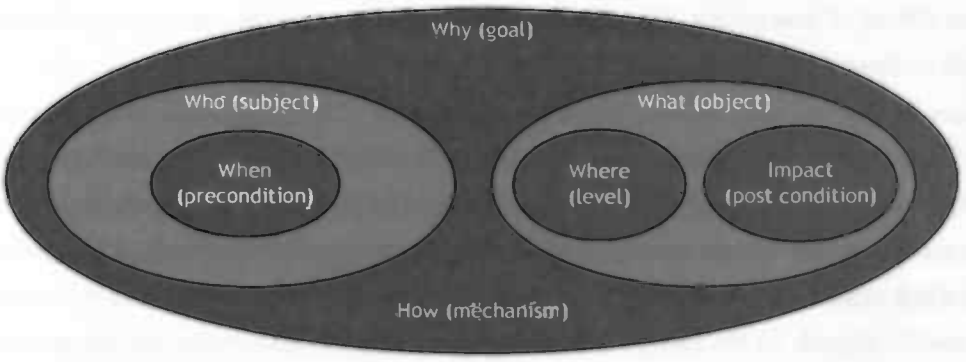


Figure 2-5: Grouping of adaptation concerns

The subject and object of adaptation share the *why* and the *how* concerns. For a goal such as self-management, the subject needs to *know* what can be managed and the object needs to *express* what can be managed. Furthermore, both the object and the subject need to have an understanding how to use and provide the adaptation mechanism. The *when* concern (precondition) is part of the reasoning of the user or software that wants to adapt the software.
The *where* concern is part of the object that provides the adaptability. Associated with the *where* concern is the *impact* or post condition of the object after it has been adapted.

Since the goal of this thesis is to model adaptability, the *who* and the *when* concerns are not addressed. Furthermore, the *why* concern is not taken into account, since that depends on the software application and is defined during design and implementation. This thesis assumes that the goals for adaptation are known. Examples of this concern are mentioned in previous chapters.
This leaves us with the concerns of *what*, *where*, *how*, and *impact* that need to be modeled for runtime adaptability.

2.5.2 ADAPTATION PROCESS

When a goal for adaptation is defined, such as 'optimize bandwidth usage', the *what* question is automatically answered. The object of adaptation is in this case the bandwidth usage. This means that a concept such as bandwidth usage must be available for the subject of the adaptation. In general, the object of adaptation is a resource (e.g. bandwidth, memory) or a quality attribute (e.g. utility, precision of data, reliability). Additionally, in order to know *where* to adapt, you need to know where you can influence the object of adaptation.

In an ideal world, if you know the object of adaptation, you also know where to do the adaptation, since there is a one-to-one correspondence between them. Take for example a car that you want to steer to the left (goal). You know that when you turn the steering-wheel (where) to the left, you have instant

effect (impact) on the orientation of the front wheels (object) of the car that consequently turn to the left. Such a simple connection between *where* and *what* is often not the case in software. There, multiple 'steering-wheels' can influence the same object (and others in parallel) and make 'turning left' far more difficult. Consequently, it is often only possible to influence the resource or quality attribute you want to change indirectly.

Figure 2-6 depicts the conceptual adaptation process. The *what* concern has an association with the software under adaptation (depicted by the dashed line). The goal of the adaptation is to change that what is represented by the *what* concern (depicted by the arrow pointing towards it). Therefore, you need to adapt at the locations in the software that have impact on the *what* concern. *Where* you will adapt exactly, depends on the *impact*. This needs to be determined just before the decision is made. This means that if you want to adapt the object, you need to trace the arrows in Figure 2-6 back to the subject to find out *where* and *how* to adapt the object. If you know *where* you need to adapt with the desired effect, you need to know *how* you can achieve this effect.

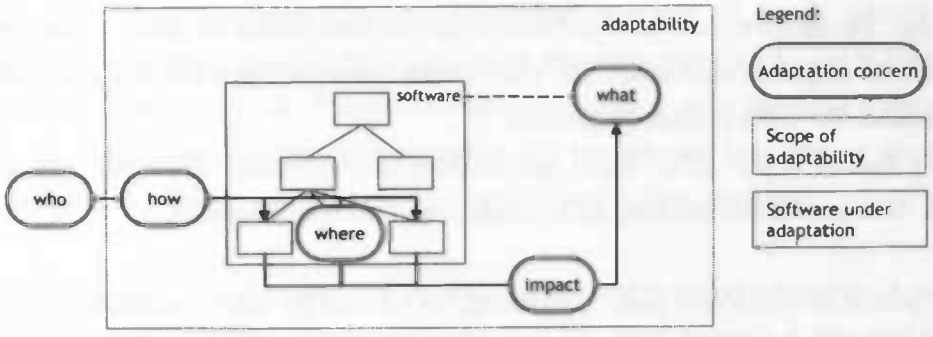


Figure 2-6: Conceptual adaptation process

There are different ways to define *how* you can adapt something. For example, the option dialog in Figure 1-3 (p. 8) describes several ways to adapt Microsoft Word at runtime. For the measurement units option, there is the possibility to select one out of five different metrics. The Recently used file list defines a selection from an integer range between 1 and 10 entries. The Mail as attachment option is visualized by a checkbox that enables the selection of a Boolean value of true or false. Consequently, selection is one of the mechanisms to express runtime adaptability.

As was mentioned in Section 1.2.2, the mechanisms needed for variability in SPLs express the adaptability needed to adapt to a specific context. Therefore, we can reuse these concepts for adaptation mechanisms.

2.5.3 ADAPTATION MECHANISMS

An SPL heavily relies on a generic asset repository that is shared between all the members (products) of the SPL [15, 31]. Since each member represents a different product used in a different context, the generic assets possess variability that enables reuse of the generic asset. This means that a generic asset

can be changed at certain locations to allow for different behavior, needed for the different products that are created from the generic asset repository. The locations "at which the variation will occur" [25] are defined as variation points and represent the actual location(s) in the generic asset *where* it can be changed.

A variation point is accompanied by a mechanism and a specification. The mechanism makes the adaptation possible and the specification describes the possible adaptations for that specific variation point. Several variability mechanisms exist, e.g. described by Muthig [31], Schmid and John [36] and van Gurp, Bosch and Svahnberg [44], but most are only applicable during development time (e.g. inheritance, overloading, or frame technology). Becker [10] describes the variability mechanisms on a more abstract level in such a way that they are also applicable at runtime:

1. Selection – allows for a choice from predefined variants. The size of the set of choices is fixed. This means that no new variants can be added, i.e. the variant points are closed. It supports the selection of zero or one variant (Boolean selection, also know as optional selection), or exactly one (selecting one alternative) or more than one (multiple selection) from a set of variants. Here, a variant represents a piece of functionality that exhibits certain behavior.
2. Generation – uses a generator to generate a specific variant, based on a specification language that describes how to generate the variant. Examples are macros or script languages that are interpreted at runtime.
3. Substitution – allows the replacement of a variation point by the actual value of the respective parameters. This allows for individual tailoring of the asset (e.g. adding user-defined code) for a specific context. Therefore, this is the most flexible mechanism. This means that the variation point is open, i.e. the set of variants can be extended. A special case of substitution is parameterization, which allows a parameter to be replaced by a new value.

Figure 2-7 describes these mechanisms formally in a diagram.

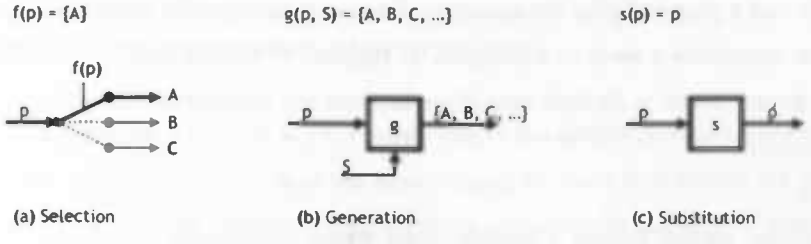


Figure 2-7: Representation of the three different variability mechanisms. Inbound arrows describe the parameters of the function; outbound arrows describe the output of the function.

Selection (a) can be seen as a function *f* that takes a parameter *p* and uses *p* to select from the predefined set of variants *{A, B, C}*. Generation (b) uses both the parameter *p* and a specification *S* for the generator *g* to generate a variant. The number of parameters can be extended to provide for a more

sophisticated generation. An example of generation is the Mozilla-based browsers (e.g. Firefox and Seamonkey) [30] that use an XML-based markup language (XUL) to describe the user interface of the browsers and uses a generator to generate the actual user interface on the user's screen. Substitution (c) uses the parameter *p* as the variant to use. *p* can be for instance a piece of compiled code that can be run. An example of substitution is the dynamic loading of libraries at runtime e.g. for loading the right decoder algorithm for decoding a movie.

The mechanisms shown in Figure 2-7 get from left to right more powerful, but also more complex. It is for example possible to substitute a substitution with a selection, which not only replaces the variation point provided by the substitution, but also creates a new variation point for the selection. These complex substitutions and their handling are out of the scope of this thesis.

The specification of variation points addresses the constraints on the variables that describe the behavior of each of the mechanisms. For selection, you have to describe the set of variants to choose from, and the function *f* that maps the input parameter *p* to one or more variants. For generation, this description is more difficult since it involves specification language (*S*) that is different in every application. This also means that describing it in a more detailed pattern than the one in Figure 2-7 (b) is not possible for the general case. The same is true for substitution, since parameter *p* can be used to substitute anything. Therefore, only parameterization will be addressed here, since it constraints the parameter to numerical values and is therefore far easier to specify than 'anything'.

The pattern community has also identified different mechanisms to realize adaptation. A pattern is a general repeatable solution to a commonly occurring problem. Each pattern is a three-part rule which expresses a relation between a certain context, a problem and a solution [2]. In this case, the context is adaptation. Avgeriou et al. provide in [5] an overview of adaptation patterns that address the problems of evolution, modifiability, reusability, and integrability in software. They describe the following patterns for adaptation realization:

- Microkernel.

  The microkernel pattern allows for the realization of a common architecture of a product family and a plug-and-play infrastructure for the system-specific components. A microkernel pattern introduces a level of indirection as requests or method calls are directed through the microkernel where is decided how these requests are handled by the variant system-specific components.
- Plugin.

  The plugin pattern defines a specific point where components committing to a predefined interface can be plugged in. This pattern allows for changing or adding components without the need to modify the existing application by rebuilding and redeployment.
- Reflection.

  In a reflection architecture, all structural and behavioral aspects of the system are stored into meta-objects and are separated from the application logic components. The architecture is

consequently divided in a meta level and a base level. The base level components can query the meta level for structural and behavioral information about the system. Vice versa can the meta level e.g. restructure the components of the base level. This allows for the coping of unforeseen situations.

- Interceptor.

  The interceptor pattern allows for the interception of business methods. Events are registered in dispatchers that invoke the interceptors when a specific event that they are interested in is raised. This allows for easy extension of application functionality without changing the core of the application. Similar to the microkernel pattern does this pattern also introduce a layer of indirection.

2.5.4 IMPACT OF ADAPTATION

The last concern left to address is the *impact* concern, which is used to decide which of the possible variants to use at a certain point in time. It is the relation between *where* (the variation points) to adapt and *what* effect (on quality attributes or resources) is wanted. As has been said, this relation is not one-to-one, and consequently complex. Figure 2-8 depicts this complexity with an example from the earlier introduced example in Section 2.4. It shows Particles that influence the resource 'Bandwidth' and quality attribute 'Accuracy' of the system. (UML stereotype notation is used to show distinction between the different types.)

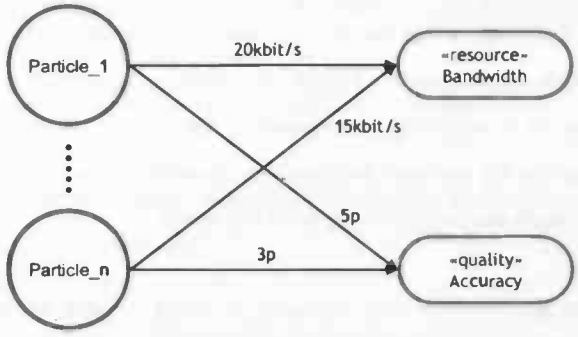


Figure 2-8: Influence of Particles on resources and quality attributes

In this (simplified) example, a Particle is a variation point in the system and can be enabled or disabled. Since the Particles gather information from the environment, the more information the system receives, the more accurate the system is. Particle_1 uses 20 kbit/s of bandwidth and adds 5 'points' of accuracy to the system when enabled (since accuracy is a qualitative attribute, it is difficult to measure. Therefore, a point system is used here, which is easier for computers to calculate). Particle_n has a similar effect, though a little less than Particle_1.

In the case of too much bandwidth usage, it has to be decided which of the particles should be disabled. Disabling Particle_1 can be overkill, resulting in a system with less accuracy than when only Particle_n would have been disabled. The following decisions have to be made:

Table 2-2: Decisions and their impact for the example in Figure 2-8.

Decision	Type	Impact		
Disable Particle_1	Option: yes/no	Yes:	Bandwidth – 20kbit/s,	Accuracy – 5p
		No:	Bandwidth – 0kbit/s,	Accuracy – 0p
Disable Particle_n	Option: yes/no	Yes:	Bandwidth – 15kbit,	Accuracy – 3p
		No:	Bandwidth – 0kbit/s,	Accuracy – 0p

2.5.5 DECISION MODEL

In SPLs similar decisions have to be made when a variability has to be resolved during product derivation. SPLs use a special document, the decision model, to collect these decisions and their effects [31, 36]. For instance, the decision model in KobrA [31] describes (i) a question that represent the variation point (what question needs to be answered in order to resolve the variation point), (ii) the type of variation point (selection, parameter), (iii) the possible variants (called a resolution set), and (iv) the effect on other decisions. The rationale for choosing one variant over another is often added to the decision model to help the developer choose the right variant [16]. Consequently, the decision model is a good starting point for designing a model that captures the adaptation knowledge required for runtime adaptation. The problem, however, is that the decision model in current product line approaches is in an informal form and requires a conversion to a formal form that can be used at runtime in such a way that it can also be interpreted by a computer. Additionally, the effects of decisions are expressed in a functional way, i.e. changes in the software architecture, component design, or other decisions, and not on more abstract concepts such as quality attributes or resources.

Now adaptation has been analyzed to find out what is really needed for runtime adaptability and similarities with SPLs are identified, it is time to analyze related work that might help to create a model for runtime adaptability.

3 RELATED WORK

Section 2.5 on Modeling Adaptation in the previous chapter identified several similarities with runtime adaptability and software product lines. This chapter will analyze what is done in the SPL domain that can be used for modeling runtime adaptability. Additionally, it will have a look at some approaches for modeling adaptation. Each of the approaches will be analyzed for the following aspects, analogous to the concerns for adaptation:

- Modeling of variability (where to adapt)
- Modeling of mechanisms (how to adapt)
- Modeling of impact (impact of adaptation)
- Modeling of resources and quality attributes (what to adapt)

Furthermore, the modeling of decisions is analyzed. Decisions make up the current system configuration and are needed to find out if an adaptation is necessary. Additionally each approach will be analyzed for applicability at runtime. Each of these aspects is important for the explicit specification of runtime adaptability.
Consequently, the following concerns are added to the analysis:

- Modeling of decisions (current system configuration)
- Applicability at runtime

The analysis is based on the following eight approaches:

- Feature models based on FODA (Feature-Oriented Domain Analysis) by Kang et al. [26]
- KobrA by Atkinson et al. [3] and Muthig [31]
- Adaptation Support in Software Product Families by Becker [10]
- Design Spaces by Geyer [24]
- COVAMOF (ConIPF Variability Modeling Framework) by Sinnema et al. [39]
- Koala by van Ommering et al. [45]
- Chameleon by Trapp [43]
- Madam (Mobility and Adaptation-enabling Middleware) by Floch et al. [22]

Many of the approaches use meta-models: models that describe model elements at one abstraction level higher than that of the model itself. These meta-models describe the attributes and the relations among the modeling elements and show the properties of these elements. Therefore, they show the information that is captured in a particular approach and allows us to compare those properties.

In the following eight subsections, each of these approaches is described. Section 3.9 gives evaluates how well the approaches support the mentioned aspects for runtime adaptability.

3.1 FEATURE MODELS

Software product line engineering requires the identification of commonalities and variabilities in the products developed by the SPL. A convenient method for this is a feature-oriented approach called FODA (Feature-Oriented Domain Analysis), introduced by the Software Engineering Institute in 1990, which uses features as elementary abstractions that both customers and developers can understand [26]. Features are used to group product requirements and describe a piece of behavior of the system. Since features can be defined as mandatory, optional or as an alternative for another feature, they can be used to describe which behavior should or should not be incorporated into the product. Therefore, features represent the "decision space" for product development [27]. An extensive overview on feature modeling can be found in [18, Chapter 4].

In order to describe the expressiveness of the variability of a feature, its meta-model is depicted in Figure 3-1 using UML.

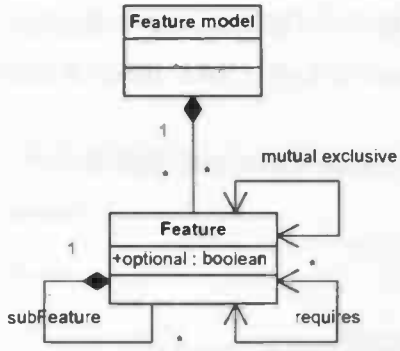


Figure 3-1: Feature meta-model (taken from [43])

A feature model consists of zero or more features. Each feature can have subFeatures, which allows for a hierarchical tree structure. Furthermore, a feature can have a requires relation with other features, which means that in order to let a certain feature function properly, it requires the availability of another feature. The mutual exclusive relation expresses the fact that two or more features cannot be combined together in a product, e.g. when they are incompatible with each other. These two relations are called 'feature interaction'. The boolean attribute optional defines whether a feature is optional.

FODA is combined with a convenient notation, which is used in Figure 3-2. It shows the feature tree of one of the sensors in the BelAmI-project, the iCup. The iCup – introduced in Section 2.4, to measure the drinking activity of a person – consists of several features that can be mandatory or optional. An optional feature uses a small open circle on top of the box of the feature to indicate that it is optional.

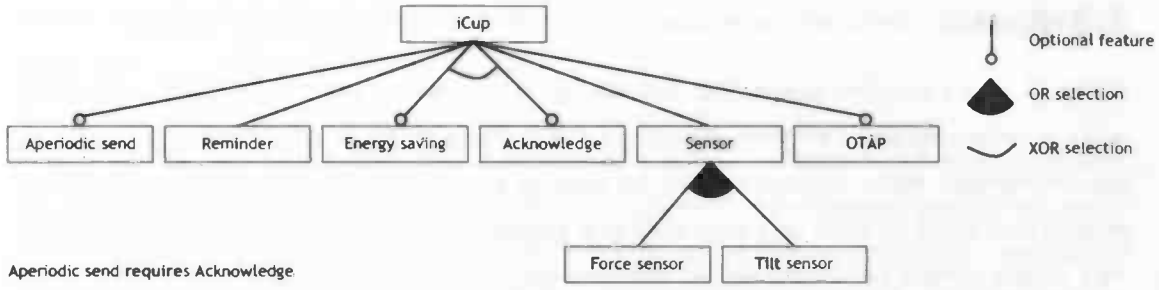


Figure 3-2: Feature model of an iCup used in the BelAmI project

The Aperiodic send feature is shown as optional, the Reminder feature is mandatory. The XOR selection makes sure that only one of the variants is selected, as seen in the relation between Energy saving and Acknowledge. The iCup can use different sensors to measure drinking activity. The OR selection (the filled quarter circle), describes that the iCup can use a Force sensor, a Tilt sensor, or both. Right now, the diagram shows a tree, due to the grouping of different features and since the requires constraint is not shown graphically (only in text). Adding these graphically and adding more constraints between different levels in the feature tree, the tree will become a feature graph and consequently more complex to read.

A feature model is accompanied by a feature configuration that describes which features are selected for a certain product.

Feature models are able to express different types of the selection mechanism:

- Optional: Boolean selection (yes/no)
- OR selection: Multiple selection (select one or more out of many)
- XOR selection: Single selection (select exactly one out of many), the different choices are mutually exclusive with each other.

Parameterization is not part of the feature modeling approach. Furthermore, feature models lack the functionality to express resources and quality attributes and the impact of decisions on them. The feature model can be seen as the decision model for a product and the feature configuration as the decisions that are made for a product.

Feature models are only used for modeling the variabilities and commonalities during the design phase of the product (line) and are consequently not available during runtime. In order to use feature models for runtime adaptability, it is required that there is a mapping between the model and the implementation of the features and the mechanisms that are used to express variability using a feature model. Furthermore, all variants that are available for a specific variation point, should be available at runtime and possess the ability to be chosen and run.

3.2 KOBRA

KobrA [3, 4] is a complete product line methodology to implement a software product line. It heavily relies on component-based software engineering, which allows for the practical application of product line development. Since components allow for reuse on a small scale (e.g. implementation level) and product lines allow for reuse on a large scale (e.g. product level), the combination of both makes it a very useable method for introducing and implementing a product line in a large variety of enterprise contexts.

KobrA uses a generic framework that comprises all the product variants (common parts and variant parts) that make up the family. A key artifact of KobrA is the KobrA component, 'Komponent' for short, the basic building block and is treated as an independent product. The generic framework is build up from Komponents and is organized as a tree. Therefore, a product is developed by a hierarchical composition of Komponents. Consequently, a complete product is a Komponent and a Komponent can be a complete product, which allows for intensive reuse of these artifacts on different levels of abstraction.

A Komponent is described on two levels of abstraction: specification and realization. See Figure 3-3.

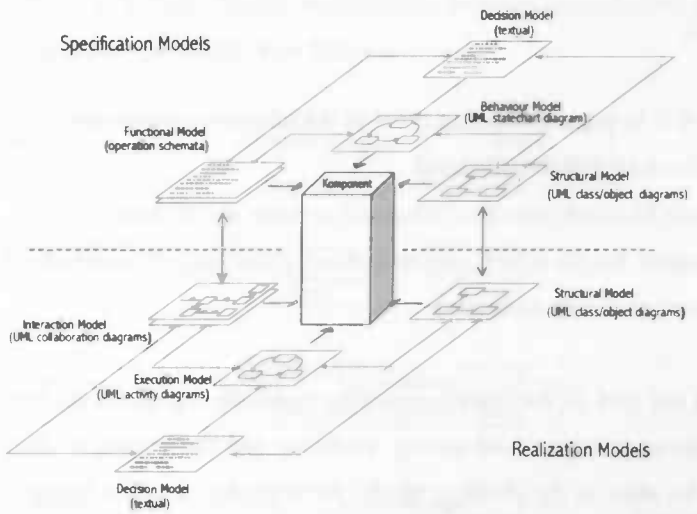


Figure 3-3: Komponent specification and realization (source: [3]).

The specification describes the external visible properties and behaviors and allows it to be used as a black box for other Komponents. It is the interface and the services offered by the Komponent to the outside world. The realization describes how the Komponent can realize that what is described by the specification in terms of interaction with lower Komponents in the tree.

The variation of each Komponent is captured in the decision model. The decision model describes, in textual form, how all the other models (structural models, interaction model, functional model, behavioral model and execution model) change when a specific design decision is made. When a

product is instantiated and the variability is resolved, the decisions are documented in the Resolution model.

Muthig [31] describes a lightweight product line approach that is a refinement of KobrA and enables companies to make an evolutionary transition from a single-systems development process to a multi-product process using software product lines. An important part of this work describes a meta-model for product line infrastructures that captures all the concepts used in his approach, including variation points, depicted in Figure 3-4.

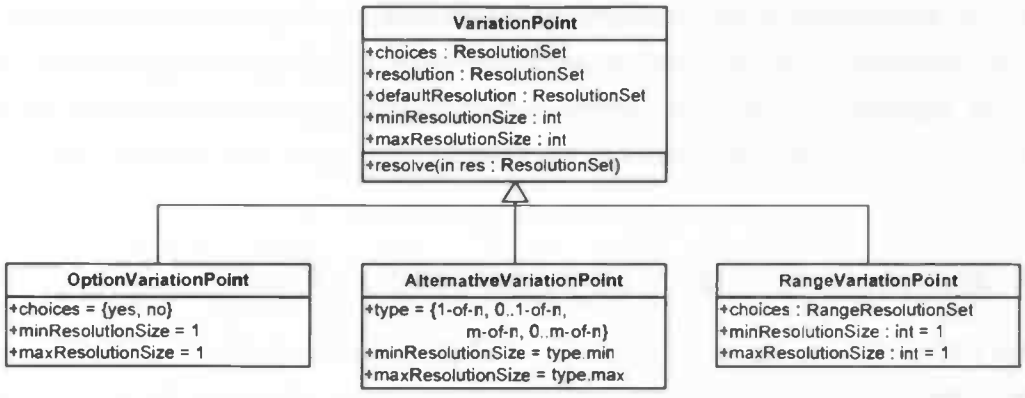


Figure 3-4: Meta-model of variation point types

A VariationPoint captures the general information for a variation point. It is specialized to capture specific selection mechanisms: optional selection, alternative selection and selection from a specific range. Every VariationPoint has choices to be chosen from, i.e. the variants that are stored in a ResolutionSet, which is a set of possible resolutions. When a variation point is resolved (using the resolve() operation) the chosen variant(s) are stored in the resolution attribute. A defaultResolution can be set to a typical choice for this variation point. The size of the resolution set is constrained by minResolutionSize and maxResolutionSize and their values are set for the specific specializations:

- OptionVariationPoint – allows for Boolean selection, the resolution size is always exactly one.
- AlternativeVariationPoint – constraints the resolution size from the generic VariationPoint in four different ways. It allows (i) single selection from many variants, (ii) optional single selection from many variants, (iii) multiple selection from many variants, and (iv) optional multiple selection from many variants.
- RangeVariationPoint – allows for continuous data types, such as numbers and characters. These values are stored in the choices attribute by a specialized resolution set that simulates the behavior of a finite set of variants.

A Decision extends a VariationPoint with an attribute question. This question (e.g. "does the particle need the energy saving functionality: yes/no?") needs to be answered in order to resolve the variability. The Decision Model is consequently composed of a linear list of Decisions.

Resolution constraints are used to specify that a certain decision constraints other variation points. This is for example the case when the Energy saving feature is chosen in the feature model in Figure 3-2. The Energy saving feature is mutually exclusive with the acknowledge feature (since the acknowledge feature requires extra energy to send and receive acknowledgment packets), which means that the selection of the Energy saving feature results automatically in a 'no' answer for the decision to select the acknowledge feature for the product. A ResolutionConstraint class associates a Decision with a VariationPoint in the meta-model. A logical expression is used to define these constraints.

The meta-model of Muthig is more expressive than the feature meta-model, since it allows for parameterization using the RangeVariationPoint. As with the features, this meta-model is also missing a relation with resources and quality attributes and the relation's impact and the ability to express them. Decisions are stored in the resolution attribute. As been said, the decision model is in a textual form and needs to be converted to a form that is usable at runtime.

3.3 ADAPTATION SUPPORT IN SOFTWARE PRODUCT FAMILIES

Becker [10] describes an approach that focuses on adaptation support in SPLs. It also uses a meta-model to describe all the information that is necessary to increase the adaptability of a software product line. As with Komponents in KobrA, Becker distinguishes between specification and realization of a variability. The specification of variability is as follows:

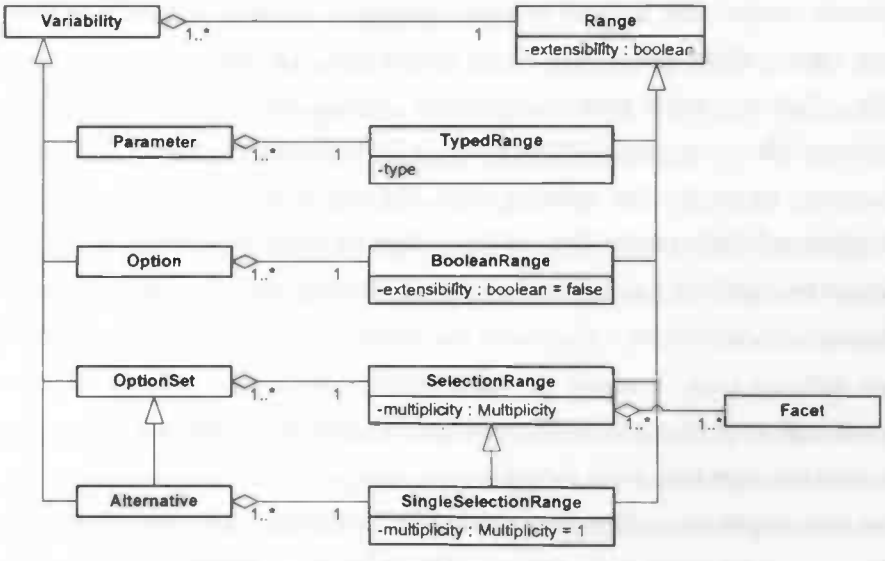


Figure 3-5: Meta-model for the specification of variability types

Figure 3-5 shows the concept of Variability, which is specialized in four types: Parameter, Option, OptionSet and Alternative. The Range quantifies the possible variants and is also specialized for each of the variability types. The extensibility attribute is used to define open ranges, i.e. it allows for the addition of variants later in the development process.

When the variability type is defined as a Parameter, its range is specialized to define the data type of the parameter that is used (e.g. integer, or floating-point numbers). An Option exhibits the same behavior as the OptionVariationPoint in the previous section, it allows for optional selection using a boolean range: yes/no. An OptionSet allows for multiple selection (set size defined by the multiplicity attribute) from a SelectionRange. The Alternative is a specialization from OptionSet and only allows for a single selection from a specific range (therefore is the multiplicity attribute defined as '1'). Facets stand for the possible variants that can be selected in a (Single)SelectionRange.

The similarities between Becker and Muthig are apparent, but there are some differences. Becker generalizes the Range of a variability more than Muthig, while Muthig combines the Selection- and SingleSelectionRange in one meta-model class. Furthermore, Becker makes a distinction between variability (specification) and a variation point (realization of the variability) (see Figure 3-6), and distinguishes variability types, while Muthig distinguishes variation point types.

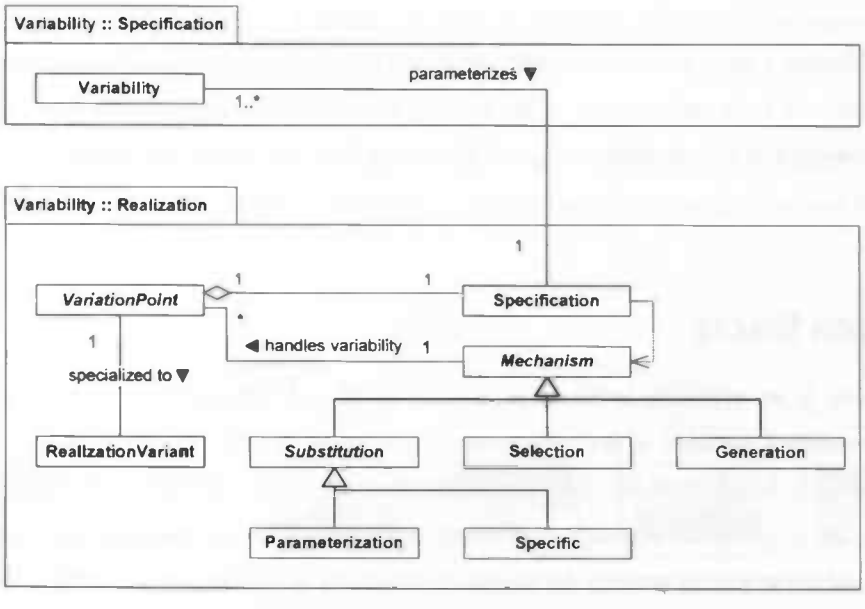


Figure 3-6: Specification and realization of variability

Figure 3-6 shows that Variability parameterizes the specification for a VariationPoint, i.e. it defines the specification by using parameters described in Figure 3-5. The specification has a dependency on the mechanism that handles (realizes) the variability on the implementation level. The mechanisms are specialized in different variability mechanisms that were described in Section 2.5.3. Becker describes patterns for each of the mechanisms to support the realization of the mechanisms.

To bind variants to their variability, Becker uses a binding assignment that is stored in a profile. This profile is similar to Muthig's resolution model and describes the rationale for a bound decision. The binding between variant and variability also supports the notion of binding time, which allows for a more fine grained variability specification (e.g. it can be specified that a variability must be resolved before link time).

Becker's meta-model allows for the specification of binding constraints, similar to Muthig's Resolution constraints. Binding constraints define constraints that one variation point binding has on other variation point bindings. They allow three different kinds of constraints:

1. Range tests, to check if a certain variant is chosen or if the variant lies in a certain range. E.g. `EnergyConsumption < 60 %`.
2. Binding time tests, check if certain variation points are already bound.
3. Logical combinations of the first two, such as `EnergyConsumption < 60 AND EnergySavingFeature.selected == true`.

Furthermore, the model makes a distinction between static variation points and dynamic ones. Static variation points are bound during development time, whereas dynamic variation points are bound at development time but allow for a reselection of a variant at runtime when the adaptation mechanism supports it.

The goal of Becker's meta-model is to make variability management in SPLs more explicit and easier. Although it allows for a specification of the binding time of variability, its focus is on development time management and not on runtime support for variability. The model is therefore not available at runtime.

3.4 DESIGN SPACES

Design Spaces is an approach to assist component-based and reuse-oriented system development. Generic components are used to leverage reuse by designing them to be easily customizable to the specific setting in which it should be used, without manual intervention [7]. To this end, generic components are provided with generic parameters that allow for both functional and non-functional adjustment and consequently provide the necessary variability for those generic components. When all values for those parameters are available, a specific component can be generated from a generic component.

Generic parameters can be classified in three different categories: selection parameters, generative parameters or code parameters. These categories are identical to the variability mechanisms described in Section 2.5.3. Selection parameters allow for choosing from a finite set of pre-build possibilities. Generative parameters are used to configure tools that enable the generation of a specific part of a component and code parameters replace the parameter by user-supplied code.

A Design Space is a multidimensional space of design choices in which each dimension describes a specific design decision. The possible alternatives (or variants) for each dimension are called 'categories'. To express positive and negative impact from one category on another one, or on a complete dimension, weighted correlations can be defined in a range from -1 to +1. A weighted

correlation of -1 between two categories means that those two categories cannot exist together, and a weighted correlation of +1 means that those two categories must exist together. Therefore, correlations indicate good and bad combinations of choices and define design rules that guide an engineer to make the right decisions for creating a working system.

The original Design Space concept described in [37] distinguishes between two subspaces: requirement and structural. The requirement subspace addresses the externally observable behavior, and is similar to the specification level of KobrA. The structural subspace addresses the internal structure and implementation issues and is similar to KobrA's realization level. Correlations between these two subspaces can be used to express interdependencies between specification and realization and bridge the gap between these two levels of abstraction.

Geyer et al. [7, 24] extended the original Design Space concept to, among others, allow for continuous dimensions (allowing floating-point and integer values) and grouping of categories. For continuous dimensions, the correlation concept has been extended, allowing for the combination of several categories in boolean expressions and formulas.

These extensions result in the fact that Design Spaces are more expressive than Feature models and allow for a more fine-grained variability modeling. Figure 3-7 depicts the Design Space of the iCup.

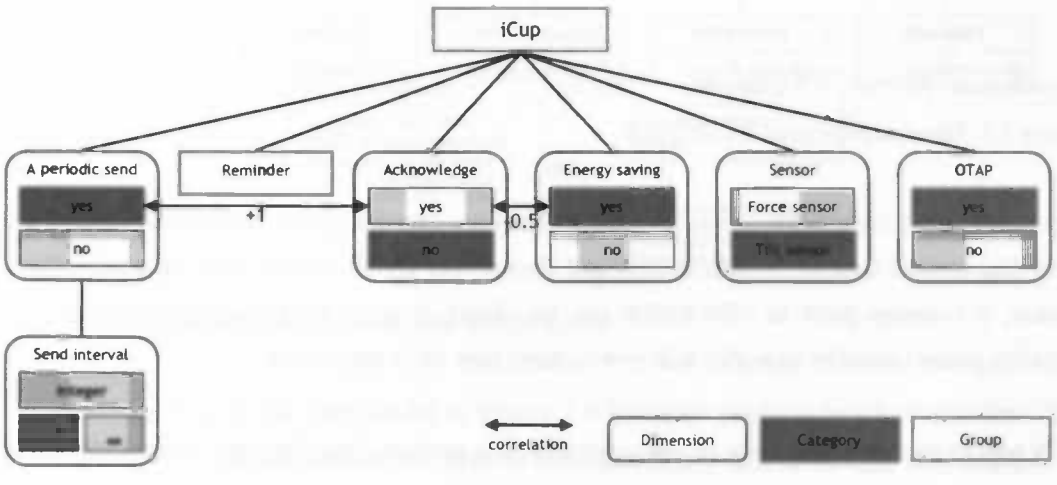


Figure 3-7: Design Space of the iCup using the notation from [24]

Groups can be seen as mandatory features that are always available in the product. Every dimension has several categories that can be chosen from. The drawing notation used in Figure 3-7 does not distinguish between single selection (alternative) and multiple selection. It is possible to use correlations between alternatives, but when a larger set of alternatives is required, such an approach reduces the clarity of the diagram.

The figure also shows the ability to express a range of values in the dimension of Send interval. It shows an integer range from 1 to infinity.

Correlations are convenient since they allow a fine-grained way to constrain specific configurations. The correlation between the inclusion of Acknowledge and Energy saving for example shows that they can be used together, although not that easy. The correlation between Aperiodic send and Acknowledge shows that they should be used together. Since generic components provide generic parameters that allow for both functional and non-functional adjustment, design spaces allow for the modeling of non-functional requirements and consequently quality attributes and resources.

3.5 COVAMOF

COVAMOF (ConIPF (Configuration of Industrial Product Families) Variability Modeling Framework) is a framework that supports the modeling of variability in industrial software product lines and is developed by Sinnema et al.[38]. Variability in COVAMOF is defined in three layers of abstraction: features, architecture and component implementations. Variation points can be identified in each of these layers, and can be categorized in five basic types, as shown in Figure 3-8 (the same categorization is also defined in [36]).

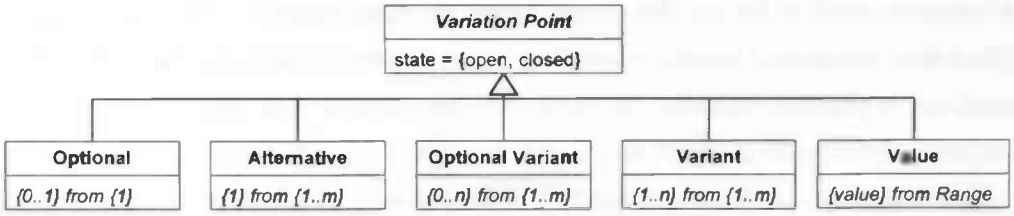


Figure 3-8: Variation point types in COVAMOF.

The figure shows the possible ranges for each of the types. The distinction in five basic types is a little more fine-grained than those from Muthig and Becker, but allows for the same variability. As with Becker, a variation point in COVAMOF can be closed or open using the state attribute; closed variation points cannot be extended with new variants later on in the process.

COVAMOF provides two views on the variability of a product. The first view is the variation point view that shows all the variation points on the three abstraction layers and consequently shows the variability of the complete product line. The second view is used to view the dependencies that are associated with one or more variation points and the constraints that are imposed upon those dependencies. This view restricts the variability and consequently the possible configurations that can be made from the variability defined in the variation point view.

Goal of the dependency view is to have more control on the almost unmanageable amount of variation points and their dependencies during the derivation process of products in large software product lines. Sinnema et al. identified that the industry needs a lot of (expensive) expert knowledge during the derivation process, since the dependencies and interaction between dependencies are so complex that

engineers need that knowledge in order to make a right decision for a variation point. Therefore, COVAMOF models dependencies first-class in their approach [39] and supports the engineer with this complex task.

The knowledge required during the derivation process can be split up into three different types:

- Tacit – contains information that is only available in the minds of experts. An example of tacit knowledge is that one variation point is dependent on another one, but you do not know how exactly.
- Documented – contains information that is expressed in informal models and descriptions, such as 'the energy saving feature reduces the energy consumption on a particle'.
- Formalized – contains information that is documented in a formal way that can be interpreted by a computer, such as [energy_consumption = 1.0 - [energy_saving_feature_enabled * 0.4].

This is an interesting view on the knowledge required for decision making, especially for adaptive systems, where decision knowledge must be computable in order to reason about an adaptation. Consequently, adaptive systems can only use formalized knowledge, but adaptable systems where an administrator issues the adaptations can use all three types of knowledge.
According to COVAMOF, a dependency represents a system property and specifies what the impact on the system property is when variation points are bound. This means that a system property can express a resource or a quality attribute.

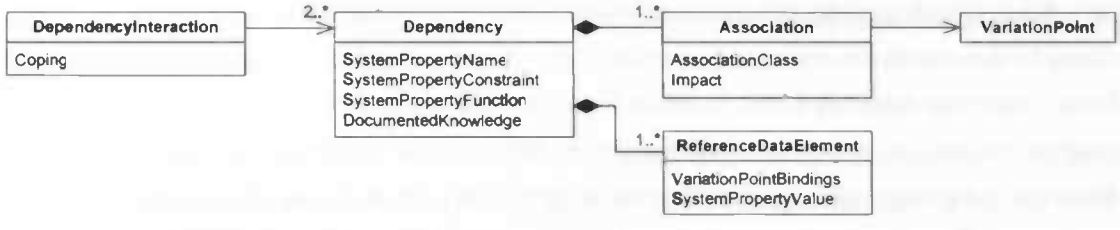


Figure 3-9: Meta-model for defining dependencies in COVAMOF

The Dependency class in the meta-model in Figure 3-9 has three relations and four attributes. The SystemPropertyName attribute defines the name of the dependency, such as 'Energy Consumption' or 'CPU Utilization'.

The SystemPropertyConstraint holds the constraint for the dependency, such as EnergyConsumption ≤ [MaxEnergyConsumption]. The constraint can hold parameters, in this case MaxEnergyConsumption, which takes a real value later on in the process of derivation. The value of the SystemProperty is calculated using the function defined in the SystemPropertyFunction attribute in the case that there is formal knowledge available. The function is then composed of the impact values of the associated variation points that are defined in the Association class. A Dependency can be influenced by one or more associated variation points and each Association can represent one of the three knowledge types described above (defined by the AssociationClass attribute in the Association class).

DocumentedKnowledge is the last attribute in the Dependency class and is used for describing where relevant documentation can be found for this dependency. The engineers can use this textual information for decision making during the derivation of a product. This attribute is used in case of tacit or documented knowledge dependencies.

The relation with ReferenceDataElements is used for gathering test data that will help engineers to determine the values of the system properties. The bindings for that configuration are stored in the attribute VariationPointBindings and the value is stored in the SystemPropertyValue.

The Coping attribute in the DependencyInteraction class is used to describe how to cope with dependencies that interact with each other. When e.g. the Acknowledge feature is selected for the iCup in Figure 3-2, it will on the one hand influence the (quality attribute) reliability of the wireless communication positively. On the other hand it requires more wireless communication, and will consequently increase the (resource) energy consumption of the iCup. The Coping attribute therefore adds explanation or describes best-practices for dealing with these interactions.

3.6 KOALA

The Koala approach deals with diversity and complexity of embedded software in the consumer electronics domain by the use and reuse of software components that work with an explicit architecture [45]. Koala is both a model and a component-based architectural description language (ADL) which allows for the explicit description and structure of a product's configuration in terms of its components. Every component describes which interfaces it requires and which it provides. Koala is then used to combine the required interfaces of components with the interfaces provided by other components. When one component requires an interface that is partially or completely implemented by two or more components, Koala can use a so called *module* to connect these components interfaces to a single interface using different implementation mechanisms, such as pre-processor directives (#ifdefs), but also runtime C-code, allowing dynamic (re)connections. This allows for compositional variability and adaptation.

Parameterized adaptation is made possible by the use of *diversity interfaces*, which are the variation points in Koala. These interfaces require properties to be set by other components or modules. Figure 3-10 depicts an example (taken from [45], that describes the composition of components of a TV-set) and shows how a diversity interface works.

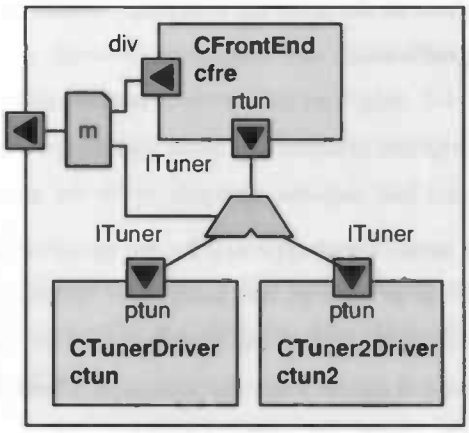


Figure 3-10: Diversity interface in Koala

Provided and required interfaces are depicted as a triangle in a square. Triangles pointing inwards are provided interfaces, required interfaces point outwards. The figure shows three components: one front-end component and two tuner drivers, both providing a ptun interface. The front-end can only use one of the tuners, and which one will be used depends on the settings provided by the diversity interface div. To select one of the tuners, the topped off pyramid in the middle of the figure can be used as a switch which is operated by module m and connects the right interfaces with each other. The behavior of the front-end component can be changed by the diversity interface div and will consequently select the appropriate tuner using module m. Module m has also a diversity interface for the whole super-component in Figure 3-10 that abstracts from the details about how the parameter actually influences the wiring of the sub-components. This allows for the development of independent components in the same way KobrA also tries to achieve.

Depending on the position in the software lifecycle, Koala uses different mechanisms to create a working component. When the value for the parameters in the diversity interface is known before runtime, the components (i.e. their interfaces) can be hard-wired and unused components can be removed from configuration. Switches are in that case also unnecessary, as with the module that operates the switch. Koala can then optimize the code, by removing the unnecessary parts.
When the information for the parameters is not yet available, the switch and module will be converted to runtime executable code that can reconnect the interfaces at runtime.

As a result, Koala allows for runtime adaptation, but only supports adaptations that are predetermined by the engineer on forehand. Furthermore, it allows for functional-based composition using interfaces, not quality-based composition, and consequently quality attributes or resources are not modeled in this approach.

Koala is focused on component-based reuse and the automation of component composition for products, but not on online-determined reconfiguration of those components. Koala leverages

flexibility during the design phase of the software lifecycle, runtime adaptability requires flexibility during the runtime phase of the software.

3.7 CHAMELEON

Chameleon by Trapp [43] is a model-based approach for the specification of adaptation behavior for embedded systems at runtime. It is used in the automotive domain to cope with the enormous complexity of dynamic adaptation in critical car software such as the ESP (Electronic Stability Program) system that can be found in almost every car nowadays. Chameleon is based on three models: a feature model, an environmental model, and a service model.

The feature model is used to capture the requirements of a product in a feature configuration. Since standard feature models can not be used to describe adaptation behavior, Chameleon extended the feature model defined by FODA [26] (see Section 3.1) to allow for the specification of dynamic feature configurations. This is necessary because the feature configuration changes when it is adapted during runtime.

The feature model in Figure 3-2 shows for example two sensors: a tilt sensor for the detection of the angle of the cup with the ground, and a force sensor for weighing the cup's contents at rest. Both sensors can be used to measure drinking activity, although the force sensor can do this with more precision than the tilt sensor. The feature model allows for the selection of both sensors in the static configuration, but only one can be used at runtime. The selection of both sensors allows for graceful degradation in case the force sensor fails, but there is no way to define that only one can be used at runtime. Therefore, the feature meta-model needs to be extended to allow for the selection of features in the dynamic feature configuration.

Figure 3-11 depicts the extended meta-model for features in Chameleon.

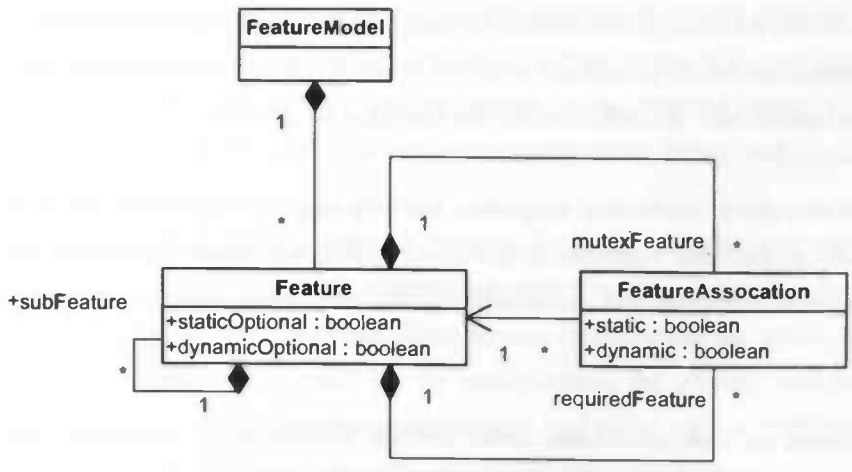


Figure 3-11: Feature meta-model in Chameleon

The optional attribute of the feature meta-class has been changed into staticOptional and dynamicOptional compared to the feature meta-model in Figure 3-1. This allows for the optional selection in both static configuration (design time) and dynamic configuration (runtime). The sensors in the previous example would be set to be dynamic optional and not static optional (which is the equivalent to static mandatory).
The associations mutual exclusive and required of the original feature model have been replaced by an association class with two attributes: static and dynamic. This allows for the specification of static and dynamic relations between features.

The extended feature model provides a means to select which features should be included in the product (i.e. the requirements for the product) using the static feature configuration and additionally define the requirements on the dynamic system configuration (i.e. the adaptation behavior). The biggest benefit of this approach is that all the dynamic system configurations can be checked for inconsistencies during design.

The environmental model uses variables to describe formalized domain knowledge needed for defining adaptation behavior. Since the environment changes, these variables change too. The changing of those variables is subsequently the source for the need of adaptation and consequently adaptation variants. Since each variant that realizes a value for the variable has different characteristics, pros and cons, it has a different effect on the quality of that variable. Therefore, Chameleon uses variable-types, a hybrid of data type, value, quality and semantics, to define a variable that describes its own quality at the same time.
Additionally the environmental model describes how the quality of a variable is determined from other variables. This allows for the analysis of the quality flow in a system, the quality of the adaptations a system can support, and consequently an assessment of the quality after adaptation.

Each variable-type defines variable-modes that are the variants, which realize the variable value. An example for the measurement of drinking activity by the iCup is shown in Figure 3-12.

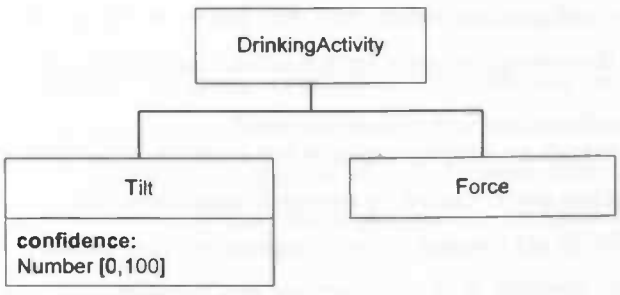


Figure 3-12: Variable-type and variable-modes and their quality in Chameleon

DrinkingActivity is the variable-type that supports two different qualities, depicted by the variable-modes Tilt and Force. It shows that there are two variants that can be used to provide a value for

drinking activity: by measuring using the tilt sensor or using the force sensor. Each variable-mode is accompanied with a semantic description (not shown in the figure) about the differences with the other variable-modes. Since Chameleon addresses predetermined reconfiguration, this textual description is sufficient because the developers define the actual adaptations. The variable-modes can be seen as a qualitative description of the quality of a variable-type, but can also use a quantitative description for more detailed information.

The Tilt variable-mode for example shows that it has a variable-quality confidence that defines how confident the sensor is about the fact that the tilt described a drinking motion. This quality can be used to define whether there was really a drinking activity. A variable-quality is associated with a range, that defines the possible values a variable-quality can have. In Figure 3-12 this is depicted by a number between 0 and 100 for the variable-quality confidence.

The use of variable-modes has several advantages. Variable-modes are more expressive than a definition of quality attributes, especially when formalized quantitative information is not available (see also COVAMOF). Furthermore, the use of variable-modes is more fine-grained and less complex in terms of mappings from (low-level) variables on predefined quality attributes.

The feature model defines not only the static and dynamic configurations, but also which variables are required and in which quality they are required when realizing that feature. The environmental model defines how and if the required variables for a feature can be provided in order to realize that feature.

The service model describes functional units of execution. Each service has one or more service configurations that can be seen as behavioral variants of that service. A feature is linked to a service configuration that realizes that feature. Every service configuration has a guard that describes whether that configuration (i.e. variant) can be activated. These guards are boolean expressions that are based on the variables defined in the environmental model. Every service has a local interface that is used to pass messages between services. This local interface is defined in terms of variable-types. This allows a service to be very detailed about the quality of the required variables. It can for instance define that only a certain variable-mode or variable-quality is required for that service or service configuration. Furthermore, a service configuration defines how the quality of the provided variables is affected, based on the quality of the required variables for that service configuration if that configuration is used.

When a certain variable-mode or variable-quality is not available in the environmental model, service configurations in the service model can not be activated, since their boolean guards that depend on that variable-mode or -quality do not evaluate to true. These service configurations are therefore invalid. In order to assist for the selection of a valid service configuration, the service configurations are prioritized by the developer to define favorable configurations, and added with an "Off" configuration that allows for disabling the service when none of the configurations can be activated.

Additionally, Chameleon allows for system wide constraints on the adaptation behavior of groups of services, since local optimizations are not always beneficial for the system behavior itself. This constraints the possible service configurations and reduces the complexity and increases the predictability.

Chameleon defines a very sophisticated, but also quite complex way for adaptation. This is mainly due to the fact that Chameleon is developed for the automotive domain where every possible adaptation has to be validated on forehand, since invalid adaptations can have severe consequences. Consequently, this adaptation modeling approach is focused on predetermined reconfiguration and the simulation and validation of each configuration.

Variability is specified in three different models that are connected to each other: the feature model uses optional features, the environmental model uses variable-modes for variable-types, and the service model uses service configurations. Selection of these configurations is the main mechanism that enables adaptation in Chameleon.

The modeling of impact of adaptation is implicitly defined in the different configurations of a service and the impact itself is local, i.e. it only concerns the service itself. The decision to make that adaptation is therefore predefined. Additionally, there is no knowledge on how it will affect the rest of the system.

Chameleon variable-types provide a very sophisticated way to define resources and quality attributes, since it supports both qualitative and quantitative specification.

All the models defined by Chameleon have the goal to validate and simulate adaptation behavior of product configurations. All these processes take place before the system is executed to create a valid system. When a valid system is created, all modeling elements will be removed and a static system is executed with predefined configurations and decisions for adaptation.

3.8 MADAM

Madam (Mobility and adaptation-enabling middleware) [22] focuses on adaptive application development for mobile computing using architectural models. Madam describes a complete adaptation framework that allows for the separation of application and adaptation behavior for leveraging self-adaptation at runtime. It exists of three models: the context model, the framework architecture model and the instance architecture model.

The context model describes the elements that represent the operating environment in which the application runs and is similar to Chameleon's environmental model. The context model is handled by an overall context manager that captures the different context variables.

The framework architecture model describes the complete architecture of the application. This model is built when the application is launched and is the runtime representation of the whole application, including all its variants at runtime. The model supports two types of variability mechanisms: component variability (selection) and parameterization. Parameterization allows for fine-grained adaptation by changing parameters of components. Component variability uses component types to define variation points. Component implementations that implement such a component type can be considered as the variants for that component type.

Component implementations are distinguished by properties that are attached to an implementation. Properties can be used to specify requirements on resources and quality attributes and are used to map those requirements on the elements of the context model. When a resource (e.g. network bandwidth) in the context model decreases, the requirements specified by the properties of a component implementation might not be fulfilled anymore and an adaptation is required.

The qualities of the provided services of a component implementation are also specified by properties. In order to distinguish between the different services provided, ports are used to specify different collaboration possibilities. Each port specifies in turn the associated properties of that collaboration.
Not all the values of the properties can be specified as constant (e.g. code size) and therefore every implementation is associated with property predictor functions, which can be used to specify more sophisticated functions to predict a property's value (e.g. a composition based on sub-component implementation properties).

The instance architecture model describes the current running instance of the framework model in which all the variability has been resolved. When a variation point has several variants and a decision has to be made which variant fits best in the current context, the adaptation manager supervising all adaptations, uses the utility of a variant. The utility of a variant is a scalar function that is defined by the weighted sum of all the properties. The weighing is used to specify which of the properties is more important for that particular variant and prioritizes the properties. The architect initially specifies the utility. The user can then refine the weights of the properties to his liking.

3.9 ANALYSIS

This section will compare the eight approaches that have been discussed on their properties to specify variability, their applicability at runtime, the use of different mechanisms, how the impact is modeled (if at all) and how resources or quality attributes are specified. Not all approaches can be compared on all these aspects, since each approach tries to achieve a certain goal.

Table 3-1: Overview of the approaches (first row) and their support for the six requirements (first column) for runtime adaptability. A '–' means the requirement is not supported, and a '+' means that the requirement is supported by the approach.

	Feature models	KobrA/ Muthig	Becker	Design Spaces	COVA-MOF	Koala	Chame-leon	Madam
Variability	-	+	+	+	+	+	+	+
Mechanisms	-	-	+	-	-	+	-	+
Impact	-	-	-	+	+	-	-	+
Resources & QA	-	-	-	+	+	-	+	+
Decisions	+	+	+	+	+	+	+	+
Applicability at runtime	-	-	-	-	-	-	-	+

Table 3-1 shows an overview of all the analyzed approaches and their support for the six requirements for runtime adaptability, as defined at the beginning of this chapter.

3.9.1 MODELING OF VARIABILITY

All the approaches model variability by using selection or parameterization, except Feature models. In Feature models such as FODA, only selection is possible (and therefore gets a '–' in the table). Furthermore, Feature models only allow for the modeling of variability and decisions (in the feature configuration), the rest of the modeling requirements are not supported by standard feature models.

The variability model allows for the specification of variation points, i.e. the points where the software can be adapted. All approaches support different types of selection, such as single selection, multiple selection, or optional selection. Becker and KobrA have a very distinct separation between specification and realization. Chameleon distinguishes between static variation points and dynamic variation points in feature models. The environmental and service models also allow for the specification of variability. Koala does not have a meta-model for the specification of variability, it is specified by modules and switches. The variability of Madam is specified in terms of component types and parameterization. Design spaces are similar to feature models, although they do support parameterization. COVAMOF allows for the specification of variability on the feature level, architectural level and component level.

3.9.2 MODELING OF MECHANISMS

Only Becker and Koala provide some guidelines for the realization of the variability. Becker describes several patterns and Koala allows for different realizations depending on binding time. For runtime realization Koala can use predefined code for adaptation. Madam supports selection of components adhering to a specific component type and parameterization as mechanisms to support runtime adaptation.

3.9.3 MODELING OF IMPACT

Impact can be modeled in several approaches, although mostly informal, using a decision model. COVAMOF, Design Spaces and Madam support the modeling of non-functional impact, needed for modeling runtime adaptability. COVAMOF describes a simple meta-model that describes the impact on system properties. Design Spaces use correlations between categories to define the impact. Madam uses properties to define the impact on system properties.

3.9.4 MODELING OF RESOURCES AND QUALITY ATTRIBUTES

Not all approaches support the modeling of resources and quality attributes. This is mainly because most approaches are only used at development time, where experts (should) know what the impact on the qualities will be. This approach cannot be used for the modeling of runtime adaptability. The approaches that support resources and quality attributes are Design Spaces, Madam, COVAMOF and Chameleon.

Design spaces allow for the specifications of correlations between different categories. In Madam, qualities and resources are part of the context model. COVAMOF explicitly defines dependencies that describe the system's qualities. Chameleon does local optimization by the use of qualities that are attached to the required and provided variables of a service.

3.9.5 MODELING OF DECISIONS

Feature models and Design spaces use a feature configuration that describes which features will be in the final product. In KobrA, the decision and resolution model are used to specify the decision and the resolution of that decision. Becker calls this a 'Profile'. Often some rational is attached to a decision to support the decision maker, but no formal distinction is made between the variants for a decision. In the case of COVAMOF, decisions are stored in a Product and decisions are supported by the functional and non-functional impact specification. More formal distinctions are made by Chameleon and Madam. Chameleon uses guards for each variant and some prioritization of the variants. Madam uses the utility of each variant to find out which decision is better at a certain point in time. As Madam is the only one that uses runtime models, the current configuration is stored in the instance architectural model.

3.9.6 APPLICABILITY AT RUNTIME

The overall winner seems Madam, since it is the only approach that uses models at runtime to support adaptability. Unfortunately, Madam's models focus on an adaptive distributed middleware layer for mobile computing that allows adaptation at runtime. It has therefore not a focus on SPLs compared to the other approaches.

The rest of the approaches do not have or lose their variability model at compilation time, just before the system is executed. Nonetheless, several meta-models of these approaches are useful for the specification of adaptability at runtime.

3.10 NEW APPROACH

From all approaches described in this chapter, none can be used for the goal of this thesis, due to the applicability at development time only or a non-SPL approach. Therefore, a new approach will be developed to assist the creation of runtime adaptable systems. This approach will be partially based on the analysis of this chapter, as it identified interesting properties that can be reused.

Most approaches use Feature models at the highest level of abstraction. This holds for KobrA, Becker, COVAMOF, Design Spaces (as it is similar to Feature models), and Chameleon. Feature models seem therefore a very good starting point for defining runtime adaptability.

To foster reuse, most approaches are component-based. This requires the building blocks of the software to conform to specific guidelines to allow composition. As adaptability requires runtime composition of components (i.e. the variants for a variation point), the process of creating a runtime adaptable system must be component-based too.

The next chapter will address the requirements of a new approach.

4 REQUIREMENTS FOR RUNTIME ADAPTABILITY

This chapter defines the requirements for a runtime adaptability model and the process of creating a runtime adaptable system. These requirements are based on the scenarios from Section 2.3. It will first introduce the domain model that will describe the entities and concepts used later in this thesis. Section 4.2 will focus on the stakeholders, functional requirements, and quality attributes of the runtime adaptable system.

4.1 DOMAIN MODEL

The domain model for a runtime adaptable product is depicted in Figure 4-1.

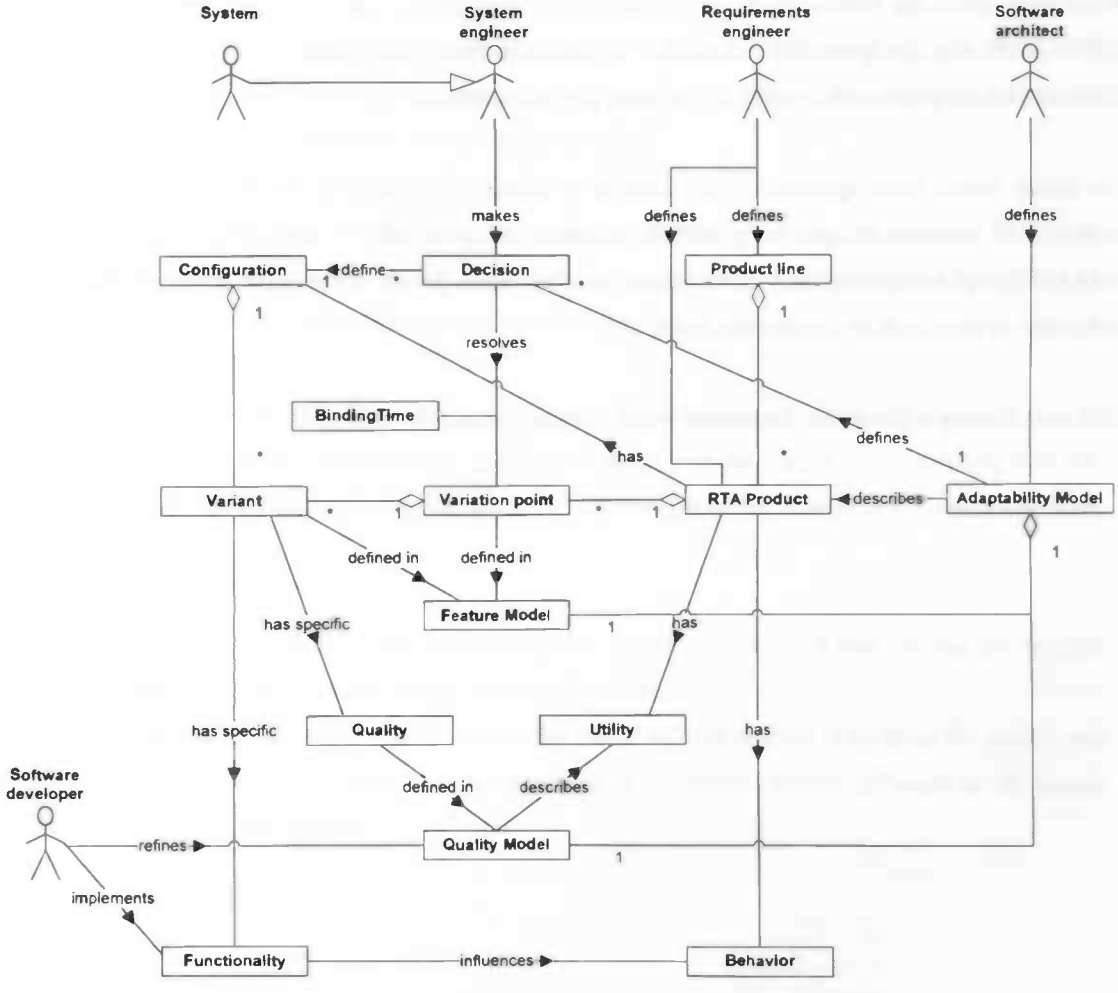


Figure 4-1: Domain model of a runtime adaptable system, focusing on the variable parts of the system. The model describes the entities and concepts and the relations among them for a runtime adaptable product. The model is an extension of the model depicted in Subsection 2.2.1. It also shows the stakeholders of a runtime adaptable system. 'RTA Product' stands for Runtime Adaptable Product.

The domain model is an extension of the SPL model from Subsection 2.2.1. A Runtime Adaptable Product (RTA Product) is part of a Product line, which is defined by the Requirements engineer. The

Requirements engineer consequently defines the required Utility and Behavior of the product. Multiple variation points and their variants realize the adaptability at runtime. The product is extended with an Adaptability Model. This Adaptability model describes the runtime adaptable system and consists of two models: a Feature model, that describes the variability of the product (i.e. the variation points and variants), and a Quality model, that describes the impact of variants on the utility of the product. The adaptability model and consequently the feature model and quality model are defined by the software architect. He defines the variation points, variants and quality of those variants in the product. The adaptability model consequently defines (and constrains) the possible design decision space (see Section 1.2.2) for the adaptability of the product.

The System engineer configures the system by making decisions. These decisions document the selected variants in the configuration of the product. As the System itself is a specialization of the system engineer, it is also possible that the system is adaptive by taking the decisions itself.

As the quality of a variant is defined in the Quality model, well-considered decisions can be made, because the decision maker can preview the influence of different variants of a variation point on the Utility of the product.

The Software developer implements the functionality (or component implementations) of the variant. As the functionality is related to the quality of the variant, the Software developer can also refine the quality model when the actual implementation's quality differs from that specified by the Software architect.

4.2 REQUIREMENTS

This section analyses the requirements for the model and the process of runtime adaptability. Chapter 2 introduced two scenarios in which the runtime adaptable model should work. From these scenarios, several requirements can be identified that need to be fulfilled by the model in order to be practically usable in those scenarios. The two scenarios can shortly be described as follows:

1. A scenario that shows how an adapted SPL process can be used at both development time and runtime, by defining that some decisions should be postponed.
2. A scenario that describes how the model can be used in our AmI-project by adapting Particles when resources are scarce.

Besides these requirements, which focus on the practical application of runtime adaptability, Section 2.5 described which adaptation concerns should be addressed theoretically in order for applications to be runtime adaptable. These concerns are also incorporated in the requirements.

The next section will address the stakeholders of the model and process. Subsequent sections will address the use cases, requirements and quality attributes.

4.2.1 STAKEHOLDERS

The domain model has shown several stakeholders of the runtime adaptable model and process of creating a runtime adaptable system. Below they are enumerated and elaborated:

- Requirements engineer

  The Requirements engineer defines the requirements of the products in the product line. He knows the domain in which the product should function and where it needs to be adaptable.
- Architect

  The architect of the system knows what components are available and how these components interact with each other. He also describes where variation points exist in the architecture and which variants are available for these variation points. Furthermore, he knows about the system's resources and required qualities. The architect has a high-level view on the system.
- Developer

  The developer implements the actual components of the system. He knows from a lower level how the system is implemented and composed, the constraints of that implementation and what quality can be delivered by that implementation.
- System engineer

  The system engineer specifies how the product is derived from the product line. He knows which components should be delivered to the client and which parts of the system should be adaptable after the system is delivered at the client.
- System

  Although not a human stakeholder, the system should in case of self-management be able to perform adaptations of the system itself. This means that the system needs to know how the system is composed at a certain point in time and how to change the system when an adaptation is required.

The last two stakeholders, system and system engineer, are the stakeholders that operate the runtime adaptable system. They are consequently the subject of the adaptation.
The described stakeholders are the stakeholders of the model and the process of creating the model, and therefore is the end-user of the application not depicted in the domain model, although he obviously benefits from an adaptable product.

4.2.2 USE CASES

Below, the use cases are depicted. These use cases capture the functional requirements of the model and the process. The use cases are based on the scenarios in Section 2.3 and the concerns for the model in Section 2.5.

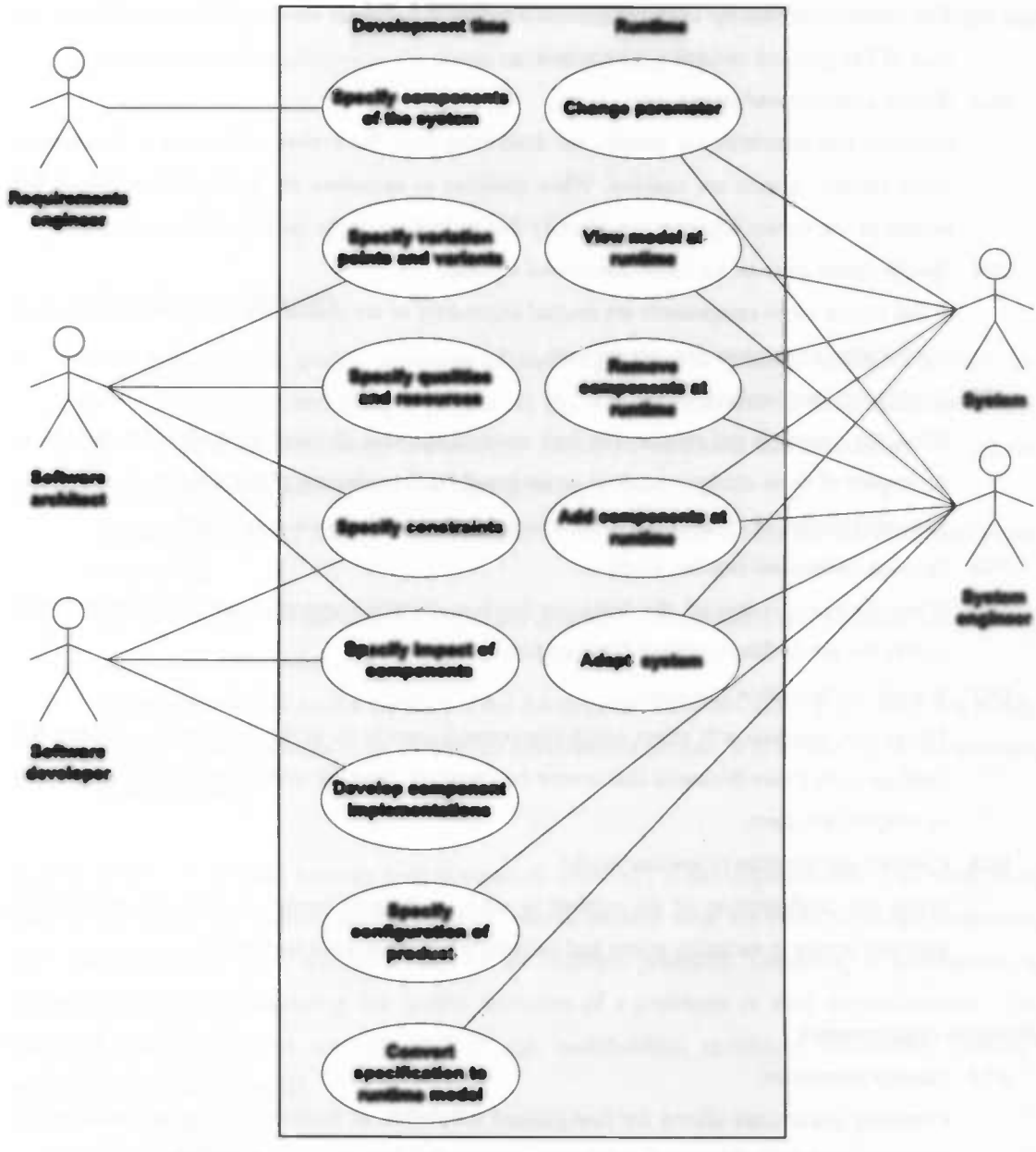


Figure 4-2: Use cases for defining the model and creating an adaptable product. The use cases are divided into development time use cases and runtime use cases.

The use cases are subdivided into uses cases at development time (left side of the picture) and use cases at runtime (right side). In the next subsection each of the use cases is described in more detail.

4.2.3 FUNCTIONAL REQUIREMENTS

Each of the use cases in Figure 4-2 are defined in such a say that they each capture a single requirement.

Development time requirements

UC1 *Specify components of the system.*

The requirements engineer is able to specify which components belong to the product.

UC2 *Specify variation points and variants.*

The architect defines for each component whether it is variant (course-grained variability). In case of fine-grained variability parameters are used.

UC3 *Specify qualities and resources.*

Qualities and resources are needed for describing how the system will be or is functioning when certain variants are enabled. When qualities or resources are getting out of range, the system or the system engineer can identify this by looking at the qualities and resources.

UC4 *Specify constraints on variation points and variants.*

When two or more components are mutual depending or are mutual exclusive with each other, a constraint can express this requirement.

UC5 *Specify impact of components.*

When the system is not functioning well enough, possible changes need to be identified and the impact of those changes needs to be analyzed for the selection of the right change. Without this specification of the impact, the decisions cannot be taken well-considered.

UC6 *Develop component implementations.*

When the model is created, the developer needs to develop component implementations which realize the actual functionality of the product.

UC7 *Specify configuration of product.*

The system engineer will select which components need to be in the final product. During this configuration phase decisions that cannot be taken yet, must be documented in order to create an adaptable system.

UC8 *Convert specification to runtime model.*

When the configuration of the product is ready, a runtime model needs to be created that provides access to variation points and variants while the system is running.

Runtime requirements

UC9 *Change parameter.*

Changing parameters allows for fine-grained adaptation at runtime. The system and system engineer need to be able to adapt the system using this fine-grained adaptation mechanism.

UC10 *View model at runtime.*

When the system is running and needs to be adapted, the system architect and the system itself need to have a view of the current system's configuration to find out what can be adapted and what impact that adaptation will have.

UC11 *Adapt system at variation points.*

When a possible adaptation is identified, it must be possible to execute that adaptation while the system is running.

UC12 *Add components at runtime.*

When the system is extended at runtime by adding components, the model must support the ability to reflect this change in the model as well and know what impact it will have on the system.

UC13 *Remove components at runtime*

When components are removed from the system, the model should reflect these changes and might need to find alternative for these removed components.

4.2.4 QUALITY ATTRIBUTES

This section addresses the quality attributes of the model and the process. Quality attributes are orthogonal to functionality, according to Bass et al. [6], which means that different decisions for the functionality of the model will result in a different quality of the model and the process. To test the quality of the architecture, Bass et al. define the following qualities [6]:

- Conceptual integrity addresses the integrity of the vision that unifies the design of the system at all levels.
- Correctness and completeness address the question if all system's requirements and runtime resource constraints are met.
- Buildability focuses on the question if the development team can develop a system in a timely manner even if changes occur during the development process. It is a measure for the easiness of development.

These architectural qualities are very hard to measure, especially at development time. This holds even more for the model and process developed in this thesis, because these are made on a high abstraction level (meta-modeling level) and try to capture the concepts generally. Therefore, an architecture is normally validated by measuring the quality attributes of a prototype or final implementation. The measured quality attributes normally include e.g. modifiability, testability, availability, security, usability, and performance [6].

Modifiability is the most important quality attributed that is addressed by the model and the process in this thesis, since modifiability is a generalization of adaptability. The architectural drivers of runtime adaptability were defined in Chapter 1. These drivers are based on the self-management ability and comprise the following:

- Self-configuring. As this type concerns the ability of an application to configure itself, the quality is called configurability.
- Self-healing. As this type concerns the ability of an application to heal, and consequently keep functioning, this quality is called reliability.
- Self-optimizing. This type concerns the optimization of the application, the quality is called optimization.
- Self-protecting. This type addresses the security of the application and the ability to protect against bad stuff, the quality is called protectability.

Figure 4-3 shows a conceptual view on these qualities:

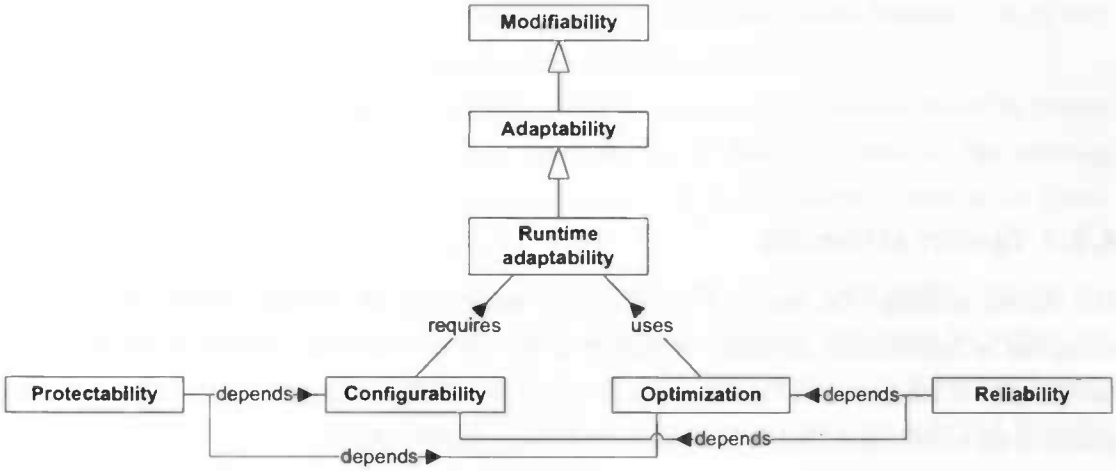


Figure 4-3: Overview of the quality attributes of the model and the process for runtime adaptability.

On the bottom of the figure, the four self-management quality attributes are depicted. Both Protectability and Reliability are depending on the quality to configure or to optimize the application. This is the case, because these two qualities react on the discovery of possible attacks or disruptions, respectively, and require consequently the ability to reconfigure or optimize the application. For example, an attack on a wireless network connection could be prevented by switching to an encrypted communication protocol, or a busy web server could be optimized by using a larger connection pool to handle an increase in incoming client connections.

Both Configurability and Optimization depend on the ability of an application to be runtime adaptable. Runtime adaptability is a specialization of Adaptability, which can take place at either development time, runtime or maintenance time. Adaptability is a specialization of Modifiability, the main system quality [6]. Since runtime adaptability is the focus of this thesis and is a quality in itself, configurability and optimization at runtime are the most important quality attributes to test for in the approach that is being developed.

Both qualities are supported by the scenarios defined in Section 2.3. Configurability is the main focus of the first scenario; optimization is the focus of the second scenario.

The process described in Chapter 5 tries to leverage the buildability of a runtime adaptable system. Although it is very hard to quantify this architectural quality without building a real system, the validation subject in Chapter 6 will make some qualitative statements about the buildability of the model.

5 MODEL-DRIVEN RUNTIME ADAPTABILITY

Based on the requirements defined in Chapter 4, this chapter presents a model and a process that will address the problems encountered when developing runtime adaptable systems. The next section will give a complete overview of the process to develop a runtime adaptable system. Section 5.2 introduces the feature model used to specify variability in the product. Section 5.3 focuses on the specification of a quality model and Section 5.4 shows how the two models are combined in the adaptability model. Section 5.5 addresses the conversion from the adaptability model to source code.

5.1 PROCESS OVERVIEW

Although a model will help to focus on the important concepts and attributes for runtime adaptability, describe the system's points of adaptation at runtime, and the explicit documentation of an adaptation's impact, describing a process that dictates the steps needed to create a runtime adaptable system will render the model more useful.

This process is depicted in Figure 5-1. Most of the entities shown in the figure, such as the feature model and the quality model, are defined in the domain model introduced in Section 4.1. New entities are discussed below.

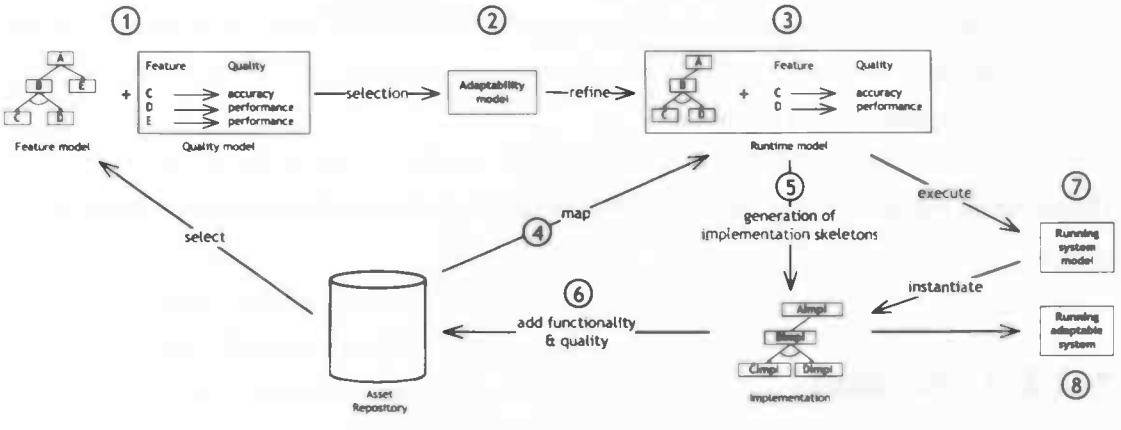


Figure 5-1: Overview of all the steps that are taken to generate a runtime adaptable system.

The following steps are required for the creation of a runtime adaptable system:

1. The feature model is defined for the product line. This model can be developed from scratch or can be composed of selected features from the asset repository. This asset repository contains previously developed features (assets) and allows for reuse of those features. The feature model describes the required variability. The quality model defines the qualities of the system and defines the impact of the variable features on the quality of the system. For new products, this information might not be available, as the component implementations are not

implemented yet, and will be added afterwards. When for example the impact on performance or specific resources cannot be estimated, these need to be measured first after the implementation of that feature is ready. The figure shows three variant features (C, D, E) with the influence they have on the qualities defined in the quality model.

2. A configuration is defined for a specific product by selecting which features should be present in the product. Decisions that cannot be taken can be delayed to runtime. The resulting configuration, with open and closed variation points, is the adaptability model. In the figure, feature 'E' is not selected for inclusion.
3. The adaptability model is refined to the runtime model. The runtime model describes the product at runtime, which components are included in the product, and where the product can be adapted. Because this runtime model must be usable at runtime, it is transformed into a model in code. During the refinement process, unnecessary details are removed from the runtime model.
4. Previously implemented features from the asset repository can be mapped to the runtime model for reuse. For these model elements, no component implementation skeletons need to be generated.
5. For model elements that are not mapped to component implementation, skeletons are generated to be completed with the actual functionality by the developer. These skeletons describe the minimal functionality for an asset to be adaptable.
6. Functionality and impact on the qualities of the system are defined and stored in the asset repository for later reuse. The system is ready for execution.
7. The runtime adaptability model is executed. All necessary component implementations are instantiated from the model. Open variation points are resolved based on the system's current context.
8. The adaptable system is running.

The process in Figure 5-1 shows how the models are used in the process. The next sections describe the models.

5.2 FEATURE MODEL

This section introduces the models for runtime adaptability. As several design decisions have been taken to shape the following models, it is important to document them to find weaknesses when validating them (validation is subject of Chapter 6). Each design decision is prefixed with 'DD' and is numbered.

DD1 *The required variability is specified in a feature model.*

Since feature models are on a convenient level of abstraction when talking to all the stakeholders, a feature model is used for the specification of the required variability. Unfortunately, a standard feature

model (such as FODA) does not describe the required variability and therefore an adaptation is necessary to support the variability required.

DD2 *OR and XOR constraints are supported by the grouping of features.*

The feature meta-model is depicted in Figure 9-6 and shows one interesting difference with FODA: A feature modeling element (FMElement) is specialized in a Feature and a FeatureGroup. Feature Groups are necessary to keep the constraints between features manageable later in the runtime model. The specification of just OR and XOR constraints to create alternatives or optional variant groups, respectively, makes the runtime analysis of which variant can be selected for a variation point far more difficult than necessary.

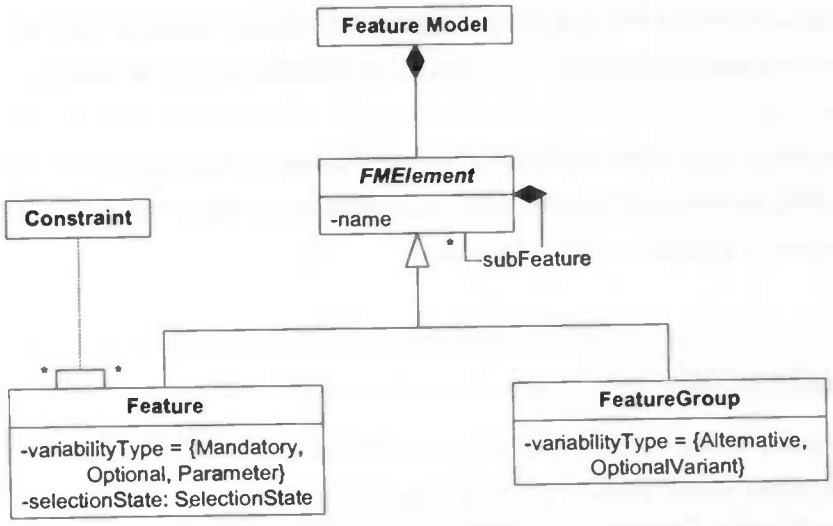


Figure 5-2: Feature meta-model. It adds a grouping element for easier selection of variants for a variation point.

DD3 *The feature model allows five different types of variability, including parameterization.*

The feature meta-model can express five different types of variability, defined in the VariabilityType enumeration:

- Mandatory – this feature must be in the final product;
- Optional – this feature may be in the final product;
- Parameter – this feature represents a specific value from a certain type (e.g. integer). This feature will be embedded in the parent feature's implementation and will be present in the final product if the parent feature is selected for inclusion;
- Alternative – this feature group represents a single selection from several sub-features, the variants;
- OptionalVariant – this feature group represents a multiple selection from several sub-features.

Although parameters are usually fine-grained and normally do not exist on such an abstract level as feature models, parameters add an important type of variability and are therefore added.

DD4 *A feature configuration is a clone of the feature model.*

Since it is important to not only know which features are selected for a product, but also know how these features relate to each other – especially at runtime – a feature configuration is defined as a clone of the feature model, added with the selection state of each feature. This is different from e.g. Chameleon, where a feature configuration is a subset of the complete Feature set of the feature model. Therefore, the Feature meta-model element is extended with a SelectionState attribute. The attribute can have the following states:

- LeftOpen – No decision for this feature has been taken yet.
- Selected – This feature is selected for inclusion in the feature configuration.
- Runtime – The decision for selecting this feature has been delayed to the runtime.
- Unselected – This feature must not be included in the feature configuration.

A disadvantage of cloning is that changes in the feature model made during or after the configuration are not seen in the feature configuration, as it must be re-cloned to reflect those changes.

Due to the possibility that dependencies exist between features not belonging to the same sub-tree of the feature model, the Constraint meta-class has been added. This class allows for the specification of a requires-constraint or a mutual-exclusive constraint.

5.3 QUALITY MODEL

Providing a quality model for the system is a non-trivial task. First, the quality model must be composable of other quality models, since an AmI system can be extended at runtime, adding the qualities of the extension to the system. Second, quality attributes can be qualitative and quantitative, depending on their kind, which means that they cannot always be formally specified by numbers. Third, every subsystem (such as the iCup) has its own quality attributes and resources, and some of them are not relevant to the complete system, but only for the local variation point. Fourth, there is a difference between localized optimization and global optimization and both can be opposing each other. This requires special optimization strategies such as 'global first and local second' when adapting the system.

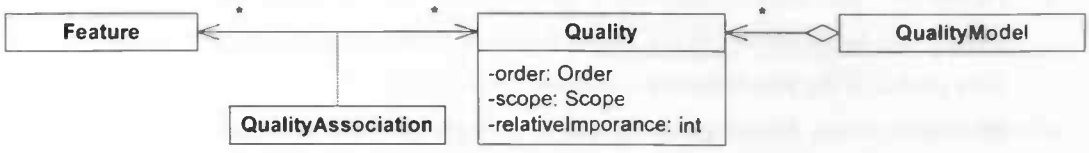


Figure 5-3: Quality meta-model. Using a quality association, a feature can define its influence on the qualities of the system.

DD5 *The quality model is based on the COVAMOF meta-model.*

The Quality meta-model in Figure 5-3 is highly influenced by COVAMOF's Dependencies. This decision is based on the simplicity of the COVAMOF dependency model and its usability in practice

[38, 39]. Chameleon's quality model is more expressive, but also more complex to implement for a runtime adaptable system. Furthermore, it is also similar to the correlations used in Design spaces.
The quality model shows that a feature is associated with a quality by means of a QualityAssociation. This QualityAssociation describes the impact of a feature on a quality. A QualityAssociation can be specialized to express a quantitative or a qualitative impact.

The Quality meta-class describes a general property of the system, which can express a quality of the system or a resource, either quantitative or qualitative. The order attribute in the Quality meta-class – an enumeration with the values Increasing and Decreasing – defines if more of this quality is good or bad for this quality as a whole. This allows the system to find out if an increase of a certain quality is good or bad for the system. The scope attribute is also an enumeration that defines if this quality scope is Local or Global. If a quality scope is local, the impact of a feature will only have local impact that is not of interest of the entire system (which would be global). If the quality's scope is defined as global, the impact affects the system as a whole. When a feature is added during execution, (adding is only possible at open variation points), the global qualities and their associations are added to the quality model of the system, local qualities and their associations are added to the quality model of the variation point.

The relativeImportance attribute defines an order among the qualities in the system. E.g. for the iCup, the Energy consumption rate is more important than code size. This allows easier optimization of the important qualities of the system.

The Quality model is composed by the architect and refined by the developer, since they have the best knowledge about what impact a certain feature may have on the system. The figure below depicts an example quality model for three features of the iCup (see the iCup feature model on page 37).

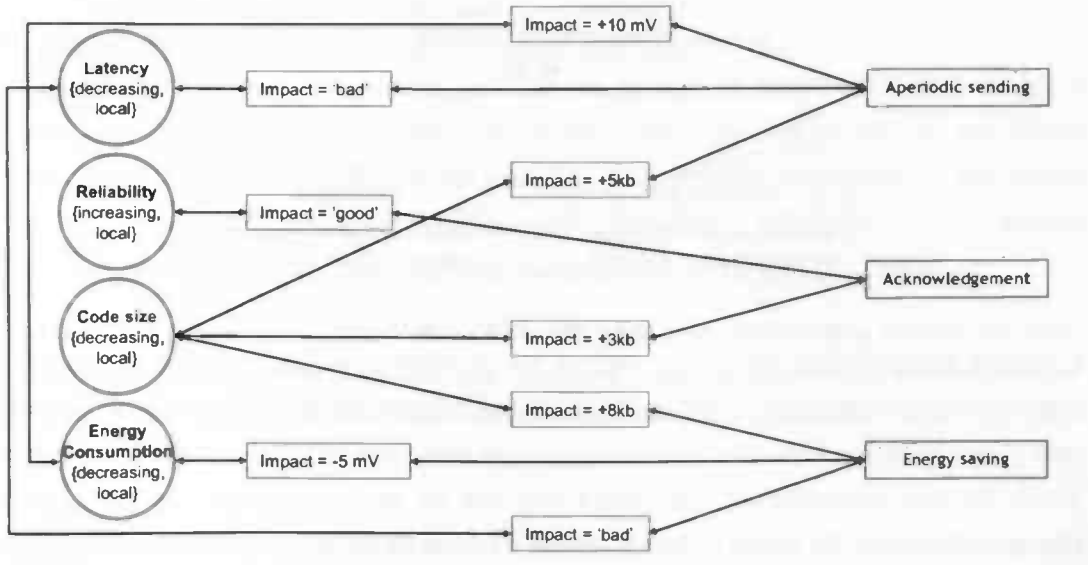


Figure 5-4: Example quality model diagram. It shows the effect of several features (on the right) of the iCup on specific qualities (circles on the left) of the system by means of a quality association (boxes in the middle).

The figure shows that the 'Energy saving' feature affects the Energy consumption positively, i.e. when the energy saving feature is selected for the iCup its 'Energy consumption' decreases by 5mV (lower is better as defined by the decreasing order). Furthermore has the 'Energy saving' feature bad influence on the latency of the iCup (the latency increases) and requires 8kb more memory of the system.

5.4 ADAPTABILITY MODEL

The feature model and the quality model together define the adaptability model. This combined model defines the features of the product line, the decisions that are to be taken for defining a product and the rationale for those decisions for creating a product that fulfills its quality requirements.

The next step is the actual configuration of the product. The system engineer, who configures the product, takes the adaptability model and selects which features have to be included in the product. For decisions that cannot be taken yet, e.g. due to the fact that it is unknown if the feature is needed, the selection state can be set to 'Runtime'. This also holds for features that are part of the runtime adaptability of the system.

Figure 5-5 shows an example selection by the system engineer for the 'Sensor' feature in the iCup's feature model. The choice for which sensor to use (that is, ForceSensor or TiltSensor) is extended to the runtime by setting the selectionState attribute to 'Runtime'. This means that the variation point stays open until a decision is made at runtime. If only one of the two sensors' selectionState was set to 'Selected', the variation point could be closed at development time, since there would only be one alternative left.

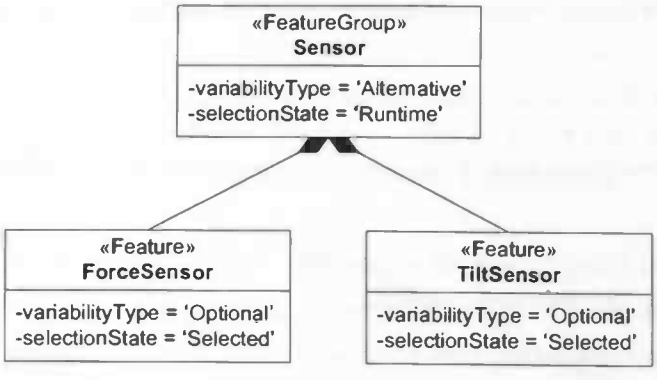


Figure 5-5: Example of the Feature model of the iCup. It shows the FeatureGroup 'Sensor' that is defined as a variation point with two alternative features: 'ForceSensor' and 'TiltSensor'. The selectionState of the 'Sensor' is defined as 'Runtime', which means that at runtime the decision is taken which one of the (selected) features will be used.

During configuration, the impact of the selection of a feature for inclusion can be found in the Quality model. This allows the system engineer and architect to make well-considered decisions.

5.5 CONVERSION TO CODE

When the configuration for the product is complete, the adaptability model must be converted to a model in code that can be used at runtime. Since such a task is similar for each variation point and variant, this step can be automated by a tool. This alleviates the developer and he does not need to care about how to implement a runtime adaptable model. The conversion step will create two assets for each feature: (i) a runtime model asset that describes how the system currently looks like, and (ii) an implementation asset that realizes the actual implementation of the feature in code. Figure 5-6 shows this in a three-dimensional graph.

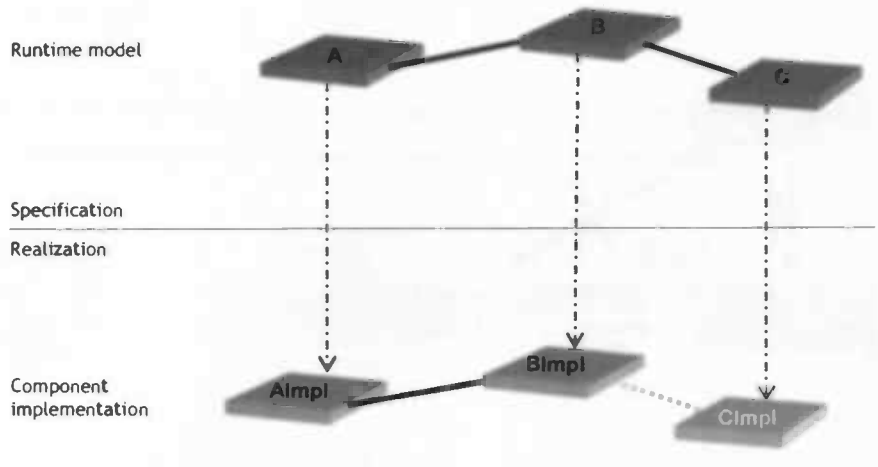


Figure 5-6: Three-dimensional view on the two assets that are created during the conversion from the adaptability model to the runtime model and the component implementations (here depicted for three features: A, B and C). Both have a similar external form, but the runtime model has access to the component implementation (depicted by the arrow) and has control over what component implementations are running during execution of the system. The implementation of C (CImpl) is in this case disabled (in gray displayed) by its model element C, the others are enabled.

DD6 *There is a distinct separation between specification and realization.*

The separation between how the system is specified (i.e. the runtime model) and how the system it realizes (i.e. the component implementation) allows for a clear separation of concerns and reduces complexity. The idea is that the runtime model essentially floats above the implementation and controls it from there. It consequently uses the reflection pattern described in Subsection 2.5.3. The runtime model is located in the meta level, the component implementation reside in the base level.

The conversion of the adaptability model to the runtime model and implementation requires a refinement of the adaptability model (see Figure 5-7), since converting the adaptability model directly to code does not foster reuse. The refinement step will convert the adaptability model in such a form that it can be stored in an asset base for later use. Furthermore, the refinement step will do the following:

- Convert open decisions (selectionState = 'Runtime' / 'Optional') to variation points in the runtime model.

- Convert sub-Features of FeatureGroups into variants and adding the qualities and the effect on those qualities to the quality model.
- Remove unselected features
- Remove variation of optional features when those are selected and need to be included in the product.
- Remove unnecessary attributes and details from the adaptability model. This will reduce the overhead of the runtime model.
- Add static information about the variability, such as minimum/maximum resolution size, and functionality for extending/reducing variation points with new variants.

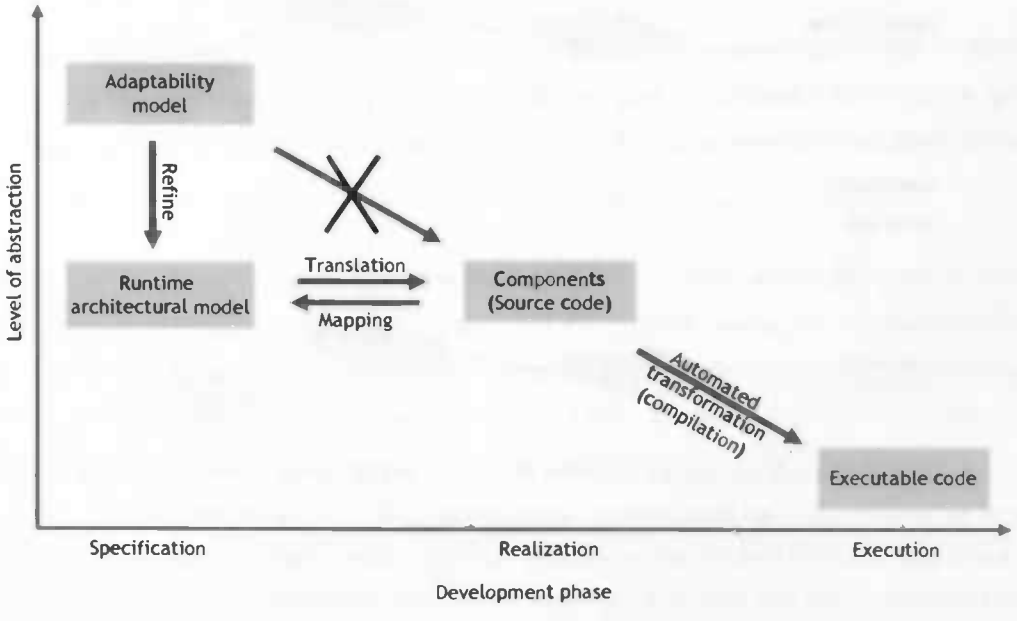


Figure 5-7: Conversion from the adaptability model to executable code requires a refinement step (and not a direct translation, depicted by the cross) to foster reuse and a translation step to convert the model into source code. The mapping back to the runtime architectural model allows for reuse of source code. Compilation will finally generate executable code that can be run. The figure is based on lecture slides by D. Muthig, during a presentation about SPLs at the University of Kaiserslautern, Germany, December 2005.

Next, code can be generated from this model. The code for the runtime model consists of two layers. The first layer describes all the elements of the system and how they are connected to each other. The second layer is a meta-model for the first layer (actually the meta-meta-model for the implementation) that describes each variation point and variants and their properties.

DD7 *The runtime model uses a component-based framework.*

When the process is component-based, it allows for reuse on a small scale (i.e. implementation level, see Section 3.2 on KobrA). This reuse is necessary for the required adaptability, when new components are added to the system. As there is a separation of specification and realization, a component is described at both levels of abstraction: for the specification level the adaptation behavior, for the realization level the functionality.

The code for the implementation is much simpler. Every implementation implements the Implementation interface that has two methods: startUp and tearDown. These methods are called when an implementation is started or stopped by the runtime model. Since non-variant features (i.e. mandatory features) of the feature model can be a sub-feature of a variant feature, these features need to implement this interface as well.

DD8 *The model supports a bottom-up approach.*

Developers can already start with the implementation when the runtime model is not yet generated. If they implement the Implementation interface, the implementation can be easily added to the runtime model later in the process.

Figure 5-8 depicts an example part of the classes generated for the feature model of the iCup in Section 3.1. It shows the main feature 'iCup' and the optional feature 'Aperiodic sending'.

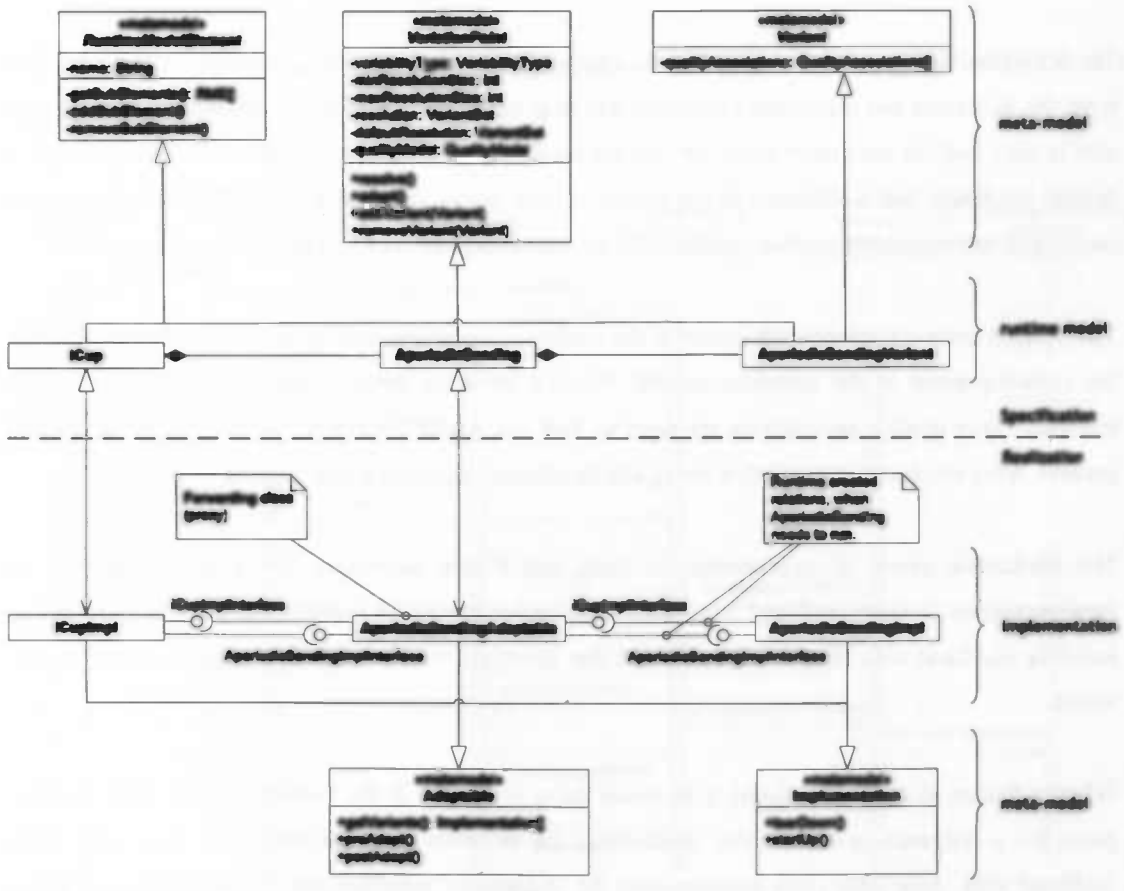


Figure 5-8: Excerpt of the classes that are generated when the adaptability model is transformed into source code. This UML diagram shows the feature 'iCup' and 'Aperiodic sending' and how they are connected to each other. There is a visible distinction between the specification of the adaptability and the realization thereof.

The diagram is split into two parts: the upper half shows the specification of adaptability and the lower half the realization. The specification will be addressed first.

Each runtime model element inherits from the RuntimeModelElement meta-class. This class creates structure among the features in the system and allows you to navigate through the model. Since 'Aperiodic sending' is an optional feature, the feature is split into two classes: a variation point (APeriodicSending) and a single variant (APeriodicSendingVariant). This concept allows the variation point to still have a reference to its parent (in this case the ICup) and vice versa, which means that the references that the iCup holds to the 'Aperiodic sending feature' are always valid. Furthermore, when 'Alternative' or 'OptionalVariant' variation points are used, the same concept can be used, provided that there will be more than one variant available.

DD9 *The runtime model is compositional.*

As the model can be extended at runtime, the model needs to be compositional. When a variant is added to a variation point, the model must reflect this addition. Furthermore, the variant's quality associations must be added to the quality model, to allow for well-considered decisions about the usage of this variant.

The VariationPoint meta-class holds several attributes needed to specify a variation point. It holds it type, the minimum and maximum resolution size (e.g. for an optional feature, the minimum resolution size is zero and the maximum one), the current resolution (the variants that are currently running), a default resolution and a reference to the quality model of this variation point and its variants. As the model can be extended at runtime, variants can be added and removed to a variation point.

The Variant meta-class holds references to the quality associations, i.e. the impact on the qualities on the variation point or the complete system. When a variation point is resolved (using the resolve method), these quality associations are used to find out which variant functions best in the current context. After resolving, the variation point will be adapted using the adapt method.

The realization exists of implementation code and a few interfaces. For every component an implementation skeleton (suffixed with 'Impl' and implementing the Implementation interface) and an interface (suffixed with 'Interface') is created that describes which methods are public to the outside world.

When a feature is marked optional, a variation point is created in the runtime model. This variation point has a reference to a class that implements the variation point on realization level. This class (suffixed with 'Adaptable', and implementing the 'Adaptable' interface) can be seen as glue or proxy between the parent feature and the optional (sub-) feature. It is based on the Microkernel pattern described in Subsection 2.5.3 as it adds a layer of redirection. It forwards method calls back and forth between both features. This allows the references of the parent feature to be always valid and specific code can be added if necessary when the specific variant is not available. Therefore, it implements the

interfaces of both the parent feature and the sub-feature (for the shown example, the ICupImplInterface and the APeriodicSendingImplInterface).
This means that the variants of a variation point are implemented as plugins. Consequently must each component implementation of a variant commit to a specific interface of the variation point (in this case the APeriodicSendingImplInterface) to allow for extending or changing variants at runtime, without the need to rebuild or redeploy.

Furthermore, this Adaptable-class holds an array of implementation variants, which allows developers to define what should be done when more than one variant is running for a particular variation point (e.g. for robustness or precision) and how methods calls should be routed to the different variants.

DD10 *The runtime model controls the implementations.*
The runtime model is in control of which implementations are running. This means that the communication between the model and the implementation is unidirectional. This keeps the implementations clean from any model-related code.

Figure 5-9 shows a sequence diagram of the specification of a variation point and the adaptation of a variation point at runtime.

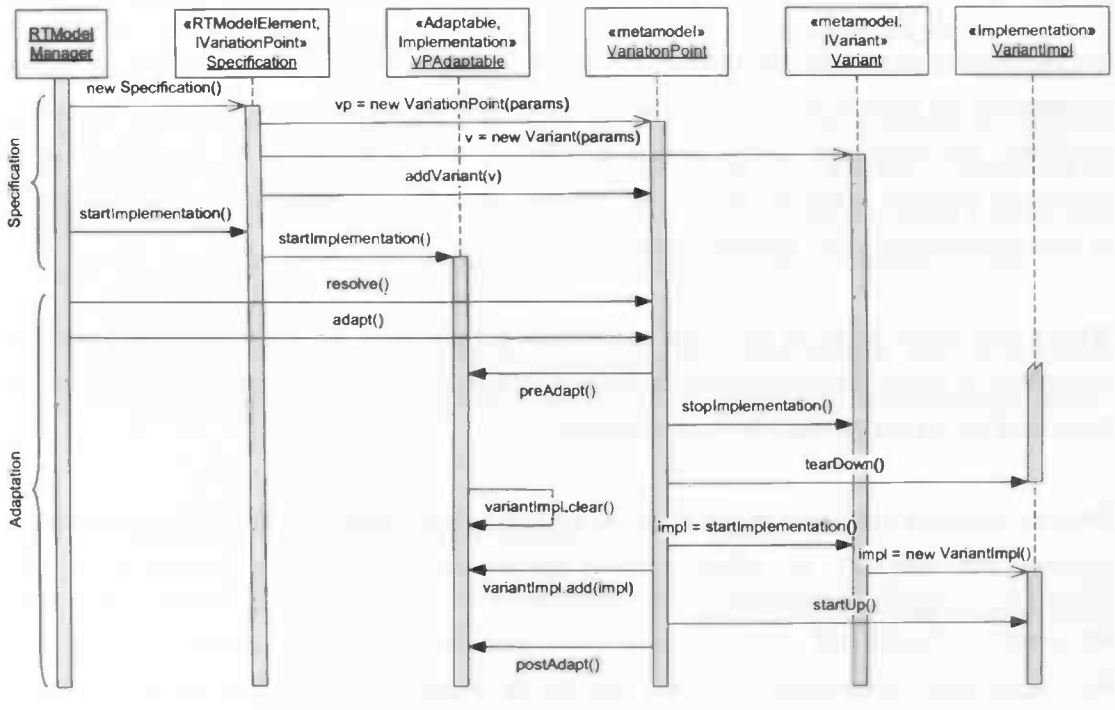


Figure 5-9: Sequence diagram showing the specification of the adaptability and the adaptation of a variation point.

On the left is the Runtime Model Manager depicted (RTModelManager). This manager will load the specification of the adaptability by instantiating the runtime model. The depicted Specification describes in this case a variation point and is part of the runtime model of the application (such as the

AperiodicSending class from Figure 5-8). When the Specification is created, a VariationPoint class is initialized that realizes the variability. This class contains all the necessary functionality to adapt and resolve a variation point. Subsequently are the variants added to the variation point. Then, the Adaptable class (VPAdaptable) is instantiated. This class implements the earlier described forwarding class between the parent component implementation and the variant component implementation (see also AperiodicSendingAdaptable in Figure 5-8).

At runtime a variation point can be resolved by calling the resolve() operation. Different strategies can be used by overriding the resolve() operation and implementing a different resolve strategy. This operation could e.g. consult the quality associations of the variants and find the best fitting variant based on the influence of that variant on the quality model. The current strategy will use the default resolution that is specified for a variation point (as this thesis does not address the reasoning part of runtime adaptivity). If no default resolution is specified, no variant will be instantiated and requires the system engineer to close the variation points.

When an adaptation takes place (by calling the adapt method of a variation point in the runtime model), the 'Adaptable' class is informed by the invocation of the preAdapt method. This allows the system to save the state of the previous running variant. After the system is adapted, the postAdapt method is called, allowing the restoration of the state or configuration of the new variant.

During the adaptation, component implementations of variants (VariantImpl) that are not required to run anymore, are stopped (by calling the stopImplementation method on the specification of the variant (Variant)). The component implementation is informed of this adaptation by the calling of the tearDown() operation. It has time to clean up its references to other objects in the application and save its state. References to those implementations are cleared.

When a new variant performs better, the component implementation for that variant (VariantImpl) is instantiated by calling the startImplementation method. Next the startUp() operation is called to restore the state of the variant's component implementation.

Since the developer only needs to implement the startUp and tearDown methods of the Implementation interface and there are no references from the implementation to the runtime model, the implementation itself is kept very clean.

6 VALIDATION

This chapter's focus lies on the validation of the model developed in Chapter 5. The input for this validation are the functional requirements and quality requirements from the previous chapter. The runtime adaptability model cannot completely be validated by these requirements (especially the quality requirements), because some of the properties cannot be measured when the model is not instantiated.

Chapter 4 identified that architectural quality attributes such as model completeness and model buildability are hard to measure. Therefore, prototype applications are used to measure if the model-based prototype meets the system qualities and, if that is the case, derive from that fact that the architectural qualities of the model are met as well.

This procedure is not possible for the model and process developed in the previous chapter, as there is not a prototype build for the BelAmI-project, yet. Consequently is the extend of validation in this thesis limited.

Chapter 1 repeatedly identified that incorporating adaptability in an application is very complex. The model and process developed in the previous chapter focus on the reduction of this complexity and identified that tool support will leverage the process of creating a runtime adaptable system.

To validate this statement, a prototype tool has been developed that supports the process of creating a runtime adaptable system by using the models in the previous chapter. This tool allows us to test if the functional requirements are met and if the complexity is reduced. Since the tool needs a computable version of the models, the tool is a good test case if the theoretical models are usable in practice.

The prototype tool will be introduced in the next section. In subsequent sections, both the theoretical model and its prototype implementation are tested for how good they meet the specified requirements.

6.1 PROTOTYPE TOOL IMPLEMENTATION

Part of the validation is the conversion of the theoretical model described in Chapter 5 to a model instantiation that shows that the theoretical model can be used in practice. That chapter also hinted that the conversion from the adaptability model to the runtime model and implementation was a perfect task for a tool. As such a conversion needs to instantiate the theoretical models it is a good test if they are practically realizable.

The 'runtime adaptability tool' provides three services:

1. The composition of a feature model and a quality model.
2. The composition of a feature (or product) configuration.

3. The generation of the runtime model and implementation skeletons.

The tool's architecture is shown in Figure 6-1. It shows a typical model-view-controller style that separates the concerns for modifying the model very well. The user interface (UI) provides several views on the model:

1. A feature model view (FM TreeView) that allows the architect and developer to define the features of the product line. Variation points are defined here too.
2. A constraints view (Constraints TableView) that allows for the specification of constraints among features.
3. A quality specification view (QualitySpec TableView) that allows for the specification of the qualities and resources of the system.
4. A quality model specification view (QM View) that allows for the specification of impact from features on qualities or resources.
5. A feature configuration view (Feature ConfigurationView) that allows for the specification of which features should be in the final product and which variation points should be kept open. In this view, the impact of each decision can be seen. Furthermore, if a decision leads to a non-functioning system – according to the quality model – the user is informed.

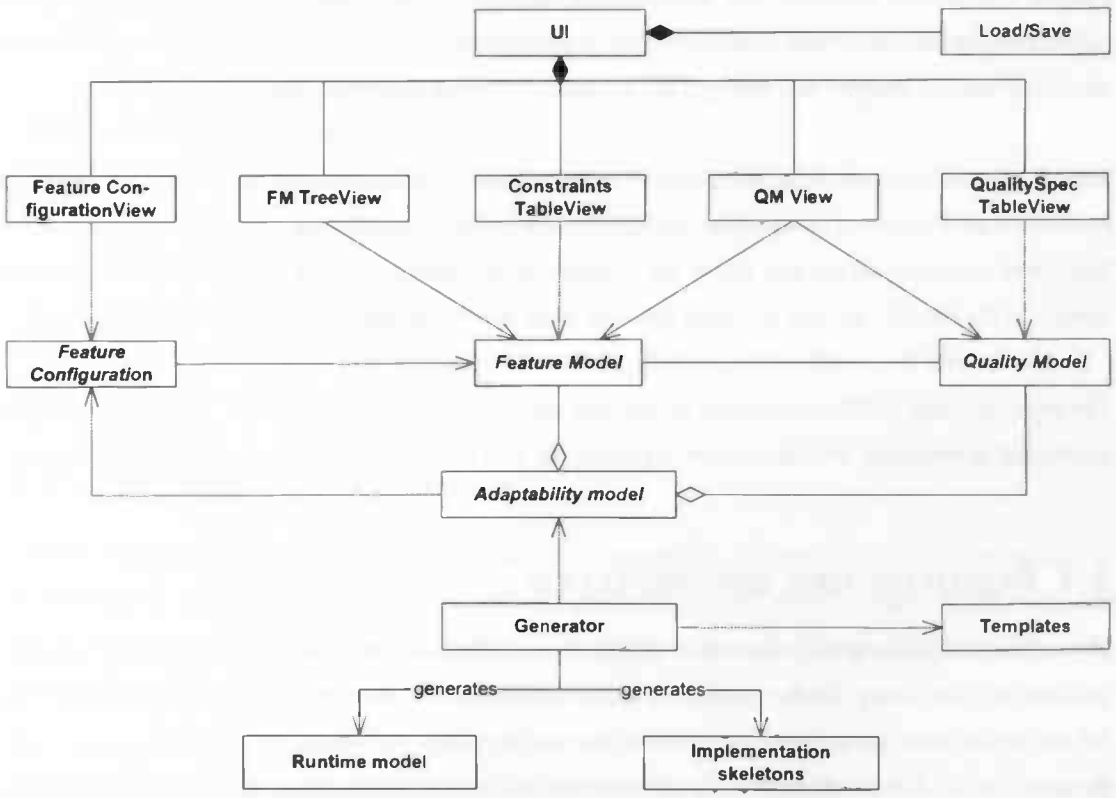


Figure 6-1: The runtime adaptability tool's architecture. It shows a model-view-controller style which separates the model from the view and makes the development less complex. The models have italic titles.

The UI also provides the ability for loading and saving a model.

Below the views are the models that can be manipulated by the tool: (i) the feature model, (ii) the quality model, and (iii) the feature configuration.

When these models are specified, the Feature Configuration is used to define the adaptability model. Based on this model, the implementation skeletons and the runtime model are generated by the Generator. The implementation skeletons must be extended with the application logic described by the feature (and its requirements). Additionally, the correct impact should be added to the adaptability model when that is not already done in an earlier step. Thereafter, the system is ready for execution.

The tool comes with a ready-made model manager that is able to instantiate the model and starts the implementations. It will also resolve open variation points and instantiate the variants defined by the resolution. Furthermore, it adds a GUI that is able to create a view of the model and which elements are running and which qualities they possess.

The tool generates Java classes by filling in several predefined Java-templates. Other programming languages can be added by extending the generator and replacing the templates. In *Appendix B: Screenshots of the tool,* screenshots can be found to get an impression of how the tool works.

Besides the design decisions made above, several other important decisions have been taken that constrain the usage of the model:

- The expressiveness of the quality model has been reduced, due to time constaints. Only quantitative constraints and impacts can be specified and it is therefore not possible to describe qualitative impacts on qualities of the system (unless they are quantified to some numerical form).
- Not all possible constraints can be specified between features in the tool. Although the model supports constraints, the tool only allows for the specification of OR and XOR constraints using FeatureGroups currently.

6.2 FUNCTIONAL REQUIREMENTS ANALYSIS

In this section, the implementation will be tested if it supports the use cases (which each address a single functional requirement) from Section 4.2.2. The table below enumerates all the use cases and shows how well the use case is supported by the runtime adaptability tool.

Table 6-1: Overview of supported use cases by the runtime adaptability tool.

Use case	Supported by the tool?
UC1: Specify components of the system	Yes
UC2: Specify variation points and variants	Yes
UC3: Specify qualities and resources	Partially
UC4: Specify constraints on variation points and variants	Partially
UC5: Specify impact of components	Partially
UC6: Develop component implementations	Yes

UC7: Specify configuration of product	Yes
UC8: Convert specification to runtime model	Yes
UC9: Change parameter	Yes
UC10: View model at runtime	Yes
UC11: Adapt system at variation points	Yes
UC12: Add components at runtime	Partially
UC13: Remove components at runtime	No

Most use cases are supported by the runtime adaptability tool. The ones that are partially or not supported are listed below with an explanation:

UC3 *Specify qualities and resources.*

The tool only allows for the specification of quantitative qualities and resources. Qualitative specification is not implemented yet due to time restrictions. It also adds extra complexity of the tool and the runtime model, since qualitative constraints must be handled differently.

UC4 *Specify constraints on variation points and variants.*

This use case is only partially supported, since the runtime adaptability tool only allows for the specification of OR and XOR constraints by means of FeatureGroups. 'Requires' or 'mutual-exclusive' constraints between features with different parent features are not yet implemented.

UC5 *Specify impact of components.*

The tool only supports the specification of quantitative impacts, due to the same reasons as UC3.

UC12 *Add components at runtime.*

Adding components at runtime is only partially supported. The generated runtime model allows for runtime addition of new components, but the Runtime Model Manager offers no functionality to actually add a new component.

UC13 *Remove components at runtime.*

This functionality has not been implemented yet, due to a limited amount of time.

6.3 QUALITY ATTRIBUTES ANALYSIS

Chapter 4 identified that measuring the completeness, correctness, conceptual integrity and buildabilty of the model is hard. To validate these architectural attributes, system attributes are normally validated in a prototype and the extend of coverage of these requirements is used to deduced the validation of the architectural attributes (if done at all).

Due to the fact that there is no prototype application to test the quality attributes (this chapter has only described a tool that leverage the process of creating runtime adaptable applications), this section will only do a qualitative analysis instead of a (measurable) quantitative analysis. This qualitative analysis is supported by an evaluation of the design decisions made during the creation of the model and the process in Chapter 5. Because design decisions constrain the possible solution space of the final implementation (see the design pyramid in Chapter 1), this evaluation will try to find weaknesses in the

model and process by analyzing these design decisions. The evaluation is based on the Architecture Tradeoff Analysis Method (ATAM) [17] for analyzing an (implementation of an) architecture. The input of the analysis is based on scenarios that have impact on the drivers and on design decisions made for the model. These design decisions are already enumerated in the previous chapter when the model was presented.
When the model is analyzed, risks, tradeoffs, and sensitivity points are found and provide information on the utility or 'overall goodness' of the model.

The most important quality attributes of the model, as defined in Chapter 4, comprise configurability, optimization and buildability. Configurability and optimization address the runtime adaptability leveraged by the model, and buildability addresses the applicability of the process to incorporate a model in a software system. Each of these three quality attributes are refined as follows:

Table 6-2: Quality attributes and their refinement.

Quality Attribute	Refinement	Description
Configurability	Runtime adaptability	Configure at runtime by adapting the system
	Extensibility	Configure for new extensions at runtime
	Maintainability	Maintainability costs of a model-based system
Optimization	Optimization	Optimization of the configuration at runtime
Buildability	Effort	Effort of implementing a runtime adaptable system
	Overhead	Overhead induced by the models at runtime

For each of the refined quality attributes a small scenario will be described. Although this scenario is quite abstract and not measurable, it helps creating a context for the qualitative analysis. Based on this context, design decisions are selected that influence the scenario and this influence is then analyzed.

Table 6-3: Quality refinement and the generic scenarios in which they are tested.

Quality refinement	Scenario
Runtime adaptability	C1: A runtime adaptable system is created with several variation points. When the system is started, the system engineer adapts the system at the variation points.
Extensibility	C2: A runtime adaptable system is running and a new component is detected. The system incorporates the new component and is able to instantiate it.
Maintainability	C3: The system engineer recalls a design decision after the system has been deployed.
Optimization	O1: The system engineer selects one of the qualities of the system and sees that one variant performs better than the currently running variant. He replaces the variant for an alternative with better utility.
Effort	B1: A runtime model is being specified and the developer wants to develop the component implementations that realize the specification.
Overhead	B2: A runtime adaptable system is running. The CPU and memory overhead for the model should not render the application's utility zero.

The specified scenarios are based on the two scenarios defined in Chapter 1. They focus, however, on manual runtime adaptation and not autonomous runtime adaptation. Scenario C1, C2 and C3 address a

part of scenario 1, where system engineer Bob specifies that a certain decision is extended to the runtime. Scenario O1 addresses an optimized reconfiguration of the system, described by the second scenario, where the system is optimized for bandwidth usage during an emergency of Mrs. Alice. Scenario B1 and B2 address the process of creating a runtime adaptable system, required for scenario 1 where Bob wants to define the AmI Home Care System for Mrs. Alice.

In the next sections, each scenario is analyzed on both the design decisions for the model from Chapter 5 and the runtime adaptability tool presented at the beginning of this chapter.

6.3.1 SCENARIO C1: RUNTIME ADAPTABILITY

Scenario C1 is the most important scenario from the runtime adaptability point of view, since it describes that the system allows adaptations at runtime. The scenario shows that by the explicit specification of variation points, the configurability of the system increases. The table below describes the context of the scenario:

Table 6-4: Analysis of Scenario C1

Scenario C1: Runtime Adaptability	
A runtime adaptable system is created with several variation points. When the system is started, the system engineer adapts the system at the variation points.	
Attributes	Configurability
Environment	Running system
Stimulus	The system engineer wants to change the behavior of the system. The system engineer takes a look at the runtime model, selects which variation point should change and commits that change.
Response	The system changes behavior without a rebuild or a restart.

Several design decisions are important for this scenario. These design decisions are analyzed to find sensitivities, tradeoffs, or risks for the design of the runtime adaptability models.

Table 6-5: Design decisions for the runtime adaptability scenario.

Design decisions	Sensitivity	Tradeoff	Risk
DD1: The required variability is specified in a feature model.		T1	
DD2: OR and XOR constraints are supported by the grouping of features.		T2	
DD3: The feature model allows five different types of variability, including parameterization.	S1		
DD4: A feature configuration is a clone of the feature model.		T3	

Analysis for the model

T1: A feature model is not the only way to specify the variability needed for runtime adaptability, other models could also suffice. Furthermore, feature models are normally used to capture the requirements of an application at a high level of abstraction. This might be a problem when fine-grained variation points need to be specified, because in that case both general features and very concrete implementation-based features are intertwined. This increases the complexity of the feature model.

However, a feature model is seen as a convenient way to talk to stakeholders and is used in most of the product line approaches [14, 31, 38]. Additionally, it allows for the explicit representation of variation points and dependencies, which is very important for defining runtime adaptability. Furthermore, feature models are implementation independent, allowing it to target any platform.

T2: The OR and XOR constraints are now expressed by using FeatureGroups. This is quite convenient since the feature tree keeps a tree form when using groups. Furthermore, variants for a variation point can directly be identified. An alternative for this approach would be adding single constraints between features to define which features are e.g. mutual-exclusive with each other. This would keep the feature model less complex, since extra model constraints such as the fact that a FeatureGroup cannot hold other FeatureGroups are not necessary in that case.

T3: The distinct separation allows for the separation of concerns and reduces therefore the complexity of implementing a runtime adaptable system. The variability can be defined in the runtime model and the functionality in the implementation. On the other hand, the separation induces more overhead due to the addition of an extra asset (the runtime model). For a single component, both the specification and the realization must be kept in synchronization with each other.

S1: The required variability is now defined as Mandatory, Optional, Alternative, Optional variant and Parameter. All these variability types can be expressed by the feature model. When other requirements arrive for variability, it might be possible that they cannot be expressed by the feature model presented in this thesis. Consequently, the resulting system will not have the required runtime adaptability.

Analysis of the runtime adaptability tool

When a system is configured with several open variation points, the model that is generated will still possess this variability. After the system is started, the system engineer can have a look at which points the system can be changed and what impact those changes will have on the system. The system engineer can now execute a change to adapt the system. The system will adapt accordingly, without recompilation or restarts, and the system engineer gets instant feedback on the new qualities of the system.

6.3.2 SCENARIO C2: EXTENSIBILITY

The extensibility scenario focuses on the extensibility of the system at runtime. This is for example required when new subsystems are added to the AmI Home Care System, such as additional iCups. The runtime model allows open variation points to be extended with additional variants, when they implement the same interfaces required by the variation point.

Table 6-6: Analysis of the Scenario C2.

Scenario C2: Extensibility	
A runtime adaptable system is running and a new component is detected. The system incorporates the new component and is able to instantiate it.	
Attributes	Configurability

Environment	Running system
Stimulus	The new component (variant) is added to a variation point
Response	The runtime model is updated and if necessary the component is instantiated

The following design decisions are relevant for this scenario:

Table 6-7: Design decisions for the Extensibility scenario.

Design decisions	Sensitivity	Tradeoff	Risk
DD7: The runtime model uses a component-based framework.			
DD9: The runtime model is compositional.			

Analysis for the model

Both design decisions are non-risks (no sensitivity, tradeoffs or risks are identified in Table 6-7). This is because a compositional model is required in order to support extensibility. The component-based aspect of the model dictates rules on how components should look like, in order to be part of the model.

Analysis of the runtime adaptability tool

The generated runtime model supports the addition of variants, but this functionality is not yet accessible in the GUI that is generated by the tool, due to limited time.

6.3.3 SCENARIO C3: MAINTAINABILITY

Research show that up to 80% of the total system costs is spent during maintenance [14, p. 5]. Therefore, reducing the costs of maintenance is an interesting business case. This scenario focuses on recalling a decision after the system has been deployed. Normally, recalling a decision requires a change in the original software and a redeployment of the system, which is quite expensive.

Table 6-8: Analysis of scenario C3.

Scenario C3: Maintainability	
The system engineer recalls a design decision after the system has been deployed.	
Attributes	Configurability
Environment	After deployment of the system
Stimulus	The system engineer recalls a design decision.
Response	The change can be easily made.

Table 6-9: Design decisions that influence the maintainability.

Design decisions	Sensitivity	Tradeoff	Risk
DD7: The runtime model uses a component-based framework.	S2		
DD9: The runtime model is compositional.	S3		

Analysis for the model

S2: When developing multiple products from a common asset base, a change due to the recall of a design decision in one of the common assets, is immediately available for all the products that use that common asset. This reduces the maintenance costs compared to redeveloping that asset for each of the products that use such functionality.

S3: When the recalled design decision is based on using another variant for a variation point, the system is able to adapt to that variant at runtime. As the runtime model is compositional, variant components can be replaced at runtime too, or new variants can be added to variation points. As these mechanisms do not require a redeployment of the complete system, they reduce maintenance costs. However, this maintainability only works to a certain extend. When e.g. changes are required in the (complex) code that handles the variability, maintenance is increased and redeployment is necessary.

Analysis of the runtime adaptability tool

Reducing the effort for these kinds of changes in the system is one of the goals of this thesis. The GUI generated by the runtime adaptability tool creates a view on the model that allows for easy selection of variants for a variation point, while the system is running.

6.3.4 SCENARIO O1: OPTIMIZATION

The optimization scenario addresses the optimization of adaptations, i.e. the ability to execute well-considered adaptations. The second scenario defined in Chapter 2 addresses this requirement for optimization in AmI systems.

Table 6-10: Analysis of the Scenario O1.

Scenario O1: Optimization	
The system engineer selects one of the qualities of the system and sees that one variant performs better than the currently running variant. He replaces the variant for an alternative with better utility.	
Attributes	Optimization
Environment	Running system
Stimulus	Optimize system based on utility
Response	The system is optimized without restarting the system.

The following design decisions are relevant for this scenario:

Table 6-11: Design decisions for the Optimization scenario.

Design decisions	Sensitivity	Tradeoff	Risk
DD5: The quality model is based on the COVAMOF meta-model.		T4	

Analysis for the model

T4: The quality model has been influenced by the meta-model for dependencies in COVAMOF. It allows the specification of dependencies on the system properties, the qualities and resources of the system. The meta-model defines a straight-forward design for the specification of impact on the system by means of a quality association. This model is simple and, as it has been tested in practice with good results [38, 39], also very functional.

Other models to specify impact on the system, such as Chameleon, are more complex. In the case of Chameleon, impact is specified on different levels, which increases the complexity. On the other hand,

it allows for complete specification of the impact, which is necessary for the predefined reconfiguration in the automotive domain.

Analysis of the runtime adaptability tool

The runtime adaptability tool has implemented a reduced version of the quality model defined in Chapter 5, as it can only specify quantitative impact by double-based numbers, and not other number types or qualitative descriptions.

However, quantified impacts can be used to find out if certain variants are better in certain contexts, which allows for well-considered decisions on adaptation. Furthermore, quantified impacts are computable, which is required for adaptive systems.

6.3.5 SCENARIO B1: EFFORT

The development of a runtime adaptable system increases the effort by the developer as it increases complexity considerably. This scenario analyses if the models are able to reduce this effort for the developer.

Table 6-12: Analysis of scenario B1.

Scenario B1: Effort for implementation	
A runtime model is being specified and the developer wants to develop the component implementations that realize the specification.	
Attributes	Buildability
Environment	Development time
Stimulus	The developer wants to develop the component implementations.
Response	This development can start immediately.

The design decisions that are relevant for this scenario are listed in Table 6-13.

Table 6-13: Design decisions for the effort for implementation.

Design decisions	Sensitivity	Tradeoff	Risk
DD6: There is a distinct separation between specification and realization.		T5	
DD8: The model supports a bottom-up approach.	S4		
DD10: The runtime model controls the implementations.			R1

Analysis for the model

T5: Separation between specification and realization offers several advantages. First, the implementation does not have to wait for the finishing of the specification. Second, the developer does not need to consider the complex adaptation code, which decreases the effort for the developer to implement an adaptable system. Downside is that the developer cannot specify variation points during the development of the actual realization of an asset; this has to be defined in the specification.

S4: As the whole process of generating an adaptable system is based on a top-down approach, the separation of specification and realization allows for independent programming of the functionality, and consequently for a bottom up-approach when developing component implementations. This allows

for parallel development and decreases the total time for implementing an adaptable system. Downside is that both specification and realization must be synchronized before the system can run.

R1: The runtime model controls the implementation by selecting which implementations need to run. All information about a variation point, its variants and the impact of those variants are defined in the specification. This means that a variant realization (component implementation) cannot change e.g. its impact on the qualities of the system or find out which other alternatives are available for that variation point. As this separation makes the implementation cleaner and consequently easier to implement (resulting in less effort for the developer), it might be possible that this functionality is needed in some applications. This requires that the transformation step from the adaptability model to the implementation skeletons must be changed and that the connection between specification and realization becomes tighter.

Analysis of the runtime adaptability tool

The runtime adaptability tool leverages the development of a runtime adaptable system by the use of the models and allows easy generation of the runtime model and implementation skeletons. This reduces the development effort for creating a runtime adaptable system considerable.

6.3.6 SCENARIO B2: OVERHEAD

The scenario for performance focuses on the overhead that the runtime model imposes on the complete system.

Table 6-14: Analysis of scenario B1.

Scenario B2: Overhead	
A runtime adaptable system is running. The CPU and memory overhead for the model should not render the application's utility zero.	
Attributes	Buildability
Environment	Running system
Stimulus	During runtime models have been loaded that describe the system.
Response	The system is not affected severely by these models.

The following design decisions are relevant for this scenario:

Table 6-15: Design decisions relevant for overhead.

Design decisions	Sensitivity	Tradeoff	Risk
DD2: OR and XOR constraints are supported by the grouping of features.	S5		
DD6: There is a distinct separation between specification and realization.	S6		

Analysis for the model

S5: Using grouping to capture OR and XOR constraints makes it easier to find out what the variants are for a specific variation point. Less CPU cycles are necessary to calculate the possible resolutions for a variation point, which reduces the runtime overhead.

S6: The separation between specification and realization introduces additional classes or code that need to be loaded at runtime. This requires more memory during runtime and additional CPU usage at startup. This decision decreases the complexity (see C1) at the cost of memory and CPU consumption.

Analysis of the runtime adaptability tool

The classes generated by the tool for the runtime model are quite small (around 30 lines of code per asset), since they only describe the implementation assets. As the functionality of an application comprises far more lines of code, the overhead in terms of memory can be neglected.

The CPU overhead is a little more compared to a system without the model, since the complete model must be instantiated at startup. Additionally, when compared to an ad-hoc based solution, such a solution will partially need the same information as described by the models, although more intertwined with the functionality of the component implementations. Furthermore, each adaptation causes the startup or shutdown of implementations, which require extra CPU cycles, memory allocation and garbage collection. As only adaptable systems are regarded here – and not adaptive systems – the overhead for finding the right variant for a particular variation point (which is a typical satisfiability problem (SAT) from complexity theory) is not measured.

6.4 CONCLUSION

The validation has shown that the models developed in Chapter 5 can be utilized in practice. This is shown by the runtime adaptability tool, which uses the models to generate a runtime adaptable system. Due to time constraints, the tool cannot meet all the functional requirements that are defined in the chapter on requirements, but the main functionality is realized.

The quality attribute analysis identified several tradeoffs, sensitivity points and risks based on the design decisions that are made during the models' development process and have impact on the configurability, optimization and buildability of the models.

Configurability

Feature models are a convenient way to define the adaptability of the resulting application. However, as feature models reside normally on a high level of abstraction, fine-grained variation points are intertwined with generic features, which increases the complexity of the feature model.

FeatureGroups allow for easier identification of variants for a variation point and keep the feature model in a tree-based form. However, FeatureGroups introduce multiple ways of specifying constraints between features. But the easiness of using FeatureGroups outweighs the more complex specification of alternative or optional variants by using several 'mutual-exclusive' and 'require' constraints.

A component-based design increases the configurability of a runtime adaptable system, because component implementations can be replaced by alternatives at runtime. Furthermore, the maintainability of the system is increased as e.g. bad functioning component implementations can be replaced at runtime by newly loaded component implementations, or the system is extended with new functionality.

Optimization

The design of the quality model defined in Chapter 5 is based on the simplicity and applicability of COVAMOF's model. It allows for a straight-forward specification of impact of features. A different model could be chosen here, such as the model defined by Chameleon, but these models do not have the described properties of the quality model in this thesis.

Buildability

The separation of specification (in the runtime model) and realization (in the component implementations) of adaptability reduces the complexity of implementing runtime adaptability. Consequently, specific parts of the process to develop runtime adaptability are moved to an earlier time in the development process of the complete application, as the software architect and system engineer are responsible for that. Furthermore, the separation allows for parallel development of the specification (top-down) and the realization (bottom-up). This process reduces the total time to develop applications.

The use of a runtime model increases the memory usage of the application, but as this overhead is just a small percentage of a complete functioning system, it won't render the application unusable. The same holds for increased CPU utilization during start-up of the application (because the runtime model must be instantiated first) and during adaptation.

A risk is that only the specification can control the realization of the application and not vice versa. Therefore is it not possible for component implementations to change their impact on the utility of the system when this changes during runtime or block a proposed adaptation by the runtime model manager. Making the relation between specification and realization bi-directional would resolve this situation, but it reduces the separation between specification and realization, too.

7 CONCLUSION

This thesis identified the need for modeling runtime adaptability and described the steps that are required to engineer adaptability. Using the input from an analysis of eight models in the area of adaptable systems and product lines, it distilled a model that enables the engineering of runtime adaptability. This model lowers the complexity by separation of concerns and identifies what knowledge is needed at runtime for software that requires adaptability. Furthermore, this model is quality-aware and allows the prediction of the impact of an adaptation. This allows well-considered adaptations by:

1. an operator of the system. The operator can see what the impact of changing a certain option will be, without actually changing the option. He can therefore optimize the change he wants to make;
2. an autonomous system. Describing the adaptability of a system is the first step for creating adaptive applications. An autonomous system uses the adaptability of the system to exhibit self-management behavior, needed in the application domain of AmI applications. As the impact of an adaptation can be predicted by the use of the quality model, an autonomous system can optimize its adaptations, leading to a system that can fulfill its requirements better.

Although this work emerged from the domain of Ambient Intelligent systems, this work can be used in other domains too. Since the modeling of runtime adaptability is based on concepts of the variability management domain, this thesis does a step towards closing the gap between development time variability and runtime variability. The ability to delay a design decision for a certain variation point as long as possible increases the flexibility of a software company towards its customers. Furthermore, revoking a decision later in the process is often very costly [13]. When the ability for delaying certain important decisions can also be extended to runtime, the costs for these product line members can be reduced.

Chapter 1 formulated several research questions for this thesis. Below the questions and their answers are given.

1. *What is runtime adaptability?* (Chapter 2)
   Runtime adaptability refers to the easiness in which an application can be changed, after it has been developed, deployed and started. Chapter 1 and 2 elaborate on runtime adaptability. There are different types of runtime adaptability, and the type addressed in this thesis is online-determined reconfiguration, which allows a system to determine at runtime which configuration of software components and parameters fits the requirements of the system best at a certain point in time. The development process of runtime adaptability has a lot in common with the development process of Software Product Lines, because they address the management of variability among different products in a product line, with the exception that the adaptation for SPLs takes place at development time and not at runtime. The idea of

runtime adaptability is to exploit previous defined variability during the development process at runtime. The information required for defining runtime adaptation is based on what can be adapted, where can it be adapted, how can it be adapted and what impact will an adaptation have on the system. This information must be documented in a model to enable adaptation at runtime. The question who will adapt is not relevant, since this thesis does not focus on adaptivity – the quality of a system to adapt itself –, but on runtime adaptability which is the quality of a system to be easily adapted at runtime.

2. *What other model-based approaches are available that can be used to engineer runtime adaptability?* (Chapter 3)

   Chapter 3 analyzed eight different approaches for software product lines and adaptable systems. The approaches are analyzed on how they define and use variability, the mechanisms for variability, how they model system qualities and resources and how they model impact on these qualities and resources. Furthermore, the analysis takes a look at how decisions are documented and if the approaches are applicable at runtime. Most approaches describe development time variability in meta-models and these meta-models are analyzed if they are usable for defining runtime adaptability. Only one approach, Madam, has a focus on runtime adaptability, but Madam's focus is on a middleware layer, which is different from an SPL-based approach. Therefore a new approach is required to develop runtime adaptable systems.

3. *What are the requirements for specifying a model-based approach for runtime adaptable systems?* (Chapter 4)

   Chapter 4 describes the requirements of for a model-based approach. The input of the requirements is based on the scenarios and the analysis of adaptation described in Chapter 2. Most important is that information about adaptation, such as what to adapt, where to adapt and with what impact, is documented in the model. This allows for analysis, simulation, and prediction of adaptation. As this information must be available at runtime, the process of developing runtime adaptable systems must allow for the specification of this information. This means that steps such as defining variation points, variants, constraints between different variants, describing impact of variants, etc. need to be incorporated in the process.

4. *How does a model for runtime adaptable system look like and how can this model be incorporated into a software engineering process.* (Chapter 5)

   Chapter 5 describes the adaptability model. This adaptability model is comprised of two models, a feature model and a quality model. The feature model describes the required variability in the runtime adaptable product. The quality model describes the qualities of the system and the impact of the features in the feature model on the qualities in the quality model. These models allow now for analysis, validation, simulation and prediction of adaptation at runtime, and consequently for well-considered adaptations.

The models only address the specification of the required adaptability, and therefore these models need to be transformed to models that can be used at runtime. This conversion process shows how the specification of adaptation can be converted to a runtime model in code that describes how the application currently looks like, where its variation points are, which variants are available for a variation point and what impact each of the variants has on the utility of the application. Thus, this runtime model describes the runtime adaptability of the running application.

Each variation point and variant links to a component implementation that implements the functionality described by the feature in the specification. As each component implementation implements a specific interface that allows them to be adaptable, the runtime model is able to reconfigure the system by starting up or shutting down component implementations. As each impact of a component implementation is documented in the quality model, well-considered decisions can be made when the system requires an adaptation at runtime.

Because all the information about adaptation must be available at runtime, the software engineering process has to provide this information during the development process. This requires the system architect and system engineer to specify and define variation points and variants early in the development process.

5. *Does the model-based solution satisfy the requirements?* (Chapter 6)
   The models provides the required variability needed to specify runtime adaptability. However, the models are defined on a very abstract level (i.e. it is quite generic), which results in the fact that validating the completeness, correctness, buildability and its conceptual integrity is difficult. Therefore, usually systems are first built from the models (e.g. a prototype) and then tested if the system requirements are met. As the models described in this thesis are not yet tested in the BelAmI-project, it is difficult to measure if the models leverage the creation of runtime adaptable systems. Consequently has the validation in Chapter 6 its limits.

   The models are validated by analyzing several design decisions that were taken during the process of creating the models. This analysis identifies sensitivity points, tradeoffs and risks that can result in poor design. The validation analyzed the configurability (by analyzing runtime adaptability, extensibility and maintainability), the optimization and buildability (by analyzing the effort and overhead induced by the models). Most design decisions influence the defined quality attributes positively. One risk identified that the uni-directional relationship between the runtime model and the component implementations might be converted to a bi-directional relation, as this allows the component implementations to block intended adaptations and allow them to change the impact of their own implementation at runtime.

Furthermore, a prototype tool is developed which assist the requirements engineer, software architect, system architect and software developer with the specification and development of runtime adaptable systems. It implements the process described in Chapter 5 and generates a runtime model and component implementation skeletons for a runtime adaptable system by using the specification of adaptability. This tool therefore greatly reduces the complexity of creating a runtime adaptable system. Furthermore, it shows that the models defined in Chapter 5 can be converted to code and can be used to create runtime adaptable systems. It consequently increases the buildability of a runtime adaptable system.

8 FUTURE WORK

Several shortcomings have been identified during the validation of the work in this thesis, but also by the scoping of this thesis. This chapter elaborates on these shortcomings and gives some guidance on how these shortcomings can be resolved by further research.

8.1 GENERALLY

The research on runtime adaptable systems will continue, as even more complex systems will be created in the future. This complexity is driven by improvement in microprocessor technology that is able to support users better than ever before, as devices get smaller, faster and cheaper. The realization of the software requirements imposed on systems that integrate these devices – especially the non-functional requirements – will be much harder, which requires the software engineering process to improve and the tools supporting this process to be smarter. Furthermore, the management of these complex systems will be more difficult to realize, because people who normally manage these systems need more education to understand the complexity, and to alleviate this (more) automation is required in this area, which consequently requires runtime adaptability.

This thesis described models and tools that leverage the development of runtime adaptable systems by abstraction. As abstractions focus general solutions, a specific solution based on these abstractions might not fulfill the requirements on which the abstractions are based as details are lost during the abstraction process. The abstractions might therefore not cover the complete intended solution space (see the design pyramid in Chapter 1). They consequently 'leak' (see Joel Spolsky's remark on http://joelonsoftware.com/articles/LeakyAbstractions.html). This means that the models and especially the tools defined in this thesis will not provide a solution in every area where runtime adaptation is necessary. More research is necessary in those areas.

8.2 SPECIFICALLY

To fully assess the applicability of this work, it is required that a real application is developed and validated. During this process new requirements might surface that need to be addressed in the models. A few of these requirements might be:

- Analysis of an adaptation transaction.

  The validation identified that component implementations might not be able to carry out a proposed adaptation. If this is the case, all steps already taken in an adaptation transaction, must be rolled back and the system must find a better adaptation. This process is very tricky, especially when adaptation occurs in parallel processes.
- Selecting the right variant.

The reasoning required for autonomous selecting the correct, best-fitting variant for an adaptation was out of the scope of this thesis, but is very complex and requires more research on how to adapt efficiently.

- Specifying impact.

  The runtime adaptability tool only supports the specification of impact by the use of rational numbers (doubles). More ways of specifying impact are required for better reasoning about adaptations. Examples are enumerations (e.g. 'good', 'neutral', and 'bad'), integers and more complex objects.

  Furthermore, the tool can only specify the impact on qualities by numbers, not formulas. Formulas, that combine different parameters and variants, are more expressive and allow for a more fine-grained impact specification.

  The COVAMOF approach specifies qualitative impacts in the form of documented knowledge and tacit knowledge. Although these types are difficult to compute (if not at all), they have greatly improved the product derivation process in practice.

- Predicting impact.

  For several quality attributes it is not possible to define the impact on forehand. Both Madam and COVAMOF allow the architect to specify predictor functions that are able to predict the impact, based on earlier experience. Such functions allow a more stable prediction of impact of adaptation.

- Specifying constraints.

  The constraints defined in the model allow for the specification of mutual-exclusive and requires constraints. The runtime adaptability tool does not support these constraints yet. When these are added, functionality for testing if combinations of features are possible should be added too, as this will assist the system engineer to define product that can exist in practice.

- Separation of concerns.

  The separation of concerns leverages the simplicity of the implementation, as the component implementations and the specification of adaptation are separated. The design decision (DD10) keeps this separation as wide as possible by allowing only the runtime model to change the configuration of the product. This might be a problem when component implementations need to change something in the model (e.g. impact on the quality attributes) at runtime, as this relation is not available.

- Applicability in other SPL-approaches.

  Companies that are already using an SPL approach (e.g. by using KobrA or COVAMOF) might be interested in delaying their design decisions to the runtime or provide runtime adaptability in their products. Further research must show if the process, models and tools developed in this thesis can be adopted by these companies' software engineering processes.

- Crosscutting features.

  A paradigm that has not been discussed in this thesis is aspects. As features, such as security in general and energy saving in specific, are often crosscutting and cannot be projected in one single component implementation, aspect weaving is nowadays used to address these

crosscutting features. More research is necessary to identify if aspects can be incorporated in this approach.

9 REFERENCES

1. Aarts, E., Harwig, E., and Schuurmans, M.: Ambient Intelligence. In Denning, J. (edt.): *The Invisible Future*, McGraw-Hill, New York, p. 235-250, 2001.
2. Alexander, C.: *The Timeless Way of Building*. Oxford University Press, 1979.
3. Atkinson, C., Bayer, J., and Muthig, D.: Component-Based Product Line Development: The KobrA Approach. In *Proceedings of the First Software Product Line Conference (SPLC-1)*, Denver, Colorado, August 2000.
4. Atkinson, C., Bayer, J., Laitenberger, O., Zettel, J.: Component-Based Software Engineering: The KobrA Approach. *Workshop on Component-Based Software Engineering, International Conference on Software Engineering*, Limerick, Ireland, June 4-11, 2000.
5. Avgeriou, P., Zdun, U.: Architectural Patterns Revisited – A Pattern Language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, Irsee, Germany, July 2005.
6. Bass, L., Clements, P. and Kazman, P., *Software Architecture in Practice*. Addison Wesley, 1998.
7. Baum, L., Becker, M., Geyer, L., Molter, G.: Mapping Requirements to Reusable Components Using Design Spaces. *Proc. of IEEE Int'l Conference on Requirements Engineering (ICRE) 2000*: pp. 159-167.
8. Becker, M., Patzke, T., Anastasopoulos, M.: Software Product Line Technology for Ambient Intelligence Applications, *Proceedings of the MAC Workshop, Net.ObjectDays 2005*, Erfurt, September 2005.
9. Becker, M., Werkman, E., Anastasopoulos, M., Kleinberger, T.: Approaching Ambient Intelligent Home Care Systems. In *Proceedings of PervasiveHealth Conference*, Insbruck, Austria, Nov. 2006.
10. Becker, M.: *Anpassungsunterstützung in Software-Produktfamilien* (in German). Schriftenreihe / Fachbereich Informatik, Technische Universität Kaiserslautern, 2004.
11. Becker, M.; Decker, B.; Patzke, T.; Syeda, H.A.: *Runtime adaptivity for AmI systems - the concept of adaptivity*, IESE-Report, 091.05/E, October 2005.
12. BelAmI-project website: http://www.belami-project.org/.
13. Bosch, J., Andersson, J., Development and use of dynamic product-line architectures, *IEE Proc.-Softw.*, Vol. 152, No. 1, February 2005
14. Bosch, J.: *Design & Use of Software Architectures - Adopting and evolving a product-line approach*, ISBN 02016749947, Addison-Wesley, 2000.
15. Clements, P. and Northrop, L.: *Software Product Lines: Practices and Patterns*, Addison Wesley Longman, Reading, Mass., 2001.
16. Clements, P. et al., *Documenting Software Architectures – Views and Beyond*, Pearson Education, 2003.

17. Clements, P., Kazman, R., Klein, M.: *Evaluating Software Archtectures: Methods and Case Studies*, Addison Wesley, 2002.
18. Czarnecki, K., Eisenecker, U.: *Generative Programming: methods, tools, and applications*. Addison-Wesley, 2000.
19. Decker, C., Krohn, A., Beigl, M., Zimmer, T.: The Particle Computer System, IPSN Track on Sensor Platform, Tools and Design Methods for Networked Embedded Systems (SPOTS), *Proceedings of the ACM/IEEE Fourth International Conference on Information Processing in Sensor Networks (IPSN05)*, Available online at http://www.teco.edu/~michael/publication/spots2005.pdf.
20. Dijkstra, E.W.: *A Dicipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
21. European Union report, Scenarios for Ambient Intelligence in 2010, Available online at ftp://ftp.cordis.lu/pub/ist/docs/istagscenarios2010.pdf
22. Floch, J., Hallsteinsen, S. O., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Using Architecture Models for Runtime Adaptability. *IEEE Software* 23(2): 62-70 (2006)
23. G. E. Moore, Cramming more components onto integrated circuits, *Electronics*, vol. 38, pp. 114--117, 1965.
24. Geyer, L.: Feature Modelling Using Design Spaces, *Proceedings of the 1st German Workshop on Software Product Lines*, Kaiserslautern, November 2000.
25. Jacobson, I., Griss, M., Jonsson, P., *Software Reuse – Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.
26. Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. & Peterson, A. S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical report CMU/SEI-90-TR-21, ADA 235785, 1990.
27. Kang, K. C., Kim, S., Lee, J., Kim, K., Kim, G. J., Shin, E.: FORM: A Feature-Oriented Reuse Method with Domain-specific Architectures, *Annals of Software Engineering*, V5, pp. 354-355, 1998.
28. Kephart, J. and Chess, D.: The Vision of Autonomic Computing, *IEEE Computer Magazine*, IEEE, January 2003.
29. Krueger, C.: Using Separation of Concerns to Simplify Software Product Family Engineering. April 2001. *Proceedings of the Dagstuhl Seminar No. 01161: Product Family Development*, Wadern, Germany.
30. Mozilla.org, XML User Interface Language (XUL) 1.0, website: http://www.mozilla.org/projects/xul/
31. Muthig, D.: *A Lightweight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. Stuttgart: Fraunhofer IRB Verlag, 2002
32. Nigel Shadbolt, Ambient Intelligence, *IEEE Intelligent Systems*. July/August 2003, pp. 2-3
33. Object Management Group: *Model Driven Architecture (MDA)*. Technical Report Document number ormsc/2001-07-01, Object Management Group, July 2001.
34. OMG Group, Unified Modeling Language, website: http://www.uml.org/

35. Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., Wolf, A. L.: An Architecture-Based Approach to Self-Adaptive Software, *IEEE Intelligent Systems* 14(3), pp 54–62, 1999.

36. Schmid, K., John, I., Generic Variability Management and its application to Product Line Modeling, In *proceedings of the 1st Workshop on Software Variability Management (SVM02)*, Groningen, Netherlands, February 2003.

37. Shaw, M., Garlan, D.: *Software Architecture – Perspectives on an Emerging Discipline*, Prentice Hall, 1996

38. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: COVAMOF: A Framework for Modeling Variability in Software Product Families, *Proceedings of the Third Software Product Line Conference (SPLC 2004)*, Springer Verlag Lecture Notes on Computer Science Vol. 3154 (LNCS 3154), pp. 197-213, August 2004.

39. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Modeling Dependencies in Product Families with COVAMOF, *Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006)*, March 2006.

40. Sofware Engineering Institute (SEI), Software Product Lines, website: http://www.sei.cmu.edu/productlines/

41. Sommerville, I., *Software Engineering*. Addison Wesley, Reading, MA, 2001. 6th edition.

42. Tekinerdoğan, B., Akşit, M., Adaptability in Object-Oriented Software Development, Workshop report, in *Special Issues in Object-Oriented Programming*, M. Mühlhauser (Ed.), dpunkt verlag, pp. 7-11, 1996

43. Trapp, M.: *Modeling the Adaptation Behavior of Adaptive Embedded Systems*. Dissertation University of Kaiserslautern, July 2005.

44. van Gurp, J., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines, *Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*, 2001, pp. 45.

45. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. *IEEE Computer*, Vol. 33, Nr. 3, March 2000, 78--85.

46. Weiser, M. The Computer for the Twenty-First Century, *Scientific American*, pp. 94-10, 1991.

Appendix A: ADAPTABLE VERSUS ADAPTIVE

There is some confusion about the difference between adaptable and adaptive. The source for this ambiguity is mainly the English language itself, as different dictionaries give the same results when there should be a difference. At least, from a non-English language point of view. Languages such as German and Dutch have a richer vocabulary when it comes to words describing adaptable (e.g. anpasbar (G) or aanpasbaar (D)). Translating these words to English will render different results when different dictionaries are used.

Asking an English friend for clarification, she replied in similar confusion: "[..] like a lot of words the original meanings were lost and English people have no idea how to use their own language so use both words for the same thing!" (...).

Trying to clarify anyways, the overview below gives translations for the words 'adaptable' and adaptive' by several online English dictionaries:

Table A-1: Translations for the word 'adaptive' by several dictionaries.

Dictionary [reference]	Translation of adaptive
Websters Dictionary [1]	Suited, given, or tending, to adaptation; characterized by adaptation; capable of adapting.
Websters Online Dictionary (The Rosetta Edition) [3]	Having a capacity for adaptation.
Dictionary.com (The American Heritage Dictionary of the English Language, Fourth Edition) [5]	1. Relating to or exhibiting adaptation. 2. Readily capable of adapting or of being adapted.
Merriam-Webster Online Dictionary [7]	Showing or having a capacity for or tendency toward adaptation
Encarta [9]	usable in different conditions: able to be adjusted for use in different conditions
Cambridge Dictionary Online (Cambridge Advanced Learner's Dictionary) [11]	possessing an ability to change to suit different conditions
Wordsmyth [13]	Capable of or suitable for adapting.

Table A-2: Translations for the word 'adaptable' by several dictionaries

Dictionary [reference]	Translation of adaptable
Websters Dictionary [2]	*Capable of being adapted.*
Websters Online Dictionary (The Rosetta Edition) [4]	Capable of adapting (of becoming or being made suitable) to a particular situation or use.
Dictionary.com (The American Heritage Dictionary of the English Language, Fourth Edition) [6]	*Capable* of adapting or *of being adapted.*
Merriam-Webster Online Dictionary [8]	*Capable of being* or becoming *adapted*

Encarta [10]	1. changing easily: able to adjust easily to a new environment or different conditions 2. adjustable: *capable of being modified* to suit different conditions or a different purpose
Cambridge Dictionary Online (Cambridge Advanced Learner's Dictionary) [12]	Able or willing to change in order to suit different conditions
Wordsmyth [14]	1. having the *capacity to be adapted*. 2. having the ability to adjust to particular needs or conditions.

The overview shows that there is no consensus on the translations, although adaptable tends to be translated to "capable of being adapted" (printed in italics in the table) by several dictionaries. This means that an adaptable system is adapted by an external force, and not by itself. This also supports the understanding of the word adaptable in other languages.

Researchers also identified this problem and provided several definitions for clarification. An overview can be found in [16] and [17]. The definition used in this thesis is that of Czarnecki et al. [15, p. 397]:

> "*Adaptable* systems can be adapted to a particular deployment environment, whereas *adaptive* systems adapt themselves to a deployment environment."

REFERENCES

Referenced on February 16, 2007.

1. http://www.webster-dictionary.net/definition/Adaptive
2. http://www.webster-dictionary.net/definition/Adaptable
3. http://www.websters-online-dictionary.org/definition/adaptive
4. http://www.websters-online-dictionary.org/definition/adaptable
5. http://dictionary.reference.com/search?q=adaptive (The American Heritage® Dictionary of the English Language, Fourth Edition)
6. http://dictionary.reference.com/search?q=adaptable (The American Heritage® Dictionary of the English Language, Fourth Edition)
7. http://www.m-w.com/dictionary/adaptive
8. http://www.m-w.com/dictionary/adaptable
9. http://encarta.msn.com/dictionary_1861583584/adaptive.html
10. http://encarta.msn.com/dictionary_1861583580/adaptable.html
11. http://dictionary.cambridge.org/define.asp?key=944&dict=CALD (Cambridge Advanced Learner's Dictionary)
12. http://dictionary.cambridge.org/define.asp?key=944&dict=CALD (Cambridge Advanced Learner's Dictionary)

13. http://www.wordsmyth.net/live/home.php?script=search&matchent=adaptive&matchtype=exact

14. http://www.wordsmyth.net/live/home.php?script=search&matchent=adaptable&matchtype=exact

15. Czarnecki, K., Eisenecker, U.: *Generative Programming: methods, tools, and applications*. Addison-Wesley, 2000.

16. Becker, M.; Decker, B.; Patzke, T.; Syeda, H.A.: *Runtime adaptivity for AmI systems - the concept of adaptivity*, IESE-Report, 091.05/E, October 2005.

17. Klamar, S. Adaptive Architekturen für verteilte Systeme, (in German), Master thesis at the University of Dresden, 2004, http://sebastian.klamar.name/pub/Diplom/Diplom.pdf

Appendix B: SCREENSHOTS OF THE TOOL

This appendix shows some screenshots of the runtime adaptability tool to get an idea how the model is created and used.

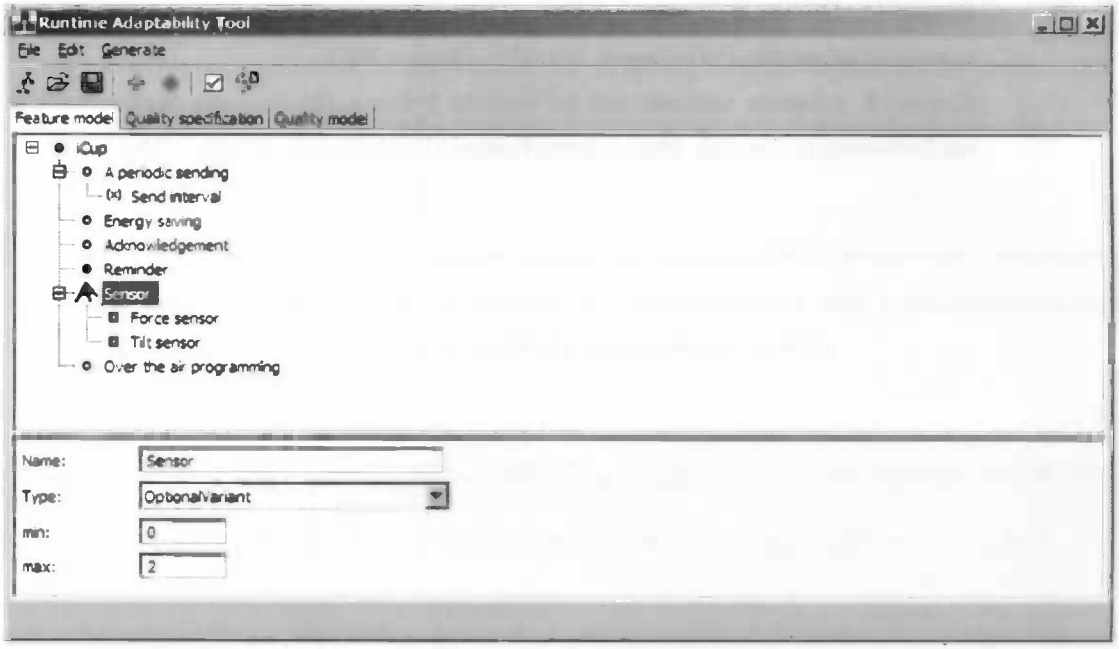


Figure 9-1: Screenshot of the tool's main window. It contains three tabs: (i) a tab for the definition of the feature model (shown), (ii) a tab for the definition of the quality model, and (iii) a tab for specifying the influence of features on the qualities of the system.

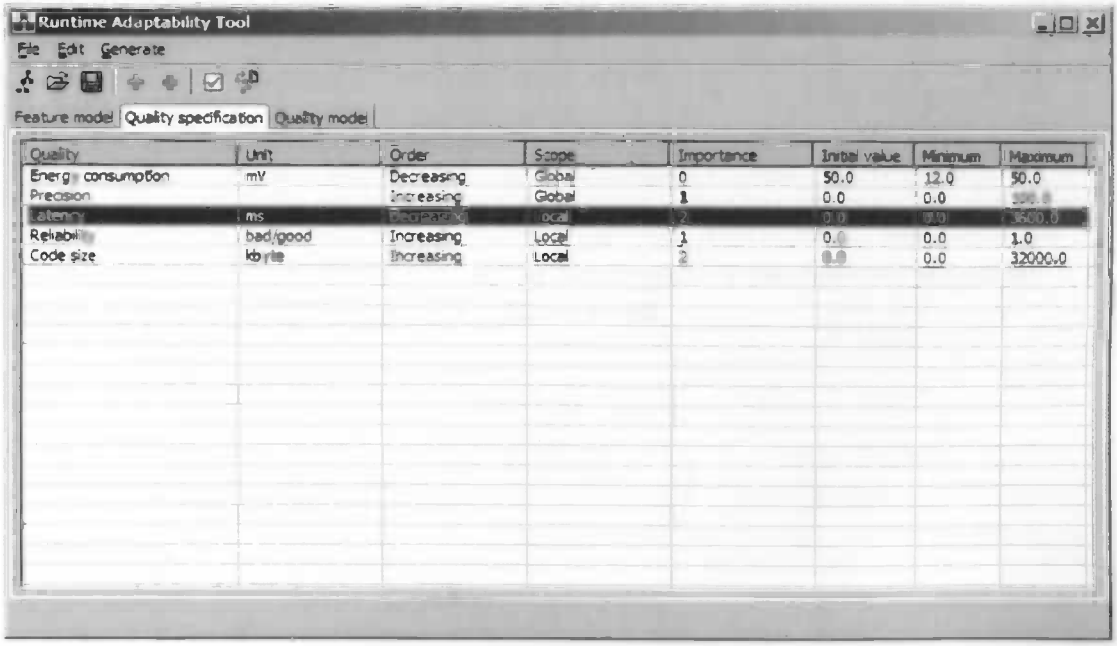


Figure 9-2: Quality specification in the tool.

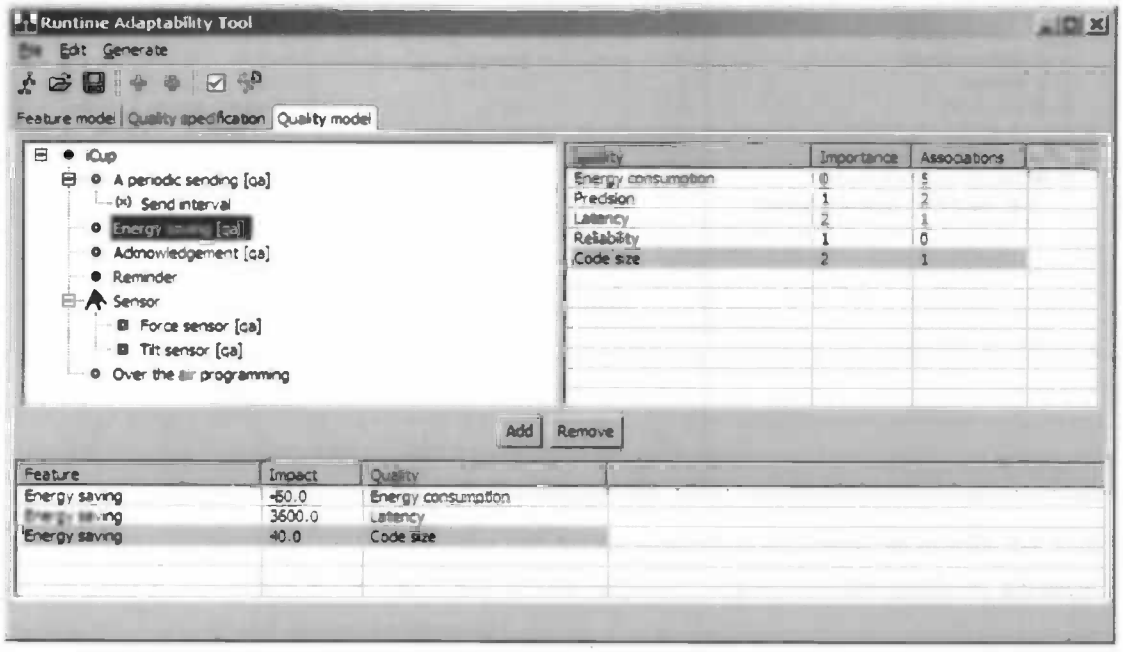


Figure 9-3: Specification of impact from features (on the left) on qualities (on the right). Every feature that is suffixed with [qa] has quality associations.

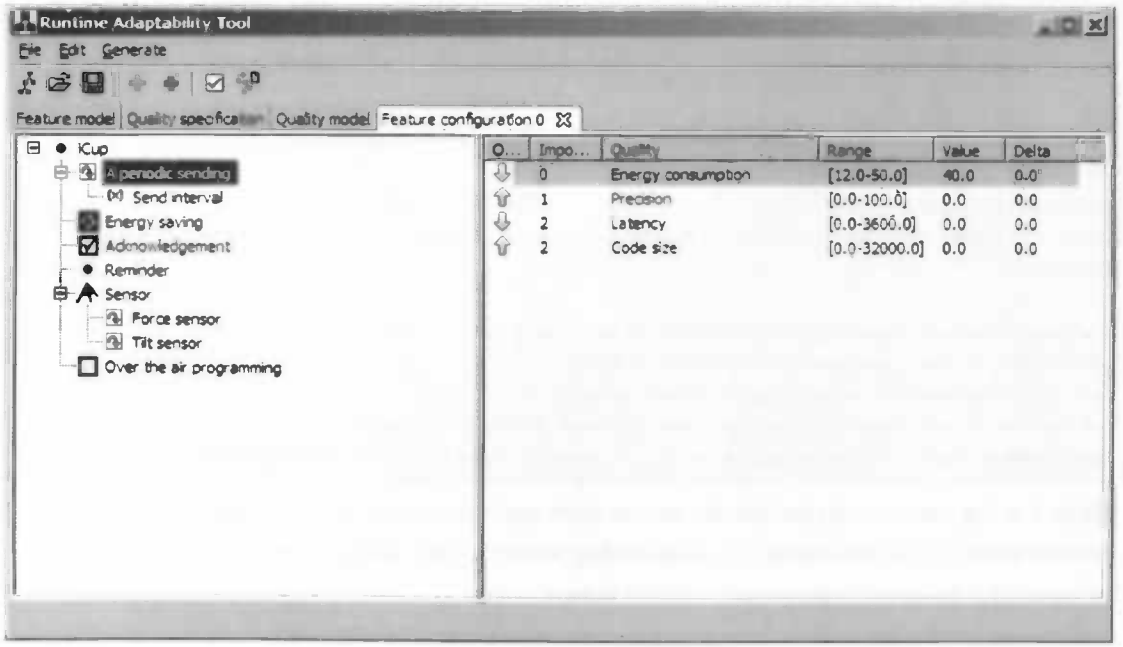


Figure 9-4: Feature configuration for the model. In this view the system engineer can choose on the left which features should be in the product (checked), and which decisions should be delayed to the runtime (round arrow). At the same time the impact on the system is shown on the right. When the system engineer is complete, the runtime model and implementation skeletons can be generated.

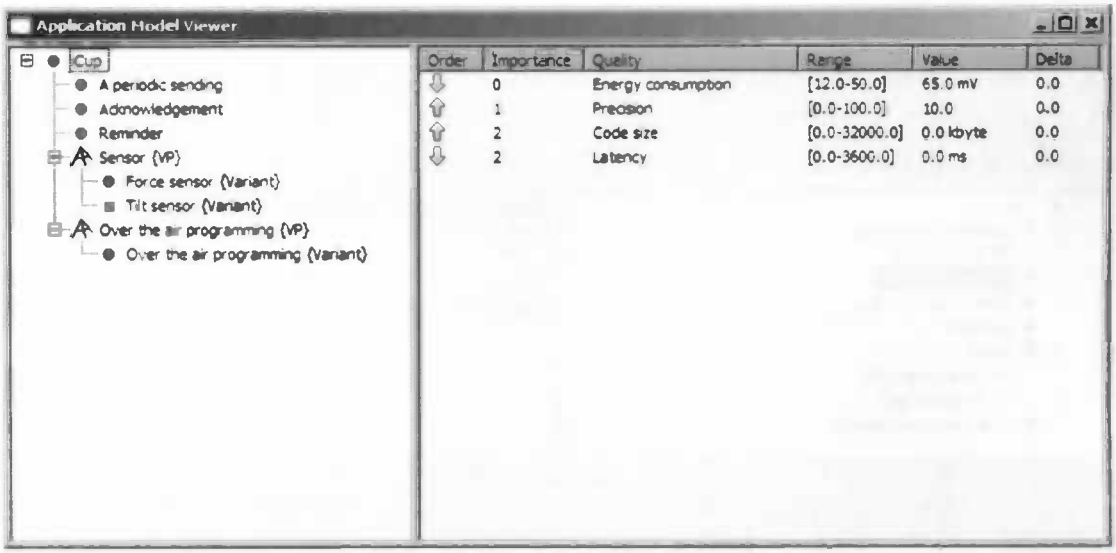


Figure 9-5: View of the model after is has been started. Red squares show that those implementations are not running (e.g. the Tilt sensor). A green circle means that the implementation is running.

```
Resolving APeriodicSending(Variation point) resolution: [],minResolutionSize=0,maxResolutionSize=1
AperiodicSendingVariant (Variant) 0
Added 1 variants for APeriodicSending(Variation point) resolution: [AperiodicSendingVariant
(Variant)]
realization.aperiodicsending.APeriodicSendingImpl@187aeca is running


Resolving Sensor(Variation point) resolution: [], minResolutionSize=1, maxResolutionSize=3
TiltSensor (Variant) 150
ForceSensor (Variant) 100
Added 2 variants for Sensor(Variation point) resolution: [TiltSensor (Variant), ForceSensor
(Variant)]

realization.sensor.SensorAdaptable@4f1d0d is getting adapted. (current: [])
realization.sensor.TiltSensorImpl@c3c749: startUp()
realization.sensor.ForceSensorImpl@12b6651: startUp()
realization.sensor.SensorAdaptable@4f1d0d has been adapted. (adapted to:
[realization.sensor.TiltSensorImpl@c3c749, realization.sensor.ForceSensorImpl@12b6651])
```

Figure 9-6: Log output excerpt just after the start-up of the application, when the model is instantiated. Open variation points such as the optional 'Aperiodic sending' feature and the 'Sensor' feature are resolved and subsequently is the correct implementation of these features started.

Appendix C: GLOSSARY

Acronym	Explanation
ADL	Architectural Description Language
AmI	Ambient Intelligence
ATAM	Architecture Tradeoff Analysis Method
BelAmI	Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems
ConIPF	Configuration of Industrial Product Families
COVAMOF	ConIPF Variability Modeling Framework
GUI	Graphical User Interface
LAN	Local Area Network
MDA	Model-driven Architecture
MDE	Model-driven Engineering
OMG	Object Management Group
SPF	Software Product Family
SPL	Software Product Line
UI	User Interface
UML	Unified Modeling Language
XML	Extensible Markup Language
XUL	XML User Interface Language