Masterarbeit

Design and Implementation of a Variant Rich Component Model for Model Driven Development

Nicolas Pavlidis

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Betreuerin: Dipl.-Ing. Andrea Leitner
Begutachter: Dipl.-Ing. Dr. techn. Christian Kreiner

Graz, im Oktober 2011

Kurzfassung

Automobile Softwareentwicklung sieht sich seit jeher mit der steigenden Komplexität der zu entwickelnden Software konfrontiert. Modellbasierte Entwicklung trägt dazu bei, diese Komplexität zu reduzieren und beherrschbar zu gestalten. Allerdings zieht steigender Variantenreichtum in der Software wiederum steigende Komplexität nach sich. Das Ziel muss es also sein auch diese Komplexität zu reduzieren. Software Product Lines bieten hierfür einen systematischen Ansatz indem Variabilität, in den der Software zu Grunde liegenden Modellen, explizit gemacht wird. Um daraus ein konkretes Modell ableiten zu können, muss eine, dem Kontext entsprechende Konfiguration gewählt werden. Eine solche Konfiguration kann dazu führen, dass ganze Teile des Basismodells entfernt werden müssen.

Die vorliegende Arbeit beschäftigt sich mit dem Problem des konfigurationsbasierten, automatischen und konsistenten Reduzierens von Modellen. Das konkrete Anwendungsprojekt HybConS beschäftigt sich mit der Entwicklung einer generischen Softwarearchitektur für hybride Fahrzeuge. In der Autoindustrie wird Simulink als primäres Werkzeug zur modellbasierten Entwicklung eingesetzt. Basierend auf solchen Simulink Modellen wurden drei Szenarien zur Anwendung von Variabilität identifiziert: 1) Alternative Implementierungen, 2) Optionales Verhalten und 3) Anwendung der genannten Szenarien auf Zustandsautomaten. Diese Szenarien und die Möglichkeit Modelle automatisiert auf spezifische Varianten reduzieren zu können bilden den grundsätzlichen Anforderungskatalog dieser Arbeit.

Ergebnis der Arbeit ist ein Komponentenmodell, genannt *Variant Component Model*, welches die Modellierung variabler Software Komponenten erlaubt. Um die automatische Reduktion von Modellen auf spezifische Varianten zu ermöglichen, verfolgt das in dieser Arbeit beschriebene Komponentenmodell die Strategie, bestehende Modelle derart zu manipulieren, dass sie nur mehr variantenspezifische Elemente enthalten.

Basierend auf dem entwickelten Komponentenmodell bietet die vorliegende Arbeit eine prototypische Implementierung, die Simulink Modelle in eine dem Komponentenmodell entsprechende Repräsentation überführt und die geforderte Reduktion auf spezifische Varianten erlaubt. Zur Vereinfachung der Modellierung von Variabilität in Simulink wurde eine Bibliothek mit hierfür spezifischen Blöcken entwickelt. Die bereits erwähnte Reduktion gegebener Simulink Modelle wird durch das Generieren von Matlab Scripts erreicht, die die nötige Information über zu entfernende Elemente enthalten.

Die vorliegende Implementierung ermöglicht schließlich die Umsetzung aller drei genannten Szenarien. Damit ist es nun möglich 1) Simulink (und andere) Modelle mit genügend Variabilitätsinformation anzureichern um diese 2) anschließend auf spezifische Varianten reduzieren zu können.

Abstract

Automotive software development was ever since faced with raising complexity of the developed software. Model based development helps here to reduce this complexity. On the other hand variability in the software adds additional complexity to it. Software Product Lines provide a systematic approach to reduce this complexity by adding additional variability information to the models, thus making variability explicit in these models. In order to derive concrete products from these variant rich models, a concrete configuration has to be selected. Based on this selection it may happen that large parts of the model need to be removed. This thesis addresses the problem of automatically and consistently reducing models according to a selected configuration. The HybConS project, being the primarily domain of this thesis, proposes a generic software architecture for hybrid vehicles. In the automotive domain Simulink is the dominant tool for model based development. Based on Simulink models three scenarios for the application of variability have been identified: 1) alternative implementations, 2) optional behavior and 3) applying those two scenarios on state machines. These three scenarios and the ability to automatically reduce models to specific variants provide the basic requirement catalog of this thesis. The result is a new component model, called *Variant Component Model*, that provides the ability to model variant rich software components. To achieve automatic reduction of models the component model follows the approach of removing elements from the model that do not belong to a specific variant.

Based on this component model, a prototypical implementation is provided that maps Simulink models to the *Variant Component Model* and reduces these models to specific variants identified by provided configurations. Furthermore, *Variant Modeling*, a library for Simulink has been developed, that provides blocks that help to enrich Simulink models with variability information. The reduction step is accomplished by generating Matlab scripts that contain information about the elements that need to be removed. Using this implementation it is possible to successfully cover all required scenarios. Therefore, it is now possible to 1) enrich Simulink models with variability information and 2) to reduce these models to specific variants.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
13.10. 2011	(signature)

Contents

List of Figures

List of Tables

List of Acronyms

AE	Application Engineering
API	Application Programing Interface
AOP	Aspect Oriented Programing
AST	Abstract Syntax Tree
AUTOSAR	AUTomotive Open System ARchitecture
CCM	CORBA Component Model
COM	Component Object Model
CVL	Common Variability Language
DD	Data Dictionary
DE	Domain Engineering
DSL	Domain Specific Language
ECU	Electronic Control Unit
EJB	Enterprise Java Beans
FODA	Feature Oriented Domain Analysis
IDE	Integrated Development Environment
JAXB	Java API for XML Binding
MDD	Model Driven Development
MOF	Meta Object Facility
OMG	Object Management Group
RTE	Run Time Environment
SWC	SoftWare Component
UI	User Interface
UML	Unified Modeling Language
SDP	Software Development Process

SPL	Software Product Line
SPLE	Software Product Line Engineering
VCM	Variant Component Model
VDM	Variant Description Model
VP	Variation Point
VPM	Variation Point Model
VPV	Variation Point Value
XML	eXtensible Markup Language

Acknowledgments

This thesis has been carried out at the Institute for Technical Informatics at Graz, University of Technology.

I want to thank my supervisors Andrea Leitner and Christian Kreiner for giving me the opportunity to carry out this thesis. In endless discussions, regarding the different design stages of this thesis, they always helped me to not get off the track and to constantly improve my work. By providing fast deployed reviews on the various drafts of this thesis they also helped to constantly improve the written part of my work.

I want to thank my parents, Barbara and Michael, for giving me the opportunity to carry out my studies. Their continuous support has let me concentrate on my work. I want to thank my sister Laura for having an open ear when I needed one.

I want to thank Marina Janisch for her continuous positive thinking especially, when I was getting too "realistic" (in terms of pessimism).

Last, but certainly not least, I want to thank my friends for encouraging and supporting me during my studies and during preparation of this thesis.

Nicolas Pavlidis
Graz, Austria, October 2011

1. Introduction

Components [Szyperski, 1997] are black boxes that define a publicly available interface to either send information to or to retrieve information from it. When it comes to configuration of such components the same behavior is desirable. Configuration information should be provided via a dedicated port to the component. The job of processing this configuration is left to the component itself. The component behaves variable. Using techniques known from Software Product Line Engineering, namely Domain Engineering and Application Engineering, should help to model such components.

1.1. Problem Scope and Definition

Complexity of automotive software grew since the first piece of software was introduced into cars [Broy, 2006]. Model based development helps to reduce complexity during software development. Beside the complexity of the software itself the complexity of handling different configurations grew. Introducing systematic management of different configurations based on a common solution family and systematic reuse of that solution family is therefore reasonable. Since such an approach does not affect only the deployment phase of a software project [Pohl et al., 2005], e.g. "installing" the software into a car, an approach is needed that provides a solution for all phases in software development.

1.1.1. The HybConS Project

The HybConS project was founded in cooperation of The Virtual Vehicle Competence Center[1], AVL[2] and the Institute for Technical Informatics located at Graz University of Technology. The aim of this project is to introduce a generic software architecture and, therefore, systematic reuse into a software project implementing control software for hybrid electrical vehicles. A generic architecture is desirable because of the various variants of hybrid electrical vehicles which turn out to be very similar.

The software is implemented using Simulink, a model based development environment provided within Matlab. Simulink models have a hierarchical structure consisting of so called *Blocks*. Blocks can be grouped into so called *Subsystems*. This can be used to store a group of connected blocks that implement a certain functionality into one named entity. The version of Simulink used for development does not provide mechanisms to model variability of such models explicitly. Although this has changed with the introduction of *Variant Subsystems* in Simulink's 2011b release one problem still remains: The ability to reduce models to those parts a specific configuration requires. Therefore, the first goal of this thesis is to provide a mechanism that can reduce Simulink models based on a provided selection to only include those parts implied by this configuration. Since Simulink

[1] `http://vif.tugraz.at`
[2] `https://www.avl.com/austria-headquarters`

is only one possible environment for doing model based development the second goal of this thesis is to abstract the desired mechanism to provide a generic representation of a *variable* component model.

1.2. Motivation

One of the goals of this thesis is to make variability explicit. Before thinking about how this can be achieved it is necessary to identify how variability can occur and which impact these occurrences have on the implemented model. In the HybCons project different scenarios were identified on how variability may be applied to Simulink models. Details are provided below. Each scenario consists of a description, its impact on the models that apply it and a brief explanation of the desired behavior.

1.2.1. Selection of Different Alternatives

Figure 1.1 shows a scenario where different alternative implementations are available. For a concrete product one of them has to be chosen.

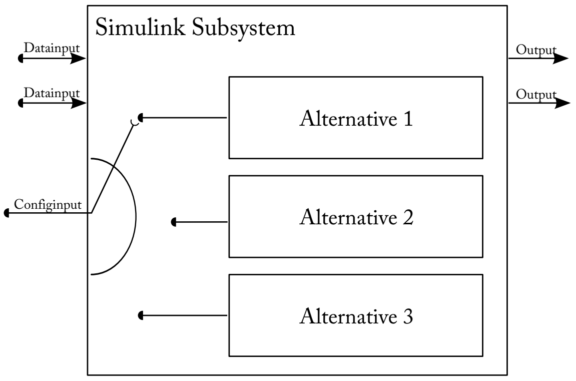


Figure 1.1.: Scenario with different alternative implementations

Switching between alternative implementations can have the following effects on the Simulink model:

Different input ports Alternative components may require different ports. Ports may differ on their count and / or on the types used.

Different output ports The same effect applies to ports provided by the different alternatives. Parts in the model consuming information provided by those alternatives must be connected to the alternative selected by a particular variant.

1.2.1.1. Desired Behavior

A mechanism is needed that supports the selection of valid variants. In order to be able to choose between alternative implementations an entity is needed that aggregates all possible alternatives identified during Domain Engineering. This entity also needs to store information when to select which alternative(s). The variant selection mechanism has to remove unnecessary parts of the implementation during the Application Engineering process.

Figure 1.2 shows the resulting model after choosing one alternative.

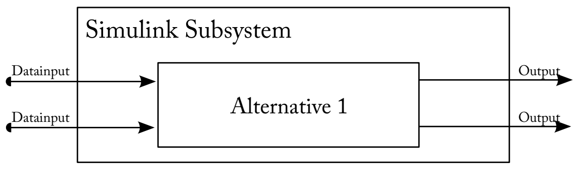


Figure 1.2.: After selecting one alternative unnecessary information is removed from the model

1.2.2. Implementing Optional Behavior

In this scenario a component provides functionality that is part of some variants and is omitted in others. Optional parts in such components are components themselves with ports they require and/or provide. Figure 1.3 illustrates this using dashed lines to mark the parts related to such an optional component.

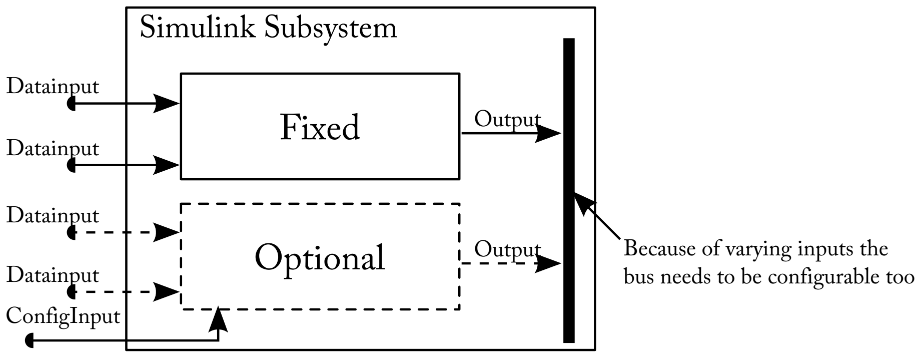


Figure 1.3.: Optional behavior (dashed) that may or may not be part of a component.

This scenario is related to the one presented in the last section. The main difference is that ports provided and / or required by optional components can be present in the public interface of a component or may be omitted completely. This changes slightly if the component that provides the optional behavior uses buses in its interface. In this case the number of required or provided signals in the bus has to be changed in accordance to the selected variant.

1.2.2.1. Desired Behavior

The needed behavior to implement optional parts is similar to alternative parts. An entity is needed that aggregates optional and non optional parts during Domain Engineering (DE). During Application Engineering (AE) the application engineer can choose between enabling and disabling the optional part. If the optional part is enabled, the resulting variant has to provide all items, e.g. the optional component itself, its ports and its connections. If the optional behavior is disabled the corresponding behavior should be removed from the concrete product. Handling of additional interfaces provided by optional behavior can be done in two different ways. Either the surrounding component provides explicit interface ports for the optional part or a bus object is used which combines all input and output data into one respective port. In both cases the varying interface needs to be handled by the entity that manages it. Figure 1.4 shows the different results that can be achieved applying the shown behavior using a bus object that collects the different signals provided by the aggregated parts.

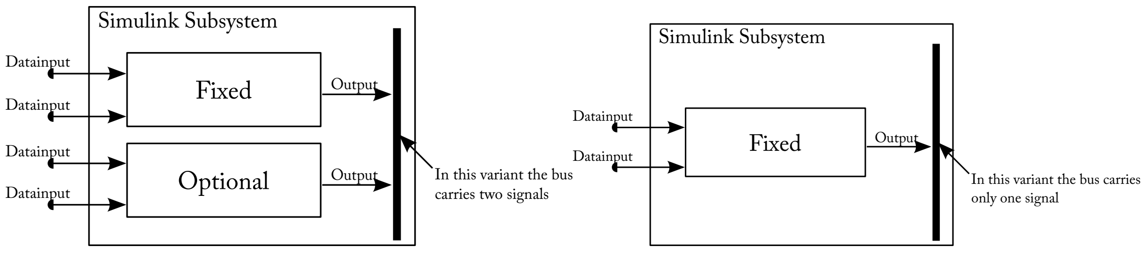


Figure 1.4.: The two variants of the component with the optional part enabled (left) and disabled (right)

1.2.3. Variable State Machines

As components have optional parts, state machines may have states that can be omitted in certain variants. State machines, as states, can be seen as special components. The main difference is that states always aggregate transitions or other states. This difference just affects the structure of the components handled within the variability management processes. From the variability management process' point of view, states and state machines are components that provide variable behavior. Figure 1.5 shows such a variable state machine component.

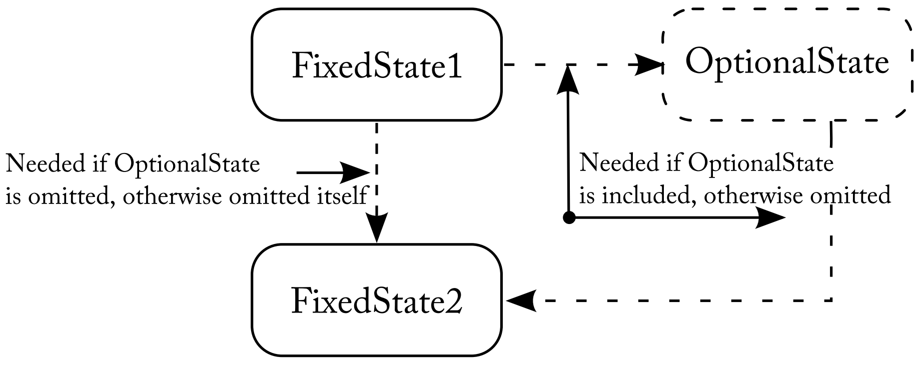

FixedState1
OptionalState
Needed if OptionalState is omitted, otherwise omitted itself
Needed if OptionalState is included, otherwise omitted
FixedState2

Figure 1.5.: State machines can be seen as special components whose optional states affect its transitions

This scenario combines two variability concepts:

Optional states are either present or missing in the resulting state machine.

Alternative transitions are needed to either serve or bypass an optional state, depending on its presence in the resulting state machine.

1.2.3.1. Desired Behavior

To overcome the problems mentioned above, optional states should be treated similarly to optional behavior as described in Section 1.2.2.1. If an optional state is selected to be member of a variant all its incoming and outgoing transitions must also be member of that variant. Alternatively, if this state is omitted, all of its incoming and outgoing transitions must be removed from that variant. If a state is omitted a transition is needed that bypasses that state. This way the state machine remains functional and valid. Figure 1.6 shows the different results that can be achieved using this work flow.

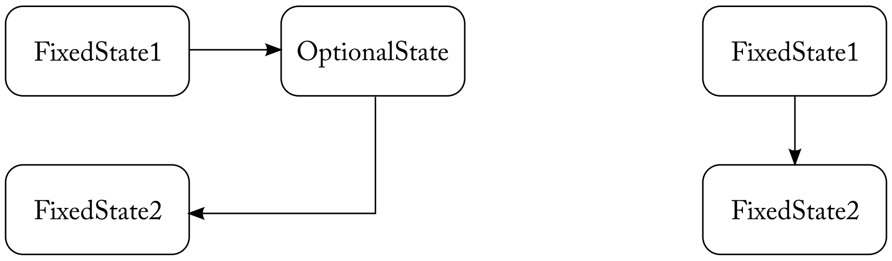


Figure 1.6.: The different variants of the state machine that either include or exclude the optional state *OptionalState*.

1.3. Outline

The objective of this thesis is to provide mechanisms that enable the implementation of the provided scenarios and to provide mechanisms that enable variant specific reduction of Simulink (and other) models based on a selected configuration. Chapter 2 provides an overview on developments related to this problems. This includes development of variable architectures and possibilities how component models can be developed in a variable way.

Chapter 3 provides the details on the design of the component model developed within this thesis. In Chapter 4 the prototypical implementation of this component model is explained in detail. This chapter also provides insights on how Simulink models can be integrated into this component model.

Chapter 5 shows how the scenarios provided in this chapter can be applied using the implementation presented in Chapter 4. Furthermore, this chapter provides inside how different variability patterns can be applied to the developed component model.

Finally, Chapter 6 summarizes the results of this thesis and gives indications on possible future work.

2. Related Work

2.1. Terminology

2.1.1. Defining Variability

Variability “refers to the ability or tendency to change” [Pohl et al., 2005]. In terms of software development this variability does not simply *“occur but is brought about purpose”*. [Pohl et al., 2005] provides three basic questions how variability can be identified:

What does vary? The answer to this question clearly identify the items or properties of items of the real world that may vary. [Pohl et al., 2005] refers to these items as *Variability Subjects.*

Why does it vary? According to [Pohl et al., 2005] there are different reasons for an item to vary ranging from different needs of different stakeholders to interdependencies between varying items.

How does it vary? The answer to this question provides concrete shapes a *Variability Subject* can take. [Pohl et al., 2005] refers to these “concrete shapes ” as *Variability Objects.*

2.1.2. Variation Points

Variation Points (VPs) [Pohl et al., 2005] are links between domain artifacts and the context of a real world application. Variation points, therefore, enrich those domain artifacts with contextual information required to fulfill the specific needs a concrete software product, derived from the domain artifacts, has.

Each variation point consists of three items:

- a unique identifier,
- a collection of values it can accept and
- its binding time

Variation points may depend on each other in a manner that the selection of one value in VP A reduces the set of possible values selectable for VP B. [Pohl et al., 2005] refers to such dependencies as *Variability Constraints.*

Since variation points link the reusable software artifacts to concrete products derived from them, it is important to decide when a concrete variation point is bound to a concrete value defined in its value collection. This decision point to ultimately bind a variation point is known as the *Binding Time* of a variation point.

2.1.3. Variant

A *Variant* [Pohl et al., 2005] is identified by a single value a *Variation Point* defines. Variants can be associated to artifacts to indicate that those artifacts correspond the the particular selection from the corresponding variation point.

2.2. Representing Variability

2.2.1. Feature Oriented Domain Analysis

Feature Oriented Domain Analysis (FODA) [Kang et al., 1990] is a method to perform domain analysis based on features. A Feature is

> *. . . a prominent and distinctive user visible characteristic of a system . . .*

[Kang et al., 1990]
Identifying those characteristics in different applications of a domain, and abstracting them into a Feature Model is the main goal of FODA.

2.2.1.1. Feature Models

A Feature Model [Kang et al., 1990], as its name imply, contains the set of features that make up the analyzed domain. Its hierarchical structure shows dependencies between the different features and makes them explicit. Inside the Feature Model features can be categorized into three levels:

Mandatory This features must be present in any application for the selected domain

Optional This features can be selected or deselected

Alternative Within a set of features exactly one must be selected

The last gap, expressing semantics between different features, is filled by composition rules:

Requires References all optional or alternative features that must be selected in order to be able to select a feature that depends on them

Mutually exclusive with References all features that must not be selected if a certain feature was chosen.

2.2.2. Using Domain Specific Languages to Describe Variability

Feature Models provide a fixed, tree like structure for defining features different products may have. There are cases when the design of such Feature Models get cumbersome. [Voelter and Visser, 2011] describes an approach how Domain Specific Languages (DSLs) can help to overcome this. Because DSLs are more flexible in terms of their definition and the domain they are designed for, they can help to formulate problems more easily than Feature Models that try to meet needs across different domains.

2.2.3. The Common Variability Language

The Common Variability Language (CVL) [Haugen et al., 2008] implements a common language to reflect variability independent from any used DSL but yet can be used with any MOF [Object Management Group, 2006] based DSL. To accomplish this, a generic, domain independent transformation process is needed. In [Haugen et al., 2008] this generic transformation process is based on three distinct models:

The Base Model is the model defined by the DSL used to implement the solution family.

The Variation Model specifies variants that can be derived from the base model by using variation elements that associate variability specifications. These variation elements are built on top of a set of model elements provided by the used base model.

The Resolution Model is responsible to derive concrete products based on choices made in the variation model.

To actually perform the transformation from a Base Model to a concrete product [Haugen et al., 2008] defines a two stage transformation process consisting of *Resolution Transformation* and *Variability Transformation*. The task of the *Resolution Transformation* is to apply resolutions defined within a Resolution Model to a Variation Model. The result of this step is a new Variation Model containing elements that indicate the resolutions done. The result of this step is the *Resolved Variation Model*. The *Variability Transformation* finally executes the transformations identified during *Resolution Transformation* on the base models referenced by the Resolved Variation Model. CVL was proposed for standardization with the Object Management Group (OMG) (see [Haugen et al., 2010])

2.2.4. Applying Variability to Model Driven Development

[Dauenhauer et al., 2009] identifies two basic approaches of introducing variability into Model Driven Development (MDD):

Variation Points in a Model In this approach variation points are represented by dedicated elements in the modeling environment.

Merging Assets In this approach the model consist of several fragments representing the variable parts. A product is derived by merging all fragments (assets) that belong to the desired variant.

Pure::variants, shown in Section 2.3.3.1, is capable to use both approaches in its Family Models, whereas the Variant Block Set shown in Section 2.3.3.2 can only use the first.

Also based on the second approach, [Dauenhauer et al., 2009] provides a modeling language which raises variability to be a first class citizen of that language. The provided language consist of *Clabjects* and *Connectors*. *Clabjects* represent placeholders for domain specific model elements, such as sensors, and *Connectors* represent connections between them. Each connector can be tagged with a *Relation* to express its intent. To express variability of certain elements the *enables* and *requires* relations are provided. As the used names indicate, an *enables* relation enables a certain *Clabject* in a specific variant, whereas the *requires* relation requires a certain *Clabject* to be member of a chosen variant.

Using this basic model elements three kinds of variable behavior can be expressed [Dauenhauer et al., 2009]: 1) enabling/disabling of *Clabjects* 2) enabling/disabling of *Connectors* 3) enabling/disabling of variant specific field values

All of this behavior is also required by the scenarios provided in Section 1.2. Although [Dauenhauer et al., 2009] state that their approach is related to asset based variability the connector concept can also be used with explicit model elements for variation points. The model developed in this thesis uses an approach based on this concept.

2.2.5. Generating Variant Specific Models

One major goal of this thesis is to provide the developer the means to generate variant specific models based on generic system architectures. [McRitchie et al., 2004] provide an approach that achieves this for embedded systems which use C++ based development. Components in this approach define so called "argument lists" to define their variable behavior. In their paper they identify three major steps in order to create variant specific components:

1. Analysis
2. Weaving
3. Generation

In the analysis phase the required work products are identified. These work products, including their implementation, are gathered from a repository. From each work product a Abstract Syntax Tree (AST) is generated which is used to find dependencies to other work products.

In the weaving phase information from the analysis phase and the specification of the argument list is used to deliver a variant specific component specification. This component specification is obtained by annotating the AST of the component with information about the internal composition of the component and identified dependencies within the component.

In the generation phase, the variant specific work product is finally generated. The first step is to generate the components implementation based on the annotated AST delivered by the weaving phase. Finally, the components interface is generated so that the newly created component can be used independently from its implementation.

Although the proposed approach is targeting code based instead of model based development the general concept seems convincing. On the other hand there are two drawbacks of the proposed solution:

Implementing different binding times Using the given approach it is not possible to implement different binding times. The argument lists proposed by the paper are always bound during the generation phase.

Configuration of Subcomponents Within the proposed approach subcomponents are always configured using arguments lists defined by their parent. Therefore, the model only supports *Plain Propagation*[Reiser et al., 2009]. Subsequent sections of this thesis will show that this propagation pattern is incompatible to the patterns possible within Simulink.

2.2.6. Compositional Variability

[Reiser et al., 2009] defines three key features in order to achieve variability in components:

1. Provide a structure that embeds variability mechanisms into components.
2. Provide means to specify the variable parts in the components interface.
3. Provide a mechanism to map variants specified in the components interface to either its internal structure or to its variable sub components.

Providing this features components can be designed with variability in mind and can be reused systematically by selecting different (supported) configurations.

Beside this key features variable components have to implement [Reiser et al., 2009] provides a catalog of five patterns on how variability may be propagated through different levels of composition.

2.2.6.1. Plain Propagation

Plain Propagation is used by adding variability specifications from subcomponents to the variant specification of the composite parent. Using this scheme of propagation the composite component decides when and, more important, how to bind its subcomponent, by propagating the selected variant back to the subcomponent.

2.2.6.2. Direct Binding

Direct Binding is used when variability of lower level components is bound to a specific variant within the definition of their parent component. Therefore, the variability of the subcomponent is not propagated to its parent.

2.2.6.3. Orthogonal Propagation

Sometimes *Plain Propagation* is not enough. This is the case when the propagation target defines a different variability scheme as its composite parent component provides. To avoid propagating such differences into the components interface the component can define mappings on how a certain configuration item has to be modified in order to serve the needs of its variable subcomponent.

2.2.6.4. Top-Level Propagation

Top-Level Propagation is the opposite to *Orthogonal Propagation.* Using this pattern the component designer decides to let the configuration of a certain subcomponent flow from the most high level parent component to it. [Reiser et al., 2009] defines two variants of this pattern. In the first variant components in the different hierarchy levels above the component in question can decide whether to directly provide a configuration (by for example applying *Direct Binding*) or to hand over the job to the next hierarchy level. The second variant strictly hands over the job of providing a configuration to the most outer hierarchy level.

2.2.6.5. Global Features / Reverse Propagation

This pattern reflects the fact that certain features are required which are needed on different subtrees of the component hierarchy but need to be configured consistently across all occurrences. Aspect Oriented Programing [Kiczales et al., 1997] refers to such features as *Aspects*.

2.3. Separating Concerns - The Software Product Line Modeling Process

For the implementation of variability an appropriate development process is required. Software Product Line Engineering provides processes provides such processes. A Software Product Line 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.*

[Clements and Northrop, 2001]

Therefore, the general idea of Software Product Lines is to generate a variety of products from a common software architecture using common mechanisms to derive those products. Software Product Line Engineering (SPLE) refers to the processes how this can be accomplished in a systematic way. The following sections deal with these processes and the tasks related to them.

2.3.1. Separating Problem from Solution

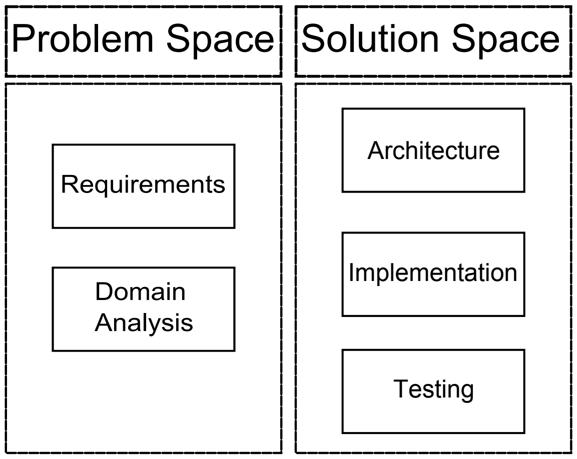


Figure 2.1.: The development activities assigned to different spaces of development (based on [Berg et al., 2005]

As shown in Figure 2.1 SPLE divides the different phases of software development, requirements engineering, domain analysis, architecture, implementation and testing, into two major groups [Berg et al., 2005]:

Problem Space In problem space, the engineer does domain analysis and requirements engineering in order to deliver a specification of the system to build. This specification is independent from any technical realization.

Solution Space In solution space, concrete systems according to the specification using architectural, implementation and testing skills of the engineers are build.

2.3.2. Splitting the SPL development process

Since variability has impact on both, the problem and the solution space, a process is needed that helps to engineer this variability. The concept of SPLE defines two major outcomes: Reusable software artifacts and concrete products. Reusable software artifacts are parts of the software that are planed to be reused systematically for different variants, i.e. concrete products, or even other projects. Concrete products on the other hand, as their name imply, are software packages derived from those reusable software artifacts and fulfill the specific needs of the desired product. Since the aim of SPLE is systematic reuse of the various artifacts it is hard to model reusable software artifacts and concrete products that rely on them within one process. Therefore SPLE splits this process vertically into two parts [Pohl et al., 2005]:

- Domain Engineering (DE)
- Application Engineering (AE)

Figure 2.2 shows the relationship between these two processes.

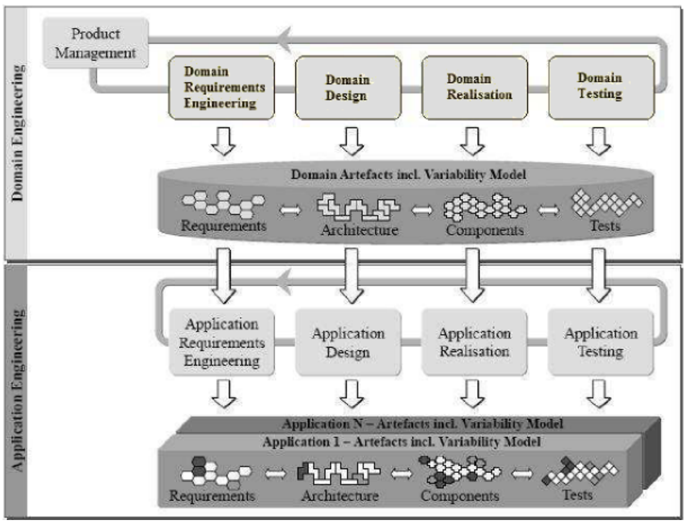


Figure 2.2.: Domain Engineering provides the basic artifacts which are used by Application Engineering to build concrete applications [Pohl et al., 2005]

2.3.2.1. Domain Engineering

The task here is to develop a software architecture consisting of reusable software artifacts that can be controlled by variation points in order to derive concrete products within the AE process. Before being able to implement this software architecture and to define variation points the different variants that are expected need to be chosen. The domain needs to be bound to specific products, because being generic in all aspects would result in unmanageable complexity. FODA is one mean to identify the needed features for a given domain (see Section 2.2.1). After variant identification developing the software architecture can start. The concept of variation points helps here to express which parts of a software artifact vary [Pohl et al., 2005]. This makes it possible to control different instances of the software core without adjusting the source code itself. The upper part of Figure 2.2 illustrates the different activities inside the Domain Engineering process.

2.3.2.2. Application Engineering

The task here is to derive concrete products from the generic software architecture. This means that all variation points have to be bound to the values needed for a specific variant and to resolve dependencies between different variation points. Beside the concrete implementation of the derived product the AE process also applies to all other artifacts developed while doing DE including requirements, testing and documentation. These activities are illustrated in the lower part of Figure 2.2.

Code generation Within automotive software code generation from models plays a major role since generating code is the one of the major goals of MDD [Schmidt, 2006]. Because of the very limited resources, in terms of processing power and memory found in hardware used within the automotive domain, it is important to optimize generated code. Taking variability into account code generation is even more important because, depending on the selected variant, the code generator can remove code that is not needed by a particular variant. To achieve this, the selection of proper binding times for variation points is essential.

2.3.2.3. Merging Processes and Development Spaces

As stated in Section 2.3.2, DE and AE have impact on both, the problem and the solution space defined in Section 2.3.1. Therefore, it seems reasonable to integrate both processes into the problem as well as into the solution space.

The resulting four quadrants, illustrated in Figure 2.3, show the general activities when doing Software Product Line based development and can be described as:

Quadrant 1 Identify variability

Quadrant 2 Technical realization of the identified variabilities

Quadrant 3 Specify concrete products

Quadrant 4 Implement concrete products

2.3.3. Tools to Implement the Software Product Line Engineering processes

The following sections describe the application of the processes shown in Section 2.3. To successfully implement the SPLE processes a tool chain is needed. In case of this thesis

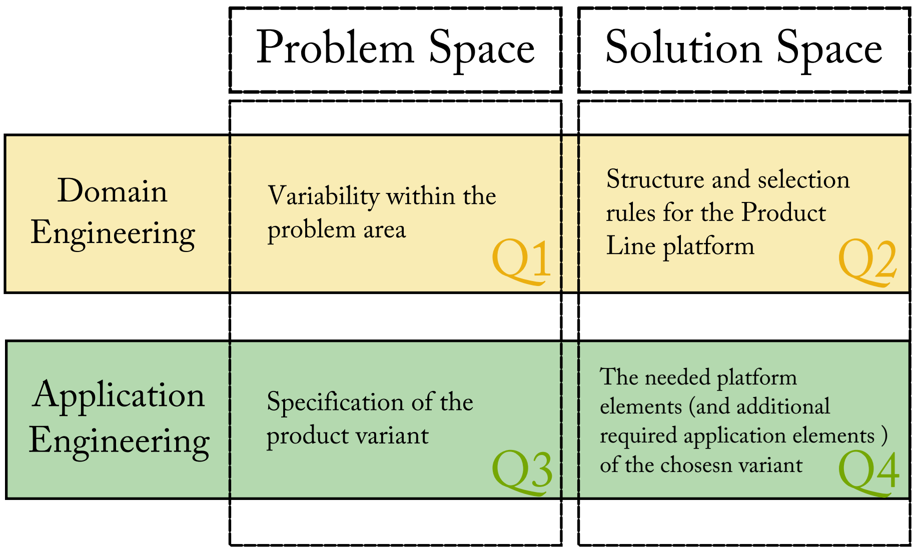

Problem Space
Solution Space
Domain Engineering
Variability within the problem area
Q1
Structure and selection rules for the Product Line platform
Q2
Application Engineering
Specification of the product variant
Q3
The needed platform elements (and additional required application elements) of the chosesn variant
Q4

Figure 2.3.: Combining problem and solution space with the Domain Engineering and Application Engineering processes results in an effective way to develop variable software architectures

this tool chain needs to support the processing of Simulink models. Therefore, Simulink, though not intended for SPLE, is analyzed. pure::variants on the other hand provides mechanisms to enrich Simulink based models with variability information and provides the ability to configure these models. Table 2.1 shows the used tools, their purpose and the variability modeling process they're involved in.

Tool	**Purpose**	**Variability Process**	
		DE	**AE**
Simulink	Functional modeling	x	
	Simulation and testing		x
TargetLink	Code generation		x
pure::variants	Feature and Variant modeling	x	x

Table 2.1.: The tool palette used to develop variant rich automotive software

Using a tool chain raises the need to define the interaction points between the tools involved with it. These interactions show which fragments are exchanged between the tools and, more importantly, which fragments are required to start a task associated with a certain tool. Figure 2.4 illustrates the identified interaction between pure::variants and Simulink including the different roles associated with the tools.

2.3.3.1. pure::variants

pure::variants provides a modeling environment for the four quadrants listed in [Beuche, 2003]. These models are associated to Domain Engineering and Application Engineering. Table 2.2 shows the models associated to these processes.

There are five steps in order to control variability within Simulink models using pure::-

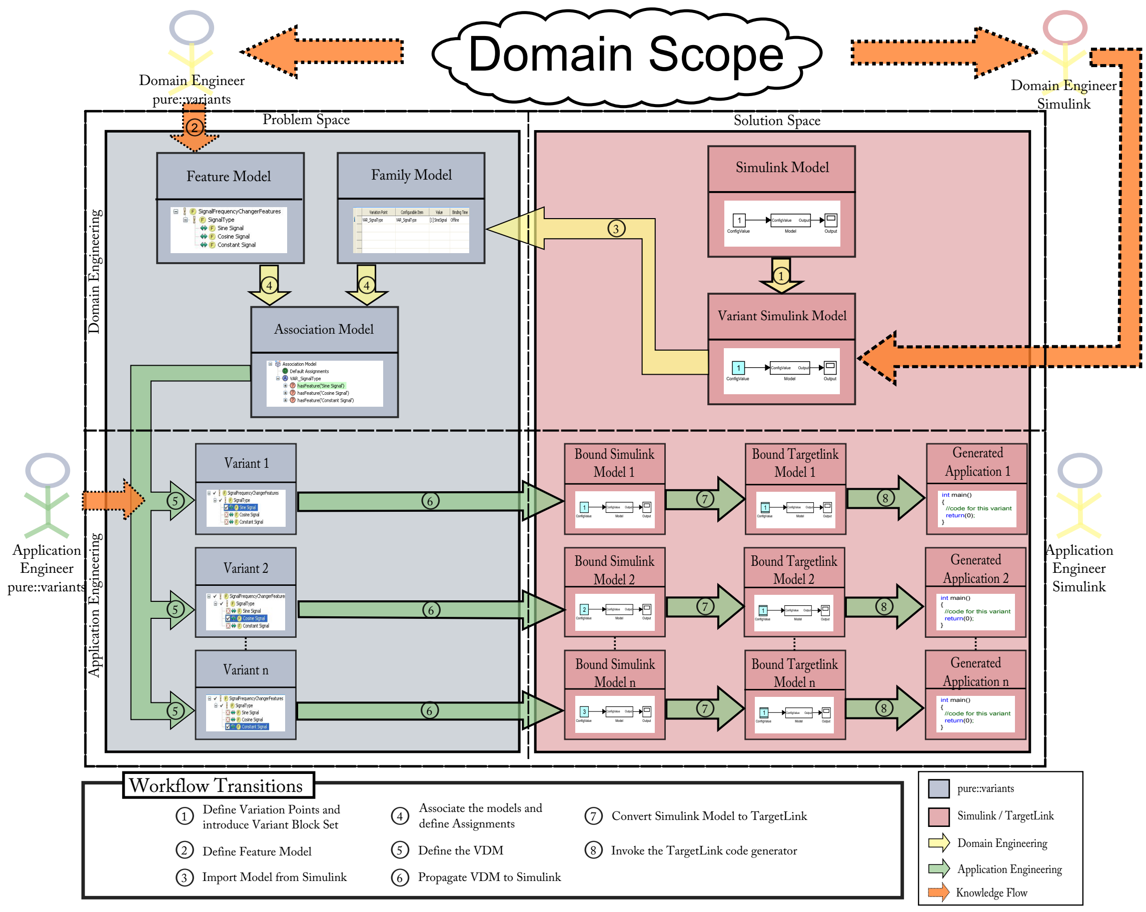


Figure 2.4.: Data and knowledge flow between pure::variants and Simulink embedded in the SPLE processes

SPLE Process	**pure::variants Model**
Domain Engineering	Feature Model Family Model
Application Engineering	Variant Description Model Variant Result Model

Table 2.2.: Mapping pure::variants models to SPLE processes

variants:

1. Define the Feature Model.
2. Import the functional Simulink model into a Family Model.
3. Associate the Feature Model to this Family Model.
4. Develop the Variant Description Model for each desired variant and
5. propagate the selected configurations back to Simulink.

Feature Models in pure::variants are an implementation of [Kang et al., 1990] and reside in the first quadrant shown in Section 2.3.2.3. Feature Models in pure::variants start with a so called *root feature* i.e. the name of the model. Below that root feature the needed features and their type can be implemented. Beside the feature types, *mandatory*, *optional* and *alternative*, defined in [Kang et al., 1990], pure::variants offers an additional feature type called *or* indicating that at least one feature from a given feature set must be selected [Beuche, 2003]. To model complex dependencies between different features pure::variants provides the ability to implement Prolog statements to express those dependencies [pure-systems GmbH, 2009].

Family Models in pure::variants [Beuche, 2003] are used to model the solution family. Elements in a Family Model can have the same relations as defined for Feature Models. Family Models consist of *Components*. Each *Component* can consist of other *Components* and so called *Part* elements. *Parts* are logical elements representing key elements of the component structure. Such elements can be the interface description (external) or a class that implements a certain facet of the component (internal). Since such logical parts need a physical representation so called *Source Elements* are provided. Such a *Source Element* can be for example a file When importing Simulink models into pure::variants such a Family Model is created. In this case the Family Model contains the different variation points defined in Simulink (see Section 2.3.3.2).

Association Models Association models [pure-systems GmbH, 2009], as their name imply, associate different models. They provide the gateway between the first and third quadrant shown in Section 2.3.2.3. In case of enriching functional models developed using Simulink, this model makes it possible to associate a previously defined Feature Model to the variation points of the Simulink models. The key feature of Association Models is the ability to define relationships between models, called *Assignments*. Unfortunately, these Association Models seem to be provided only within the Simulink package provided by pure::variants.

Assignments If assignments are added to the Association Model, a selection change in one model can automatically affect elements in another models. This can be used to adjust variation points in a Simulink model based on the feature selection in its associated Feature Model. Assignments always consist of two parts:

Condition The condition when the assignment should be executed, i.e. if a certain feature was selected.

The assignment operation itself In case of variation points the assignment operation will assign a certain value out of a variation points value collection to that variation point.

Variant Description Models Application Engineering in pure::variants is done using a so called Variant Description Model [Beuche, 2003]. Generally, VDMs are used to select specific variants by selecting variable elements defined in Feature and Family Models. The models that can be used by a VDM are collected in a so called *Configuration Space*.

Association Models mentioned in the previous section are also defined within such configuration spaces. In case of Simulink based Family Models the view for VDMs provided by pure::variants contains an additional command to propagate the selected bindings of variation points back to Simulink.

Variant Result Models Variant Result Models are responsible for storing the variant specific parts of a Family Model. Identifying these parts is based on the selection done in the corresponding VDM.

2.3.3.2. Matlab / Simulink

To be able to model variability within Simulink, pure::variants provides a special block set called Variant Block Set [Dziobek et al., 2008]. The block set is divided into two categories: *Control Blocks* and *Variability Mechanisms*. Table 2.3 shows the different blocks provided in these categories.

Category	**Blocks provided**
Control Blocks	VariantConstant Block Variant Store Variant Read Variant Write
Variability mechanisms	Switch If Model Chart Enabled Subsystem

Table 2.3.: The different blocks provided to model variability in Matlab / Simulink

The functionality provided by these blocks is the same as of their non variant-related counterparts in Simulink except that variability information can be assigned to them. According to [Dziobek et al., 2008] some variability mechanisms can be used to model the feature types presented in Section 2.3.3.1. This possibilities are summarized in Table 2.4.

Used block	**Expressed Feature Type**
If	Or Feature
Switch	Alternative Feature
Enabled Subsystem	Optional Feature

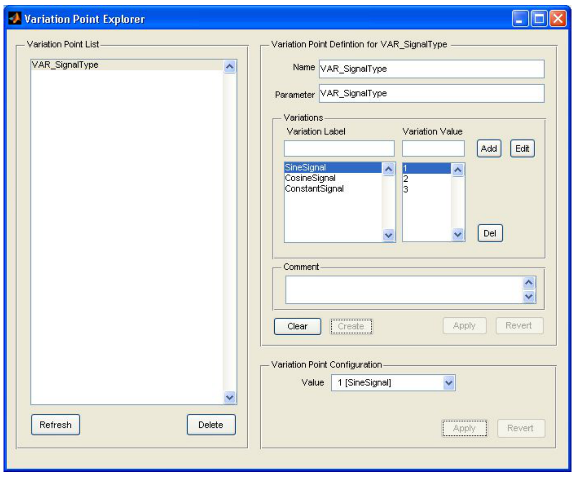
Table 2.4.: The different variability mechanisms used to model different feature types

The variability information for all of this blocks has to be provided in terms of variation points. Once these variation points are defined they can be assigned to instances of these blocks. Furthermore different instances of different blocks can share the same variability information by just assigning the same variation point to them.

This feature is essential if the selection of a certain feature has impact on different parts of the implementation. Using one variation point makes it possible to **consistently**

switch between variants by adjusting the value of just one variation point.

To define variation points the mentioned block set provides a special tool called *Variation Point Explorer*. Figure 2.5 shows this tool in action.



Variation Point List: VAR_SignalType

Name: VAR_SignalType
Parameter: VAR_SignalType

Variation Label	Variation Value
SineSignal	1
CosineSignal	2
ConstantSignal	3

Value: 1 [SineSignal]

Figure 2.5.: Using the Variation Point Explorer to define Variation Points in Simulink

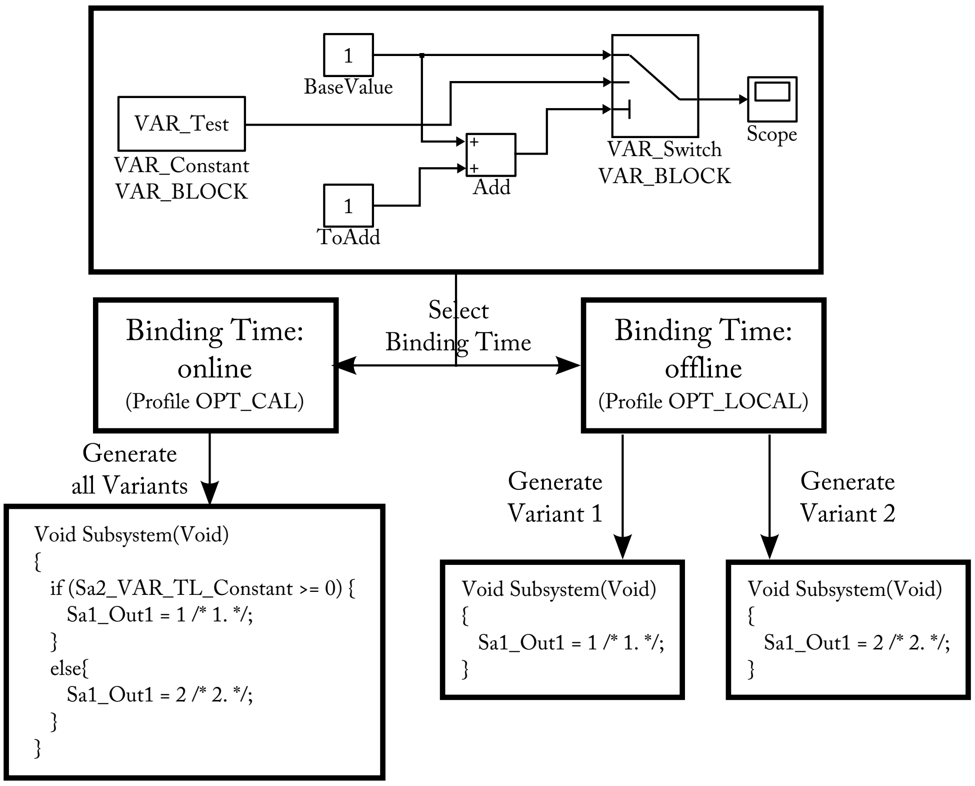
Code Generation In MDD code generation produces the executable products. Therefore, generating code from variant-rich Simulink models can be seen as Application Engineering in the Simulink domain as illustrated in the forth quadrant of Figure 2.4. Within the automotive domain *TargetLink* has established itself as the main tool for generating code from Simulink models. In order to use this code generator Simulink models need to use a special block set provided by TargetLink. TargetLink provides auto conversion for existing Simulink models. In this conversion Simulink blocks are replaced by their TargetLink counterparts. This conversion also covers the Variant Block Set. This step can be done fully automatically.

Furthermore, TargetLink provides different profiles to instruct its code generator how to handle certain cases. [Beuche and Weiland, 2009] shows two profiles provided by TargetLink that are important when dealing with the binding time issue. These profiles, called *OPT_CAL* and *OPT_LOCAL*, control how constants get transformed into source code. The first option tells the code generator that constants may change during runtime. The second that it should treat them locally which means that the value of a certain constant will not change. This behavior can be directly mapped to the different binding times mentioned earlier. By using this profiles these two binding times can be applied to the generated code, at least manually. Figure 2.6 illustrates the results of applying these code generation profiles when generating code from a Simulink model.

2.3.3.3. Evaluation

The workflow presented in Figure 2.4 shows that using the Variant Block Set can save a lot of time for modeling variability in Simulink. Indeed a bit of work in terms of

- identifying variability,



```
Void Subsystem(Void)
{
  if (Sa2_VAR_TL_Constant >= 0) {
    Sa1_Out1 = 1 /* 1. */;
  }
  else{
    Sa1_Out1 = 2 /* 2. */;
  }
}
```

```
Void Subsystem(Void)
{
  Sa1_Out1 = 1 /* 1. */;
}
```

```
Void Subsystem(Void)
{
  Sa1_Out1 = 2 /* 2. */;
}
```

Figure 2.6.: Resulting code after applying either *OPT_CAL* or *OPT_LOCAL* profiles on the same Simulink model

- defining variation points,
- defining Feature Models and
- defining assignments

needs to be done but the extra effort pays off fast when deriving concrete applications. It is pretty straight forward to propagate different configurations to Simulink models and to simulate them, by just selecting the desired variant and clicking the *Propagate to Simulink* button.

But there's a but: Within all blocks defined by the Variant Block Set only one actually consumes variability information provided by its associated variation point: The *Variant-Constant* block. All other blocks still reference variation points but do not process the information provided to them. Even the remaining control blocks, *VariantStore*, *VariantRead* and *VariantWrite* do not process variability information tough at least control blocks are intended to do so. Table 2.5 summarizes these findings.

Another important aspect of variation points is their binding time. Section 2.3.3.2 shows how important the binding time is for code generation. But, even before code is actually generated binding of variation points is important. By binding a model to a specific variant, elements can be removed that are not related to this variant. In [Beuche and Weiland, 2009] this binding time is called *ModelConfigurationTime*. Even

Controllable using VPs in	Control Block		Variability Mechanism
	Variant Constant	Variant Store	
pure::variants	✓	-	-
Simulink	✓	-	-
Can be propagated from pure::variants to Simulink	✓	-	-

Table 2.5.: Blocks from the Variant Block Set and their ability to be controlled using variation points within pure::variants and Simulink

if code is generated and deployed binding of variation points is possible and is often called calibration. Table 2.6 summarizes the possibilities of using binding times in Simulink and TargetLink using the Variant Block Set.

Tools	Binding Time	Control Block		Variability Mechanism	Automatically applicable
		Variant Constant	Variant Store		
Simulink	Pre code generation	-	-	-	-
	Post code generation	✓	-		-
	runtime	-	-		-
TargetLink	Pre code generation	✓	-		-
	Post code generation	✓	-		-
	runtime	-	-	-	-

Table 2.6.: Different binding times and tools and their applicability

Based on the scenarios provided in Section 1.2 an application of the *ModelConfiguration* binding time would be preferable. Unfortunately, the Variant Block Set does not support this binding time at all. Actually, the support of *PreBuild* binding is only achieved through applying code generator logic as shown in Section 2.3.3.2.

2.4. Component Models

A Component model

. . . defines standards for (i) properties that individual components must satisfy and (ii) methods, and possibly mechanisms, for composing components.

[Crnkovic, 2002]

A large number of component models have been developed and used in the last couple of years. Most notable examples are CORBA Component Model (CCM), Component Object Model (COM), Enterprise Java Beans (EJB) and .NET, at least on the client / server application domain of software engineering. This thesis focuses on the problem of enhancing components that 1) target the automotive domain and 2) are developed within a model based environment like Simulink with variability. Because a model based approach enforces explicitness such variability has to be defined explicitly too. Non of the mentioned component models handle variability explicitly or are usable within automotive software. Therefore, this section focuses primarily on component models that try to handle at least one of the mentioned focus points.

2.4.1. AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) has established itself as an widely accepted standard to model and develop software in the automotive domain. Organized in a layered architecture[Autosar, 2009e], [Buschmann et al., 1996], it tries do decouple application level software components from the underlying hardware. Using these scheme makes it possible that vendors of Electronic Control Unit (ECU) hardware can provide interfaces in a standardized manner that can be used by software developers to integrate them into their software.

The main problem that remains and that is not fully covered by the AUTOSAR specification is how to make existing software solutions for the automotive domain compliant to the AUTOSAR standard.

2.4.1.1. AUTOSAR Overview

AUTOSAR consists of three major layers:

- Application Layer
- Runtime Environment Layer
- Basic Software Layer

Figure 2.7 illustrates the interactions of these layers.

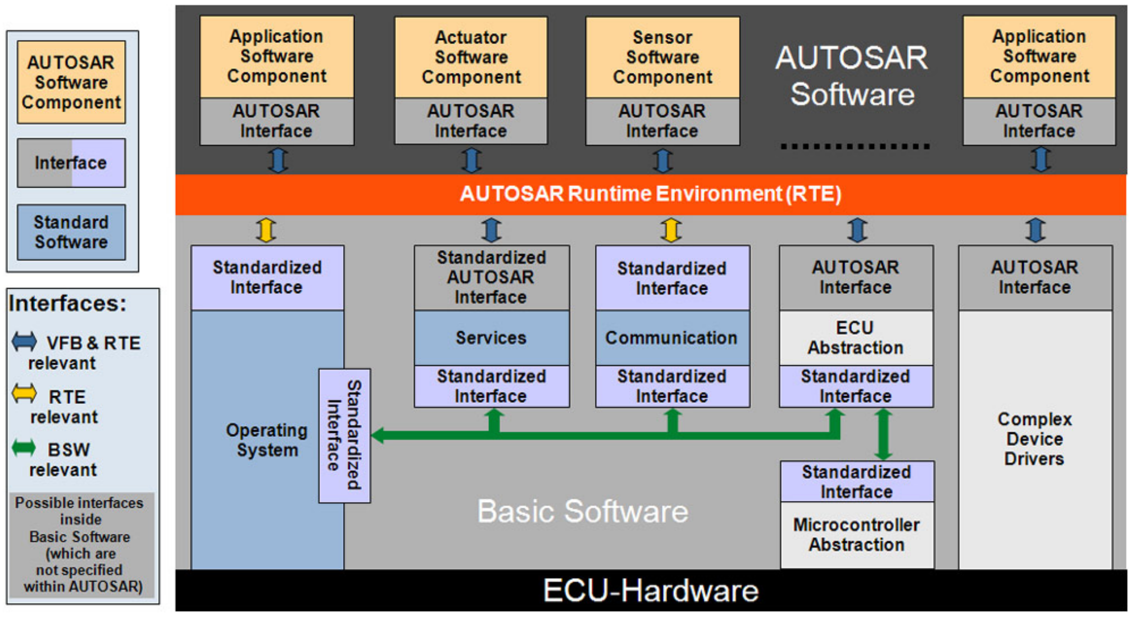


Figure 2.7.: The architecture of AUTOSAR, [Harald et al., 2004]

2.4.1.2. Components and Ports

The AUTOSAR specification [Autosar, 2009e] defines various types of components and calls them SoftWare Component (SWC). Atomic Software Components represent so called self contained software components. If composition of different components is needed the so called `CompositeType` has to be used. Furthermore, AUTOSAR defines special purpose component types to model sensors and actuators independent from their real hardware. [Autosar, 2009e] provides the details for various component types that are defined.

To let components interact with each other so called ports can be defined. AUTOSAR distinguishes between two kinds:

Receive Ports Ports of that type define required input values to the component. This ports can be compared to parameters to a function in a common programing language.

Provide Ports Ports of this type define output values a component provides. This concept is equivalent to the concept of return values of functions in most programing languages except the fact that a component can provide more than one provide port.

[Autosar, 2009e] provides more details on ports and how they have to be defined.

2.4.1.3. AUTOSAR Compliance

This section gives a short overview on how to make software compliant with the AUTOSAR standard. The main parts that need to be considered are the used hardware and the implemented software components themselves. Since the RTE and the basic software modules also need to be implemented, the implementation of this layer of the AUTOSAR architecture needs also to be tested for comformety.

Hardware In order to use specific hardware it needs to provide an interface to enable communication with the RTE.

Software Components Software Components, in the sense of AUTOSAR, are forced to either just talk to each other via their defined ports or talk to the underlying hardware only through the RTE. Therefore no direct interaction with specific hardware or specific basic software modules is possible.

Run Time Environment and Basic Software Modules To connect the more abstract software components to specific hardware the middle layer consisting of the RTE and the Basic Software Modules is needed. The RTE needs to provide the interface to software components to let them interact with the basic software modules. The basic software modules on the other hand need to interact with the hardware. Because all these interaction points need to be stable the specification [Autosar, 2009f] provides the definition of a so called *Standardized Interface* which has to be implemented by Basic Software Modules. To test these implementation to be conforming to the specification a number of test runs [Autosar, 2009a, Autosar, 2009b, Autosar, 2009c, Autosar, 2009d] need to be taken.

2.4.1.4. AUTOSAR and Simulink

Simulink is the dominant tool for development of automotive software. The main reason is the ability to develop the software model based and therefore, do development and testing of the software inside the same environment. Therefore, it seems to be necessary to embed the concepts AUTOSAR defines into the world of Simulink.

Fortunately, people who defined the AUTOSAR standard had also recognized this problem and released a guide [Autosar, 2009g] how to map AUTOSAR concepts to Simulink. Though this guide does not imply tool support it provides enough information to make existing Simulink models AUTOSAR compliant.

2.4.1.5. Migration to AUTOSAR

[Eisemann et al., 2009] defines an approach called *Push Button Migration* to migrate function models (such as the ones possible with Simulink) to AUTOSAR. The basic idea is to use the existing models and enrich them with AUTOSAR *compliant parameterization* to make it possible to generate AUTOSAR compliant models [Eisemann et al., 2009]. With that migration in mind [Eisemann et al., 2009] defines a so called *Bottom-Up* approach which starts with a set of functions that form the base of the overall architecture of the Electronic Control Unit. Another approach is to start with the architecture of the Electronic Control Unit, implement the behavior in Simulink and re-import the components back into the architecture design. This is called *Top-Down Approach* because it starts with the overall design before implementing behavior.

The methodology defined in [Eisemann et al., 2009] is somehow bound to commercial tools. [Kum et al., 2008] presents a more general approach. Looking at the very core of both papers it seams to be reasonable to start from a model of the application that should be ported to AUTOSAR instead of legacy handwritten code.

2.4.1.6. Tool Support

dSpace provides the Target Link package to be used within Matlab/Simulink. With the appearance of AUTOSAR this package was enhanced with custom blocks targeting various concepts defined within AUTOSAR. Furthermore, they provide a migration tool to port existing Simulink models into models suitable for AUTOSAR.

Unfortunately, no evaluation, especially for the migration tool, exists. So it is hard to say if the tool behaves in the described way.

2.4.2. The CompAA Component Model

The CompAA component model proposed in [Lacouture and Aniorté, 2008] defines so called adoption points for components. A component can have as much adoption points as subservices. Each adoption point is connected to the sub service it controls. All sub services are controlled by the so called *Component Core*. Each component in the CompAA Component Model has to define such a core service. Figure 2.8 illustrates how components in the CompAA component model are designed.

The CompAA component model is indented for highly distributed systems. Resolving adaption points is done by agents that exchange information about the functional and non

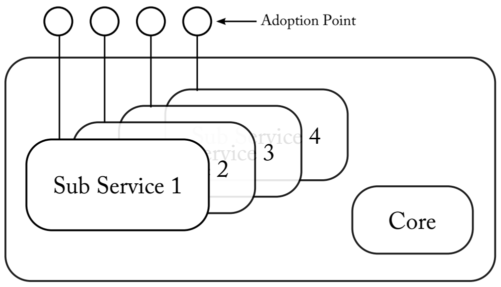


Figure 2.8.: Subservices controlled by the component core and configured by adoption points, based on [Lacouture and Aniorté, 2008]

functional properties of the subservices they configure. Although automotive software is distributed too, adopting the CompAA component model for the automotive domain is problematic. Runtime service discovery is almost not present because communication between components should be reduced to a minimum. Configuration should be done before deploying the software to the ECU network in the car. Furthermore, memory requirements are tight, therefore runtime instances of components and of agents may be problematic too. As mentioned, CompAA components need a dedicated core subservice. Such a component core is not always necessary. Especially in Simulink based software development buses are often used to collect results from subservices which are then consumed by other components. Adoption points on the other hand seem to be very convincing. The only drawback of them is that adoption points are assigned to subservices of the component rather than to the component it self making configuration of the component harder. Nevertheless, adoption points are adopted for the component model proposed in this thesis.

2.4.3. Multilevel Component Composition

Merijn de Jonge [de Jonge, 2004] identifies requirements for multilevel component composition and for multi level variability. Components that are used for multi level composition need to meet the following requirements:

- They have to provide a *required interface.*
- They have to provide a *provided interface.*
- A mechanism has to be provided that maps required to provided interfaces.

For variability of such components [de Jonge, 2004] defines the following key criteria:

Variability Interface Each component has to define a separate interface which provides information about different configurations of this component.

Variability Binding Each component has to provide mechanism to bind variation point to a selected configuration.

Variability Mapping For every level, except the model root, a mechanism is needed that maps a binding selected for a certain component to the variability interface of one of its sub components, therefore orthogonally propagating this configuration [Reiser et al., 2009].

Based on this requirements [de Jonge, 2004] provides a framework that enables the implementation of multilevel composition of executable components. Therefore, the paper discussed here, does not provide a component model that implements the mentioned requirements on its own. Instead it tries to integrate existing (binary) components into a multi level composition hierarchy. Since this thesis focuses on providing a component model that supports variability at modeling level this approach can not be taken. Nevertheless the requirements to such a component model [de Jonge, 2004] defines are reasonable. The only drawback of the proposed solution is, that it only supports *Orthogonal Propagation* as defined in [Reiser et al., 2009] directly. Since Orthogonal Propagation and Plain Propagation are very similar to each other [Reiser et al., 2009], Plain Propagation may also be possible. But especially *Top Level Propagation* can not be achieved with this approach, but is needed when using Simulink, as shown in Section 5.2.2.

2.5. Hypothesis

Non of the mentioned models or methodologies support the scenarios required in Section 1.2 completely. Based on the SPLE processes, this thesis will provide an approach, independent from concrete models, that is capable of supporting these scenarios. Furthermore, a prototype will be provided that implements the proposed approach based on Simulink and pure::variants. Because Simulink is the dominant tool for model based development in the automotive industries it is selected as the environment for developing variant rich models. pure::variants was selected based on the analysis of different tools for SPLE given in [Kajtazović, 2011]. The Variant Block Set, described in Section 2.3.3, give a good starting point to provide means to model variability in Simulink. The prototype implemented in this thesis will, therefore, reuse concepts provided by it and extend it to suit the needs implied by the required scenarios. The component models described in Section 2.4, especially the CompAA Model [Lacouture and Aniorté, 2008] and the work presented in [Dauenhauer et al., 2009] (see Section 2.2.4), gave hints how component models reflecting variability have to be designed.

3. Variant Component Model Design

3.1. Requirements to a Component Model for Variant Management

3.1.1. Generation of Valid Variants

The Generation of variants based on a set of core assets is one major goal of Software Product Lines (SPLs). The focus in this thesis is on model based components. Therefore, the core assets here are configurable components implemented in a model based environment that are reuseable for different variants. Beside functionality that all variants may have in common there are aspects that vary from one variant to another. Valid variants compose all these aspects that are needed to fulfill their intended functionality:

Requirement 1 (Minimal Variants): *The component model has to support the application engineer in selecting a valid variant that only aggregates those parts from a generic component that are needed to fulfill the intended behavior of that particular variant.*

This means that no unused functionality should be part of a variant. After deriving a variant from a generic model no further configuration should be required (except post build calibration).

3.1.2. Reusability

Components are subject to reuse across different variants and domains. Therefore, variability information provided by the component needs to be part of it and has to be provided via an interface.

Requirement 2 (Cross Domain Reuse): *Components that provide variability should be reusable across different domains.*

3.1.3. Simulink Integration

Most automotive software is implemented using Simulink. Therefore, a mechanism providing variability to components intended for the automotive domain should be integrated into Simulink.

Requirement 3 (Simulink Integration): *Integrate the mechanism that provide variability in Simulink components.*

Integration means, that variability should be representable directly in Simulink models. Furthermore, variants derived from variant rich Simulink models should be testable within Simulink too. This makes it possible to introduce variability management seamless into a Simulink based development process.

3.1.4. Variation Point Dependencies

Since components are connected to each other a feature selected in Component A may affect availability of features in Component B. This dependencies need to be resolved automatically to prevent application engineers from developing invalid variants.

Requirement 4 (Dependencies): *Dependencies between different components and/or variation points should be reflected in the component model. The component model should provide a mechanism to select consistent variants according to their dependencies.*

3.1.5. Cross Cutting Effects

Cross Cutting [Kiczales et al., 1997] appears if features "*. . . must compose differently and yet be coordinated [Kiczales* et al.*, 1997]*". A famous example for such a feature is logging. Various components may have data to log, on the other hand logging is not needed all the time. Therefore, it should be possible to switch logging on or off without the need to manually adjust components that provide logging. The difference to dependencies is that dependencies are likely to be local, i.e. affect a rather small number of components. Features that do "*Cross Cutting*" affect several components throughout the whole architecture.

Requirement 5 (Reflect Cross Cutting Effects): *The component model should provide a mechanism to reflect cross cutting effects across component boundaries.*

3.1.6. Interoperability with AUTOSAR

AUTOSAR established itself within the automotive industry as an acknowledged standard. Therefore, new developments in the area of automotive software should be compatible to that standard.

Requirement 6 (AUTOSAR Interoperability): *A component model intended to provide variability in automotive software should be compatible to* AUTOSAR*.*

3.2. Separating Concerns: Variation Points and Component Variability

When speaking about component variability two things need to be separated:

- Who controls the variability.
- Who consumes it.

If a component provides variability, information about possible configurations has to be provided. *Variation Points* are used to provide the information which configurations are possible. Components that implement that variants have to provide mechanisms to adjust themselves according to that selection. Because *Variation Points* are domain specific, and components may be reused across different domains, these two entities have to be separated from each other. Therefore, the component model consists of two parts:

Variation Point Model: Contains all variation points and the dependencies between them.

Variant Component Model: Contains the components along with their variability description.

The interaction point between those two models is the interface that is shared between *Variation Points* in the Variation Point Model (VPM) and *Variability Consumers* in the Variant Component Model. This interface, called *VariationPointInterface*, provides information about the name of a variation point, its value set and whether it is enabled or not.

The following sections provide detailed information about the two models introduced here and how they process the information provided by implementations of the common interface for variation points introduced above.

3.3. Variation Point Model

The name of this model is borrowed from [Webber and Gomaa, 2004] but, as this section will show, takes a completely different approach. The responsibility of this model is to store all variation points and the dependencies between them. This model corresponds conceptually to a Feature Model which is used by the application engineer to develop valid variants as mentioned in Section 3.1.1. Therefore, variation points should be related to features identified in an earlier step during domain engineering.

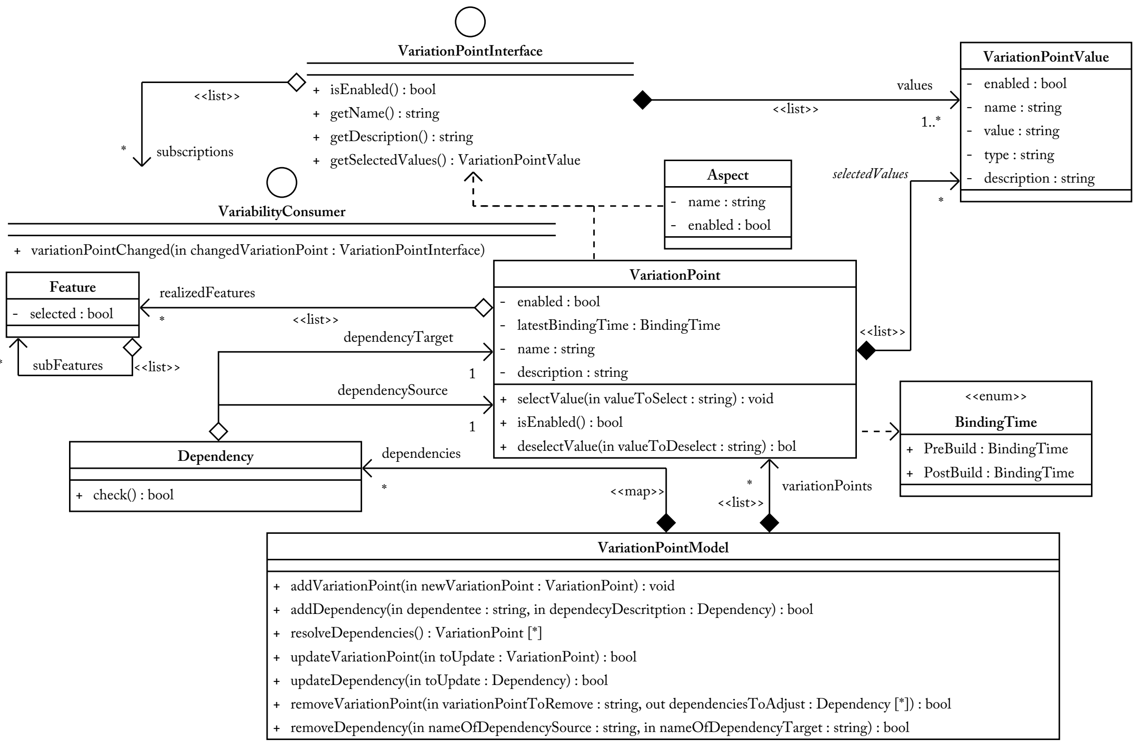

VariationPointInterface
+ isEnabled() : bool
+ getName() : string
+ getDescription() : string
+ getSelectedValues() : VariationPointValue
VariationPointValue
- enabled : bool
- name : string
- value : string
- type : string
- description : string
values
<<list>>
1..*
selectedValues
*
<<list>>
*
subscriptions
VariabilityConsumer
+ variationPointChanged(in changedVariationPoint : VariationPointInterface)
Aspect
- name : string
- enabled : bool
Feature
- selected : bool
realizedFeatures
*
<<list>>
subFeatures
VariationPoint
- enabled : bool
- latestBindingTime : BindingTime
- name : string
- description : string
+ selectValue(in valueToSelect : string) : void
+ isEnabled() : bool
+ deselectValue(in valueToDeselect : string) : bol
dependencyTarget
1
dependencySource
1
<<enum>>
BindingTime
+ PreBuild : BindingTime
+ PostBuild : BindingTime
Dependency
+ check() : bool
dependencies
<<map>>
variationPoints
VariationPointModel
+ addVariationPoint(in newVariationPoint : VariationPoint) : void
+ addDependency(in dependentee : string, in dependecyDescritption : Dependency) : bool
+ resolveDependencies() : VariationPoint [*]
+ updateVariationPoint(in toUpdate : VariationPoint) : bool
+ updateDependency(in toUpdate : Dependency) : bool
+ removeVariationPoint(in variationPointToRemove : string, out dependenciesToAdjust : Dependency [*]) : bool
+ removeDependency(in nameOfDependencySource : string, in nameOfDependencyTarget : string) : bool

Figure 3.1.: The structure of the Variation Point Model

The VPM, illustrated in Figure 3.1, consists of two major parts: The *Variation Point* and its dependencies. The following sections provide details on these parts, explaining their structure, relationship and behavior. The last section of this chapter provides details about the operations that can be executed on the VPM.

3.3.1. Variation Points

Variation Points are the basic building blocks of the Variation Point Model. As mentioned in Section 2.1.2 a *Variation Point* consists of the following basic items:

- A name,
- a set of values,
- the set of currently selected values and
- the binding time, indicating when a particular value needs to be selected.

A *Variation Point* is identified by its name. This name can be chosen freely but must be unique in the VPM.

Values identify the different variants a *Variation Point* can configure. Therefore, such values, called *Variation Point Value (VPV)* are the basic building blocks of each *Variation Point.*

As illustrated in Figure 3.1, a *Variation Point Value* in the VPM consist of five parts:

The name identifies the value in the value set of a variation point.

The data type indicates the type of the *Variation Point Value.*

The visibility indicates whether the *Variation Point Value* is selectable or not. If a value is marked as invisible it can not be selected in the particular configuration. This can happen due to dependencies (see Section 3.3.2 for more details) the corresponding *Variation Point* may have.

The concrete value contains the actual data the *Variation Point Value* stores.

The description contains informal documentation about the *Variation Point Value.*

In the VPM it is possible to select multiple values simultaneously in one *Variation Point.* This can be used to implement *Variation Points* for *m:n* feature sets as provided by the *or* operator for features as defined in [Kang et al., 1990].

Variation Point Values aggregated by a *Variation Point* must fulfill the following constraints:

Constraint 1: *Valid Variation Point Values must have set their name, data type and concrete values.*

Constraint 2: *Names used for Variation Point Values must be unique inside the value set of the corresponding Variation Point.*

Constraint 1 is needed because the VPM has to prevent invalid variants from being developed. If *Variation Point Values* would be allowed to e.g. have no concrete value, ambiguities between different variants can happen. If the data type of a *Variation Point Value* is missing, proper validation of *Variation Points* with a binding time set to `PostBuild` can not be done. Unnamed *Variation Point Values* may lead to ambiguities when binding the corresponding *Variation Point* to a concrete value. Constraint 2 is required because values must be uniquely identifiable inside the value set of their corresponding *Variation Point.* Ambiguous names make it impossible to validate a value selection done for a particular *Variation Point.*

Because *Variation Points* may depend on other *Variation Points*, as shown in Section 3.3.2, they need a mechanism to be completely excluded from the variant derivation process. To achieve this, *Variation Points* in the VPM can be enabled and disabled. A disabled *Variation Point* can not be bound to a specific value and, therefore, has no impact on any variant unless it is re-enabled.

The information about the name of the *Variation Point*, its value set and if it is enabled or not is communicated through implementing the *VariationPointInterface* interface (see Section 3.2).

The implementation of this interface for the VPM further stores the latest possible binding time for a particular *Variation Point.* The binding time can be used to verify that all *Variation Points* are bound at the correct time before concrete variants are generated. Binding a *Variation Point* is done by selecting a value out of its value set. Furthermore, selecting a value starts resolving dependencies as shown in Section 3.3.2.2.

3.3.1.1. Associating Variation Points to Features

Since *Variation Point* serve as technical realizations for features provided by a Feature Model they need to be related to those feature. Section 2.2.1.1 shows the different types of features. These types need to be mapped differently. Relating features to *Variation Points* makes it easier to develop valid variants as required by Requirement 1 in Section 3.1.1 because it reduces the need of manually adjusting *Variation Points.*

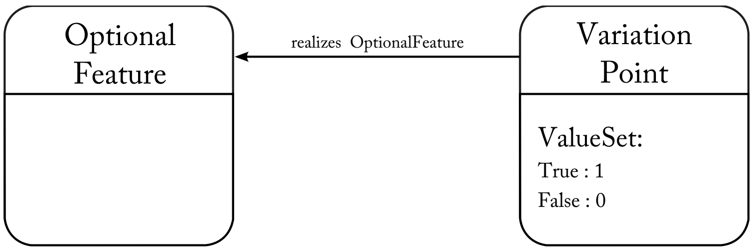
Mapping Mandatory Features Mandatory feature do not need to be mapped to *Variation Points* at all. These features are present in all variants and, therefore, no *Variation Point* needs to control them.

Mapping Optional Features Mapping optional features can be done directly if the corresponding *Variation Point* provides a value set as shown in Figure 3.2a.

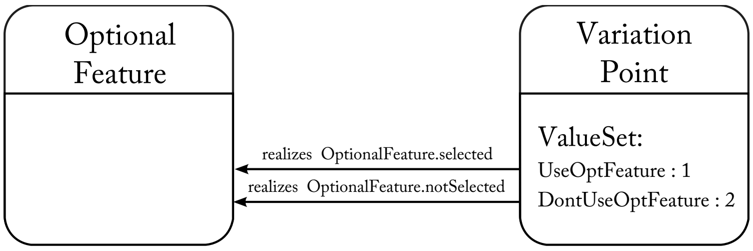
In this case, the *Variation Point* just needs to be connected to the optional feature.

If the mentioned value set can not be used, *VPVs* need to be chosen from the *Variation Point* that can be mapped to the selection state of the desired feature. This ability is illustrated in Figure 3.2b

Mapping Alternative Features Mapping alternative features is a generalization of mapping the selection state of optional features to special *VPVs.* When mapping alternative features to a *Variation Point* each alternative needs to be mapped to a single *Variation*



(a) Automated Mapping



(b) Selective Mapping

Figure 3.2.: Mapping of optional features can be done automatically or selectively

Point Value provided by the set of *Variation Point Values*. Mapping of alternative features is illustrated in Figure 3.3.

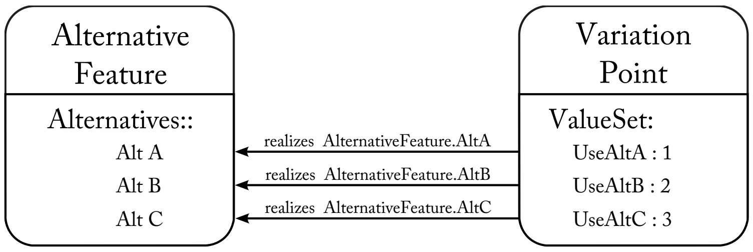


Figure 3.3.: Mapping of alternative features to variation points

Mapping of features that are combined with the *or* operator is done the same way. For each feature available in a *or* linked feature set one value from the value set of the *Variation Point* is selected. If the feature is selected the appropriate value from the *Variation Point* is added to its set of currently selected values.

3.3.2. Dependencies

Based on Requirement 4 defined in Section 3.1.4 *Variation Points* can affect each other. This information is provided via *Dependencies*. Before defining how dependencies behave and how they are validated it is important to define the different parts which form a dependency.

Definition 1 (Dependency) Any relationship between two *Variation Points*, that defines how one *Variation Point* has to be adjusted, if another *Variation Point* changes its value.

Definition 2 (Dependency Source) A *Variation Point* whose selected value affects other *Variation Points.*

Definition 3 (Dependency Target) The *Variation Point* that needs to be adjusted if its dependency source changes its value.

Figure 3.4 gives an overview how this concept is accomplished in the VPM.

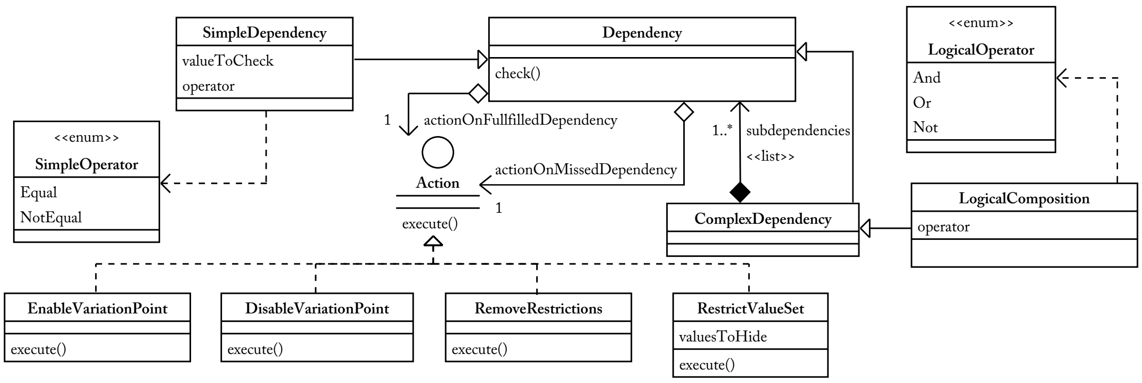


Figure 3.4.: UML diagram showing the different entities that build the dependency concept in the Variation Point Model

Generally, dependencies are defined by selecting their source and target and by defining their actions. In the simplest case a particular *Variation Point* depends on a concrete value selection in another *Variation Point.* In this case the dependency only needs to store the name of that concrete value and can compare the actual value of its dependency source with this value. The behavior of the dependency in case it was either fulfilled or missed is stored in so called *Actions.* The VPM currently supports two logic groups[1] of actions, both containing two implementations to serve both cases of either fulfilled or missed dependencies. Table 3.1 lists these groups with a description of the actions belonging to them.

Action group	**Action**	**Behavior**
Enabling/Disabling	*EnableVariationPoint*	Enables a *Variation Point.*
	DisableVariationPoint	Disables a *Variation Point.*
Change Value Set	*RestrictValueSet*	Restricts the set of values selectable for a *Variation Point.*
	RemoveRestrictions	Makes all values of a *Variation Point* selectable.

Table 3.1.: Actions provided by the VPM to handle different dependency cases

[1]There exists no class or other entity representing action groups

3.3.2.1. Logically Composed Dependencies

Simple dependencies in the VPM can be used to compare required and actual values of *Variation Points*. To be able to define more complex dependencies the VPM provides the ability to compose simple dependencies with logical operators into more complex ones. This is done by aggregating simple dependencies into a new dependency object that combines the evaluation result of each aggregated dependency with a logical operator. Table 3.2 lists all operators available for this task along with their possible number of arguments and their meaning.

Operator	**Number of Arguments**	**Meaning**
And	>= 2	True if all nested dependencies evaluate to true, false otherwise
Or	>= 2	True if at least one of the nested dependencies evaluates to true, false otherwise
Not	1	Inverts the result of the nested dependency

Table 3.2.: Logical operators provided by the VPM to develop more complex dependencies

3.3.2.2. Resolving Dependencies

Resolving dependencies is emitted every time the set of selected values of some dependency source changes.

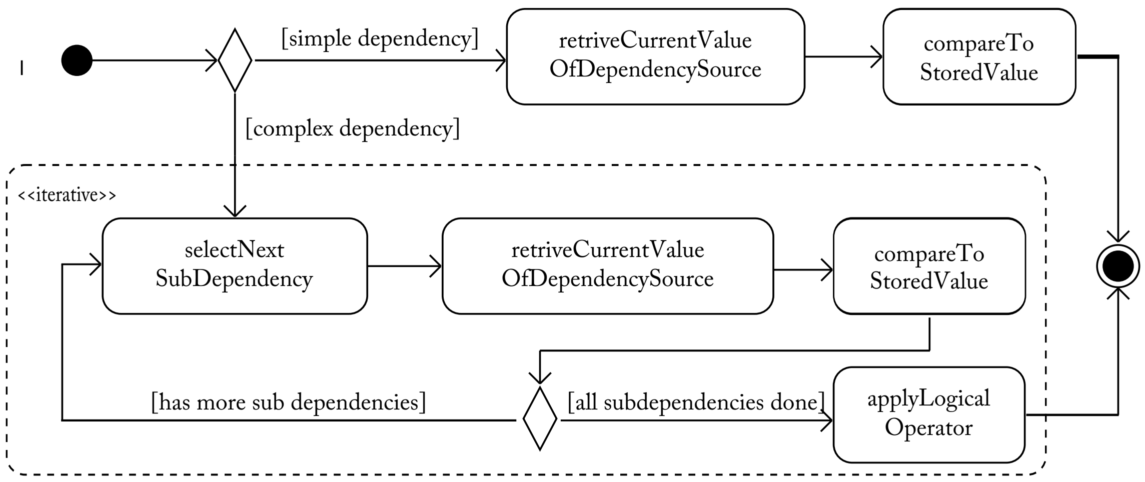


Figure 3.5.: An UML activity diagram showing the process of resolving a dependency

The first step when resolving dependencies is to check if they are fulfilled or not. In case of simple dependencies this is done by checking if the set of selected values in the dependency source contains the value stored in the dependency. If the dependency is complex each of its nested dependencies need to be evaluated. How the evaluation results of all nested dependencies are combined, is indicated by the logical operator the complex dependency uses. In any case, the result of this evaluation process is a boolean value indicating if the dependency is fulfilled or missed. Figure 3.5 illustrates this workflow. For each case the dependency provides a separate action. Invoking these actions while resolving dependencies is done by the VPM as shown in Section 3.3.4.4.

3.3.3. Aspects

According to [Kiczales et al., 1997] an *Aspect* is a part of a system that "*can not be cleanly encapsulated in a generalized procedure*". Aspects are required by Requirement 5 in Section 3.1.5. Technically speaking there is no difference between *Variation Points* and *Aspects*. Both provide the ability to either include or exclude a certain feature. The main difference between this two concepts is their usage. *Variation Points* are used quite locally, controlling features in one or at least in a small set of components. *Aspects* on the other hand have a much bigger impact on variants that either use them or not. *Aspects* normally control a larger set of components, maybe even all components, throughout the whole architecture. Contrary to *Variation Points* that are connected to components, *Aspects* are selected by components that want to implement them. This makes the usage of *Aspects* much easier when modeling components and system architectures that use them.

Because of this major difference in terms of usage, *Aspects* are elements on their own in the VPM. The same constraints defined for *Variation Points* also apply to *Aspects*, except that *Aspects* are allowed to have no value set. However, *Aspects* may define a value set to provide the ability to configure them too. A logger aspect, as mentioned in Section 3.1.5, may define a value called `logLevel` to configure how much information should be logged by each component.

3.3.4. Operations on the Variation Point Model

Since the VPM is responsible for managing *Variation Points* and providing valid variants it needs some operations to provide those functionalities. To keep the model valid at all times at least one constraint must be fulfilled after every change to the model:

Constraint 3: *All Variation Points need to have unique names.*

Unique names provide the ability to clearly identify each *Variation Point* by its name.

3.3.4.1. Adding Variation Points

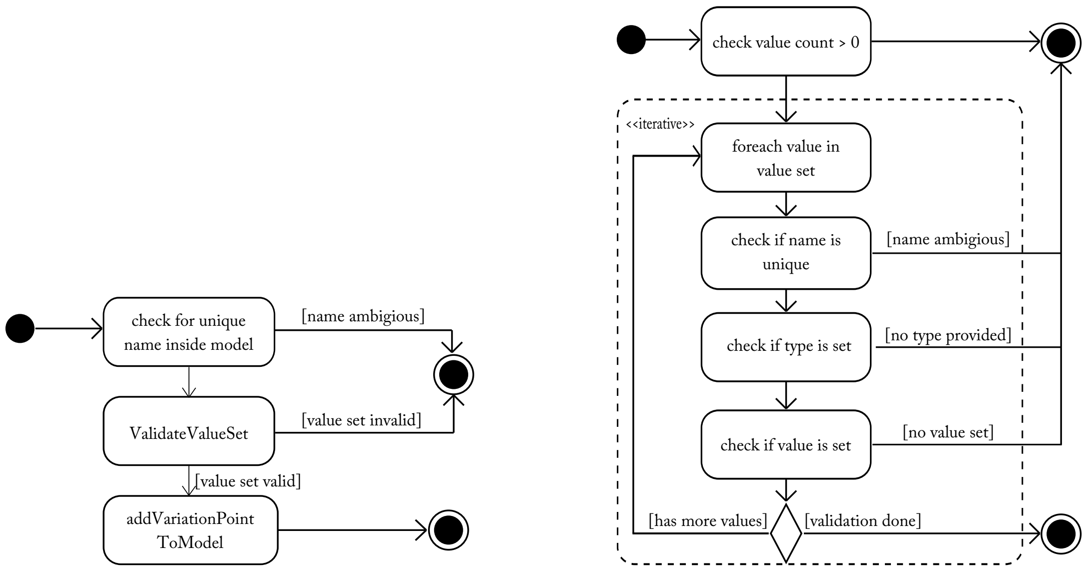
Before a new *Variation Point* can be added to the model it has to fulfill some constraints that are enforced by the model.

Constraint 4: *If a Variation Point is added, it must not exist a priori in the VPM.*

Constraint 5: *The size of the value set must be greater than zero.*

A *Variation Point* without any values makes no sense. At least one variant is always selectable, therefore, a *Variation Point* without any values is not allowed in a valid VPM. Figure 3.6b illustrates how the value set of a *Variation Point* is validated.

Figure 3.6a illustrates the work flow describing how a *Variation Point* is added to a valid VPM and shows when it is rejected by the model.



(a) Adding the Variation Point (b) Validating the Value Set

Figure 3.6.: Adding a *Variation Point* to the VPM is done after checking the constraints the *Variation Point* and its value set must fulfill

3.3.4.2. Updating a Variation Point

If a *Variation Point* changes, i.e. the selected value changes, the model needs to be informed about that change. Therefore, the VPM provides a method called `updateVariationPoint`. The task of this method is to revalidate the provided *Variation Point* and then to update its data in the model. Updating means that data of an existing *Variation Point* needs to be changed:

Constraint 6: *A Variation Point that needs to be updated must be present in the model.*

Figure 3.7 illustrates how a particular *Variation Point* is updated in the VPM.

3.3.4.3. Adding Dependencies between Variation Points

Dependencies, as described in Section 3.3.2, are stored independently from their sources and targets.

Dependencies in the VPM are stored in relation to their dependency sources. For both, the dependency source and target, the following constraint must be fulfilled:

Constraint 7: *Both, dependency source and dependency target, must exist in the VPM before adding a dependency between them.*

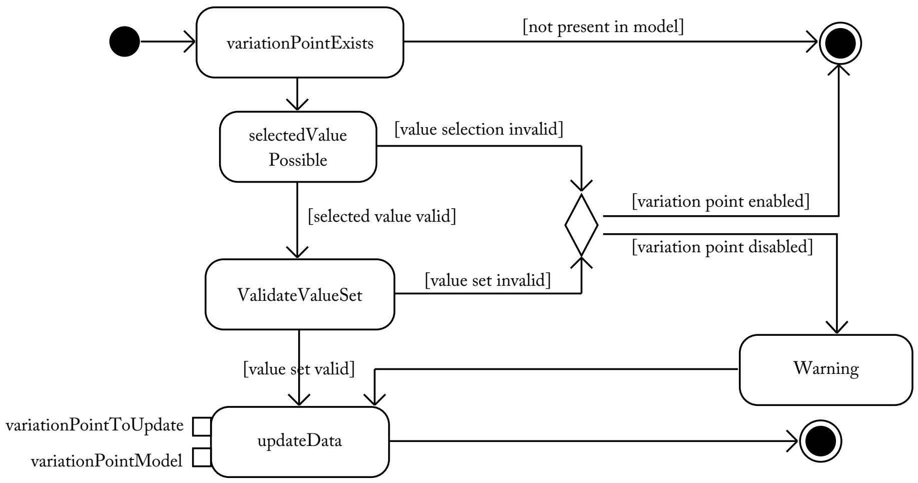


Figure 3.7.: The work flow to update a *Variation Point* in the VPM

If, for example, the dependency source would not exist, the dependency had no value to decide if it is fulfilled or not. On the other hand, if the dependency target is missing, a dependency is not necessary, because no behavior in conjunction with other *Variation Points* can be implemented. Therefore, dependencies are only added if both parts, dependency source and target, exist in the model. Otherwise the dependency is rejected.

Constraint 8: *Values referenced in a dependency must exist in the referenced Variation Point.*

Since dependencies act on values placed in either the dependency source or in the dependency target, the corresponding *Variation Points* must provide those values. If dependencies would be allowed to reference non present values it may happen that resolving them always fails and, therefore, the model produced is invalid. To avoid this, referenced values are validated before adding a dependency to the VPM.

Figure 3.8 shows validation of dependencies.

3.3.4.4. Resolving Dependencies

Resolving dependencies needs to be done every time the set of selected values in a dependency source changes. This is necessary to ensure valid variants at all times as required in Requirement 1 and to support the application engineer during variant derivation. If a *Variation Point* changes, all dependencies using it as their dependency source are collected and are checked using the work flow defined in Section 3.3.2.2. Depending on the result of this check the appropriate action provided by the dependency is invoked. as a result, *Variation Points* are always valid according to the constraints defined for them. Figure 3.9 illustrates the dependency resolution.

3.3.4.5. Remove Dependencies

During Domain Engineering it may happen that previously defined dependencies are not needed anymore in a later iteration. Furthermore, it may happen, that *Variation Points*

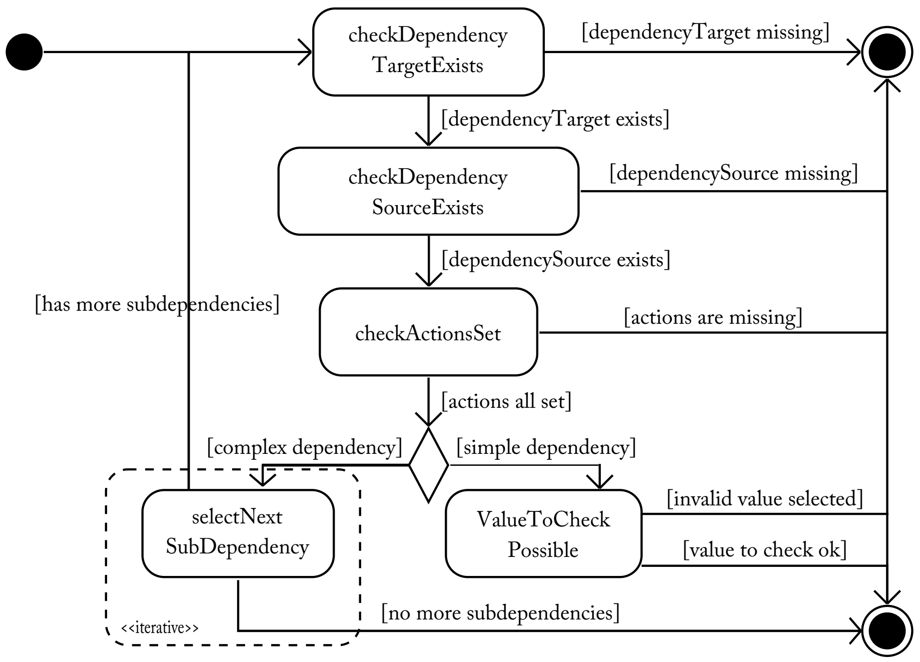

checkDependency TargetExists
[dependencyTarget missing]
[dependencyTarget exists]
checkDependency SourceExists
[dependencySource missing]
[dependencySource exists]
[has more subdependencies]
checkActionsSet
[actions are missing]
[actions all set]
[complex dependency]
[simple dependency]
selectNext SubDependency
ValueToCheck Possible
[invalid value selected]
[value to check ok]
<<iterative>>
[no more subdependencies]

Figure 3.8.: The work flow describing how a dependency is validated before adding it to the model

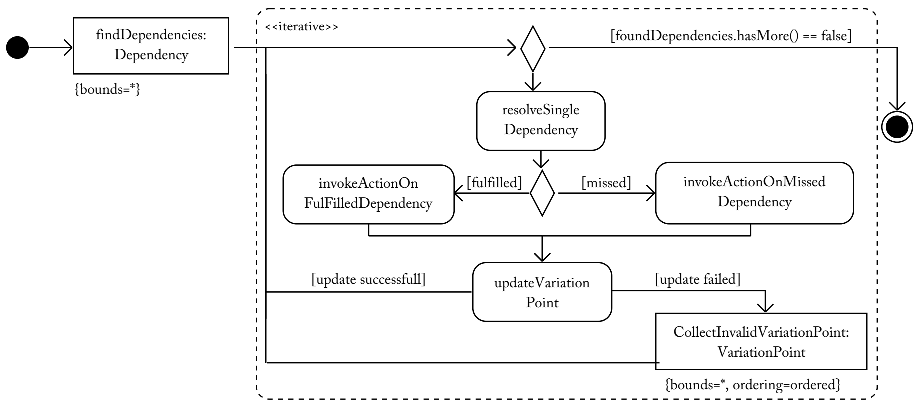

findDependencies: Dependency
{bounds=*}
<<iterative>>
[foundDependencies.hasMore() == false]
resolveSingle Dependency
invokeActionOn FulFilledDependency
[fulfilled]
[missed]
invokeActionOnMissed Dependency
[update successfull]
updateVariation Point
[update failed]
CollectInvalidVariationPoint: VariationPoint
{bounds=*, ordering=ordered}

Figure 3.9.: The work flow describing how dependencies in the VPM are resolved

should be removed from the model that either act as dependency source or target. Therefore, it is necessary to provide the ability to safely remove dependencies from the model.

A dependency can be uniquely identified by its source and target *Variation Points*. This fact can be used to locate the dependency that should be removed in the VPM. First, all dependencies that use the provided dependency source are collected. If such dependencies exist, the dependency referencing the given target is selected. Since dependencies manipulate their targets deleting them affects those targets too. Therefore, before removing a dependency from the model its dependency target is reseted. After resetting a *Variation Point* it is enabled and all of its values are selectable again. Therefore, any changes a dependency may have done to a *Variation Point* are reverted. Figure 3.10 illustrates the steps used to remove a dependency from a VPM.

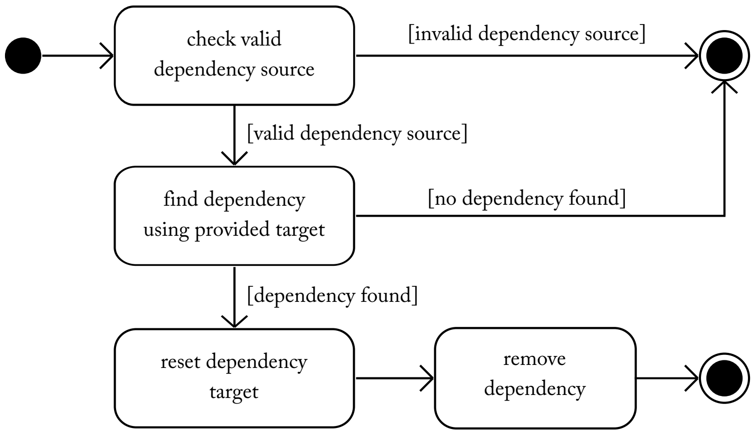

check valid dependency source
[invalid dependency source]
[valid dependency source]
find dependency using provided target
[no dependency found]
[dependency found]
reset dependency target
remove dependency

Figure 3.10.: Required steps to remove a dependency from the VPM

3.3.4.6. Remove Variation Points

Removing a *Variation Point* has a major impact on *Variation Points* that depend on it. Because Constraint 7 requires that both dependency source and target must exist to form a valid dependency, dependencies referencing the removed *Variation Point* need to be adjusted. Therefore, all dependencies referencing the removed *Variation Point* are collected by this operation. If those dependencies are either removed or just adjusted depends on the needs of the engineering process. Figure 3.11 illustrates the work flow of the remove operation.

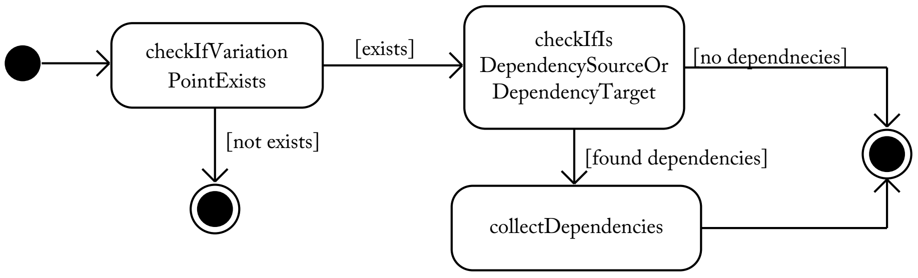


Figure 3.11.: The work flow how to remove a *Variation Point* from the VPM

3.4. Components with Variability

3.4.1. Development of Variant Rich Components

The development of variable components can be decomposed into three steps which are illustrated in Figure 3.12.

Before building a component that provides variability the required functionality has to be identified. This functionality can be provided by several other, smaller granulated,

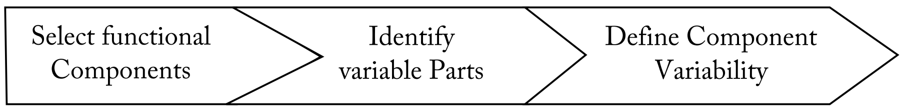


Figure 3.12.: The development process for components that provide variability

components or has to be implemented from scratch. The component model refers to these "smaller granulated components" as *Functional Components*.

The next step is to separate variable from common functionality. Common functionality is always part of the component independent of any variant selection. The component model proposed in this thesis supports the component developer to reflect the variable part of a component during its development.

The final step is to define the variability of the new component. This step can be decomposed into two sub steps illustrated in Figure 3.13.

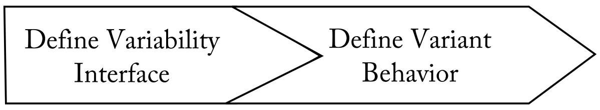


Figure 3.13.: Steps to define variability for a component

The first of these steps is to define the configuration interface that the new component should provide to configure the different variants. This interface is shared between components and variation points as shown in Section 3.3.1. It consists of a set of values identifying the different variants provided by the component. After defining this interface the values need to be mapped to the appropriate parts of the component. The component model described here refers to these mappings as *Connectors* which are shown in more detail in Section 3.4.5. To store such *Connectors* and actually map one connector to a specific value, a special entity called *VariabilityRealizer* is used which is described in Section 3.4.4.1.

Figure 3.14 gives an overview of the structure of the Variant Component Model.

The result of the process shown is always a new component aggregating functional components and variability information as illustrated in Figure 3.15.

3.4.2. Terminology

Before providing more detail on the Variant Component Model itself, some terms used in the subsequent sections need to be defined.

At first the core of each component model needs to be defined, the component:

Definition 4 (Component) "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" [Szyperski, 1997]

Following this definition, a component needs to be developed decoupled from its environment and possible other components [Crnkovic, 2002]. The only way a component can communicate to its environment is via its interface.

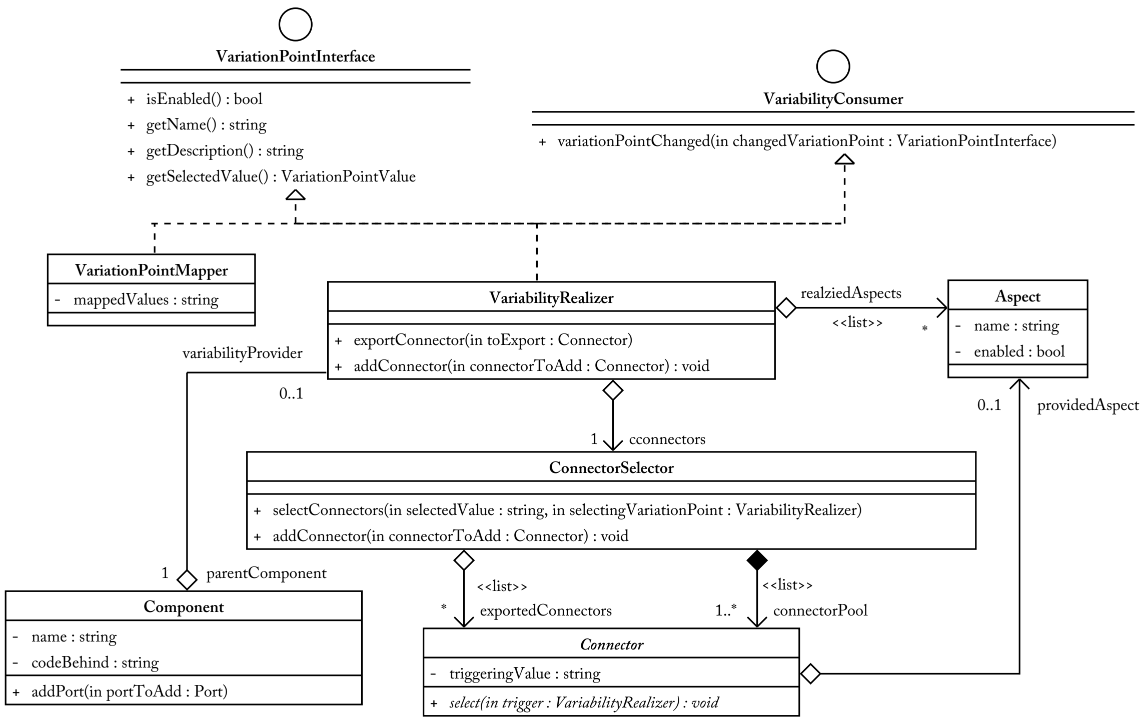

VariationPointInterface
+ isEnabled() : bool
+ getName() : string
+ getDescription() : string
+ getSelectedValue() : VariationPointValue
VariabilityConsumer
+ variationPointChanged(in changedVariationPoint : VariationPointInterface)
VariationPointMapper
- mappedValues : string
VariabilityRealizer
+ exportConnector(in toExport : Connector)
+ addConnector(in connectorToAdd : Connector) : void
realziedAspects
<<list>>
*
Aspect
- name : string
- enabled : bool
variabilityProvider
0..1
0..1
providedAspect
1
cconnectors
ConnectorSelector
+ selectConnectors(in selectedValue : string, in selectingVariationPoint : VariabilityRealizer)
+ addConnector(in connectorToAdd : Connector) : void
1
parentComponent
<<list>>
*
exportedConnectors
<<list>>
1..*
connectorPool
Component
- name : string
- codeBehind : string
+ addPort(in portToAdd : Port)
Connector
- triggeringValue : string
+ select(in trigger : VariabilityRealizer) : void

Figure 3.14.: A class diagram showing the structure of the Variant Component Model

Definition 5 (Compositional Component) A *Compositional Component* is a component developed by selecting several smaller granulated components and weaving them together into a new component [Sora et al., 2004].

Definition 6 (Subcomponent) A *Subcomponent* is a component aggregated by a *Compositional Component.*

Component composition is the heart of the proposed component model, since variability of components is achieved by selecting subcomponents and weaving them into a new compositional component along with the description of variability this new component provides.

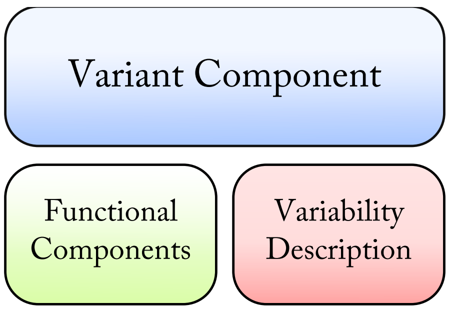


Figure 3.15.: Combining functional components with variability results in a new component.

Definition 7 (Aspect Implementer) An *Aspect Implementer* is a component that implements a certain aspect of the system, as defined in Section 3.3.3.

Aspect Implementers are used by components that provide functionality required by aspects defined for the software system.

3.4.3. Component Structure

Based on the development process shown in Section 3.4.1 the structure of components needs to be defined.

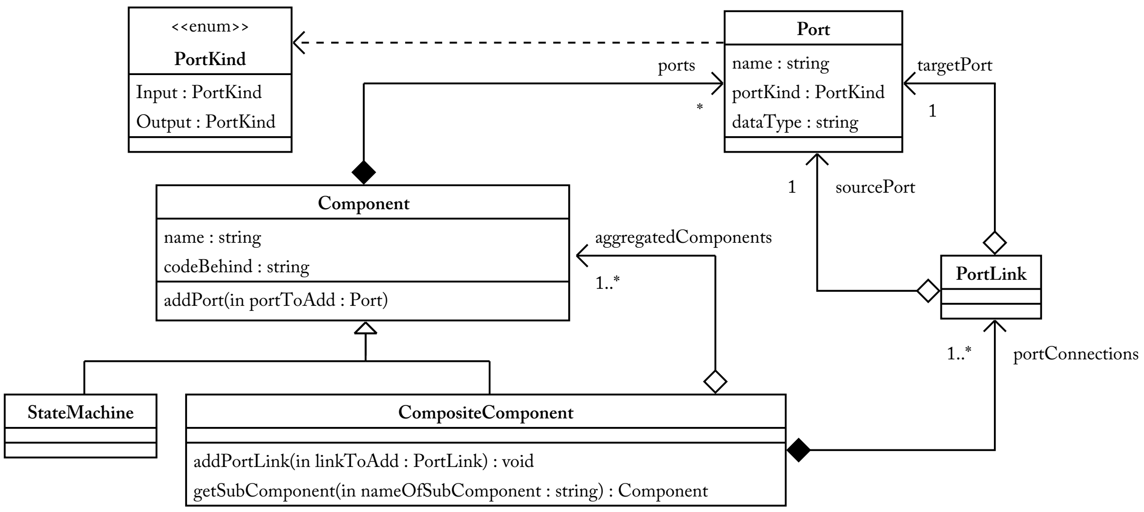


Figure 3.16.: The structure of components in the Variant Component Model

At its very core each component has to provide a name and an interface. The interface provides information about how a component can be embedded into a bigger system. Basically, this interface has to be split into two parts [Crnkovic, 2002]:

Required Interface: Information that is needed by the component to fulfill its task.

Provided Interface: Information that is provided by the component to other parts of the system.

Based on the port concept defined in [Autosar, 2009e] the required and provided interfaces are modeled as ports. Inside the Variant Component Model each port knows by itself if it is required or provided.

Compositional components aggregate several other components to form a new one. As stated in Section 3.4.1, compositional components are subject to variability. When designing components to take part in a SPL based development process a third part in their interface is required: the configuration interface. The configuration interface provides the information about different variants that can be served by the component. This configuration interface is realized by the *VariabilityRealizer* concept described in Section 3.4.4.1. To connect ports used in subcomponents to ports exposed in the interface of a compositional component so called *PortLinks* are used. *PortLinks* connect ports to indicate the flow of data between the compositional component and its subcomponents.

Because the component model does not reflect implementation details, these details need to be provided elsewhere. To link the component descriptions to their corresponding implementations, each component stores an identifier called *codeBehind* that serves as a link to the implementation of the component.

3.4.3.1. AUTOSAR Interoperability

The structure of components used in this approach is very similar to the one used in [Autosar, 2009e]. Since AUTOSAR does not define how component implementations need to be represented too, the component description used can be directly mapped to AUTOSAR component interface descriptions represented as XML files.

3.4.3.2. State Machines

As stated in Section 1.2.3, state machines can be seen as a special form of compositional components. The main difference is, that state machines can not aggregate any component. The possible subcomponents are limited to states and transitions. States and transitions are very specific components too. They can only aggregate each other. Normally, the direction of this aggregation is not important. For the proposed component model it was defined that states aggregate transitions. This makes it easier to handle variability issues with optional states in a particular state machine. Optional states need transitions that integrate them into a state machine. If the state is omitted these transitions have to be replaced with transitions that bypass the optional state. Referencing incoming and outgoing transitions in a state makes it very simple to find those transition that need to be replaced. Figure 3.17 illustrates the structure used for state machines.

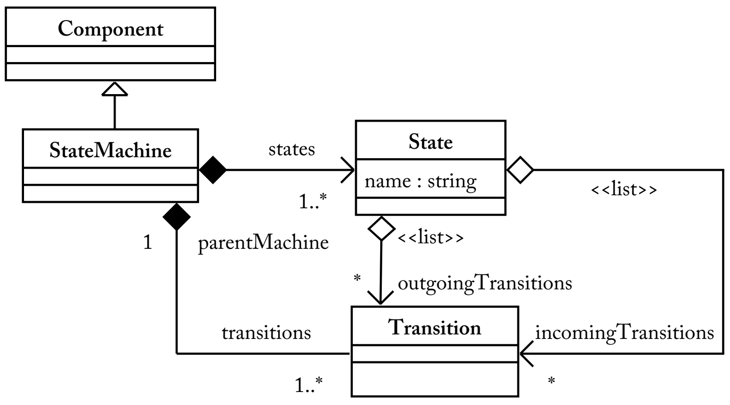


Figure 3.17.: The structure of state machines used in the Variant Component Model.

The Variant Component Model does not validate the provided state machines. This task is delegated to state machine development environments (e.g. Simulink Statecharts). Therefore, like in ordinary components, state machines in the Variant Component Model use the *codeBehind* identifier mentioned earlier, to reference those implementations.

3.4.4. Variability Consumers

The *VariabilityConsumer* concept provides an interface between components and variation points. Therefore, each component defines its own *Variation Point*, a behavior similar to private Feature Models required by [Reiser et al., 2009]. These component wide *Variation Points* also act as the configuration interface provided by that component as required by [de Jonge, 2004].

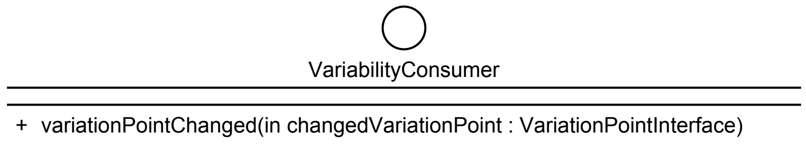

VariabilityConsumer
+ variationPointChanged(in changedVariationPoint : VariationPointInterface)

Figure 3.18.: The interface to connect components to variation points

Connecting a *VariabilityConsumer* to a *VariationPoint* means that the *VariabilityConsumer* subscribes to the *VariationPoint*. If the currently selected value of a *VariationPoint* changes, it informs all its subscribers using the method defined in the interface. This realization of the Observer design pattern [Gamma et al., 1995] makes it possible to decouple variation points from the configurable components.

The proposed component model currently supports two realizations of this interface described in more detail in Sections 3.4.4.1 and 3.4.4.2: *VariabilityRealizers* and *VariationPointMappers*.

3.4.4.1. Variability Realizers

The *VariabilityRealizer* combines the roles of providing the configuration interface and mapping values defined in that interface to connectors that select and/or configure subcomponents. Therefore, *Variability Realizers* are responsible for the adaptivity of their components and are based on the *Adoption Point* concept provided in [Lacouture and Aniorté, 2008].

Providing the configuration interface is done through implementing the *VariationPointInterface* also implemented by *VariationPoints* as defined in Section 3.3.1. Therefore, the same rules relevant for value sets in *VariationPoints* apply to value sets in *VariabilityRealizers*. To store the different connectors the *VariabilityRealizer* uses a pool of them. The so called *ConnectorPool* does the actual mapping between values from the value set and connectors.

To successfully connect a *VariationPoint* to a *VariabilityRealizer* their value sets must be compatible to each other.

Definition 8 (Compatible Value Sets) Two value sets are compatible if at least all values of one set are present in the other set.

Applying *Definition 8* to *VariationPoints* and *VariabilityRealizers* means that the used *VariationPoint* needs to support at least all values required by the used *VariabilityRealizer*. If this condition is not fulfilled, the corresponding entities can not be connected.

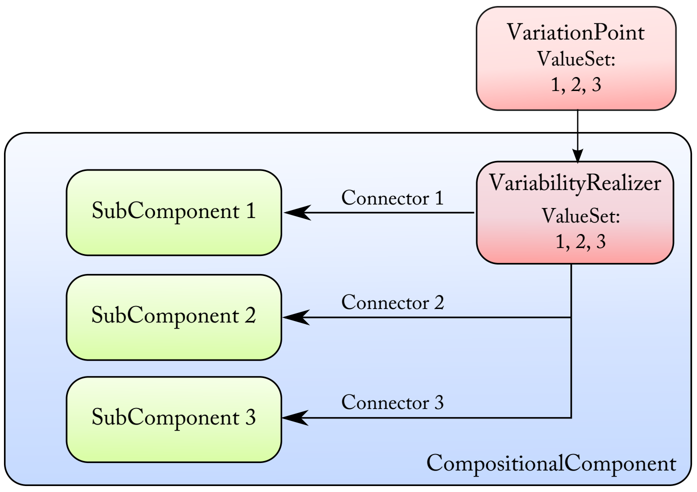


Figure 3.19.: Illustration showing how *VariabilityRealizers* and *Connectors* are used to implement variability in a component

3.4.4.2. Variation Point Mapper

Since value sets of *VariabilityRealizers* need to be compatible to the one provided by *VariationPoints*, reuse of components raises a problem. Its quite unlikely that variation points used in one project are entirely equal to variation points used in another project. Therefore, it is quite unlikely to meet the value set required by a particular components configuration interface when reusing this component in another project. One approach would be to adjust variation points so that they fit the value set required by the component. This approach may not be appropriate because one variation point may configure different components.

To solve this problem cleanly, the Variant Component Model provides a special variability consumer responsible for mapping physically incompatible value sets, called *VariationPointMapper*. Figure 3.20 illustrates the usage of such mappers.

The mapper acts as an adapter, as described in [Gamma et al., 1995]. When using mappers, *VariabilityRealizers* need to subscribe to them instead of subscribing to the *VariationPoint* directly. The mapper itself subscribes to the *VariationPoint* and informs the *VariabilityRealizer* about new values by calling the method provided by the *VariabilityConsumer* interface.

3.4.5. Connectors

Connectors, an application of the equally named concept provided in [Dauenhauer et al., 2009], provide virtual channels between a *VariabilityRealizer* and parts of the component that are controlled by it. There are two types of "parts" that may be controlled by a *VariabilityRealizer*: alternative subcomponents and subcomponents that provide variability on their own. The second case emerges from *Requirement 2* identified in Section 3.1.2. Reusing components that provide variability may also take place by composing them into a new component. Using such a mechanism makes it possible to extend components with new variants. Such a technique can be compared to extending a base class in object ori-

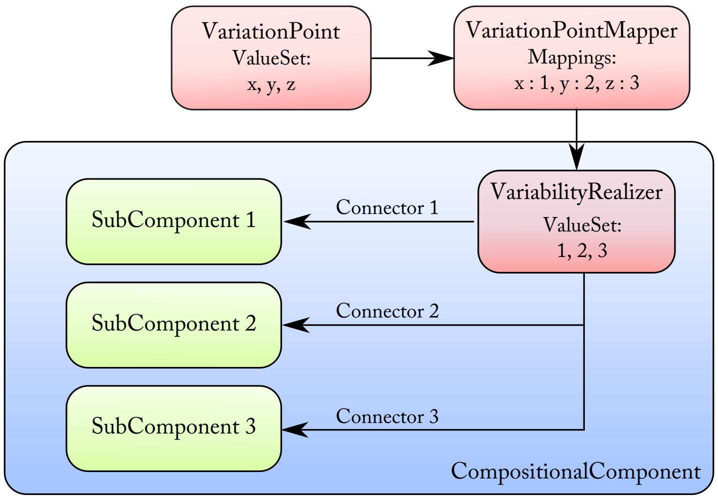


Figure 3.20.: Usage of *VariationPointMapper* to connect heterogeneous value sets

ented programing. To fulfill both use cases the Variant Component Model provides two different connectors: a connector to components, called *ComponentConnector* and one to configuration interfaces of components, i.e. to a *VariabilityRealizer* provided by such components, called *VariationPointConnector*. The structure of these connectors is illustrated in Figure 3.21.

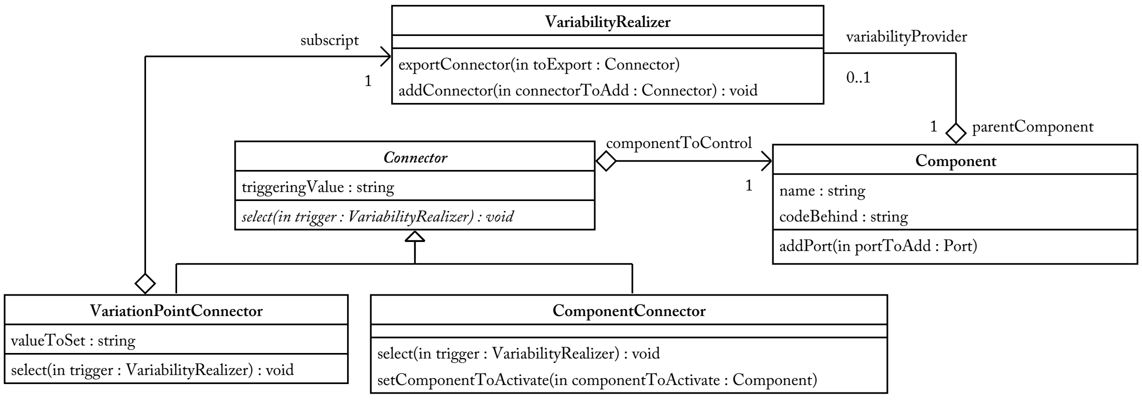

VariabilityRealizer
exportConnector(in toExport : Connector)
addConnector(in connectorToAdd : Connector) : void
subscript
1
variabilityProvider
0..1
1
parentComponent
Connector
triggeringValue : string
select(in trigger : VariabilityRealizer) : void
componentToControl
1
Component
name : string
codeBehind : string
addPort(in portToAdd : Port)
VariationPointConnector
valueToSet : string
select(in trigger : VariabilityRealizer) : void
ComponentConnector
select(in trigger : VariabilityRealizer) : void
setComponentToActivate(in componentToActivate : Component)

Figure 3.21.: Structure of connectors provided by the Variant Component Model to connect variability realizers to variable parts of a component

As mentioned in Section 3.4.4.1 *VariabilityRealizers* aggregate a pool of connectors. If a connector is selected, i.e. the corresponding variant was selected, the pool marks it as “currently selected”. These marked connectors are used to derive the concrete variant of the corresponding component. Figure 3.22 gives an overview on how connectors are validated and added to the connector pool used by the *VariabilityRealizer*.

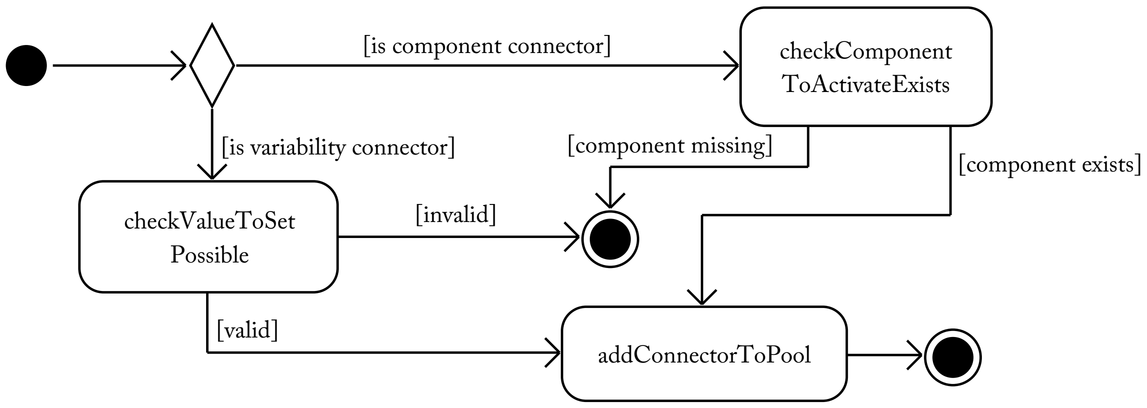


Figure 3.22.: The work flow illustrating validation activities when adding a new connector to the pool used by the *VariabilityRealizer*

3.4.5.1. Component Connector

ComponentConnectors connect *VariabilityRealizers* to components that implement variants provided by the surrounding compositional component. *ComponentConnectors* provide the ability to export interfaces provided by the subcomponents to the interface of the compositional component that provides the variability. This ability is used during the variant derivation process to build a variant specific and usable component. To identify the subcomponent to activate, the connector stores a reference to it. In order to leave a valid variability description Constraint 9 must be fulfilled.

Constraint 9: *The component that is activated by a connector needs to be member of the surrounding compositional component.*

Figure 3.23 illustrates how *ComponentConnectors* are added to *VariabilityRealizers*.

3.4.5.2. Variability Connector

VariabilityConnectors are needed to provide the ability to compose components that provide variability into a new component. *VariabilityConnectors*, therefore, do a similar job as *VariationPointMappers* do with the difference that they get their information from a *VariabilityRealizer* instead of variation points. The mechanism used to establish such a connector is the same as the one used with *VariationPointMappers*. The *VariabilityRealizer* of the subcomponent is a subscriber of the connector. The connector can now inform the subcomponent about a changed variant selection using the *VariabilityConsumer* interface. Figure 3.24 illustrates the steps that are needed to add a new *VariabilityConnector* to a component.

3.4.6. Implementing Aspects

Two parts in the Variant Component Model are affected by *Aspects*: *VariabilityRealizers* and *Connectors*. *VariabilityRealizers* handle variability in a component. Aspects provide variability across component boundaries therefore, *VariabilityRealizers* need to handle

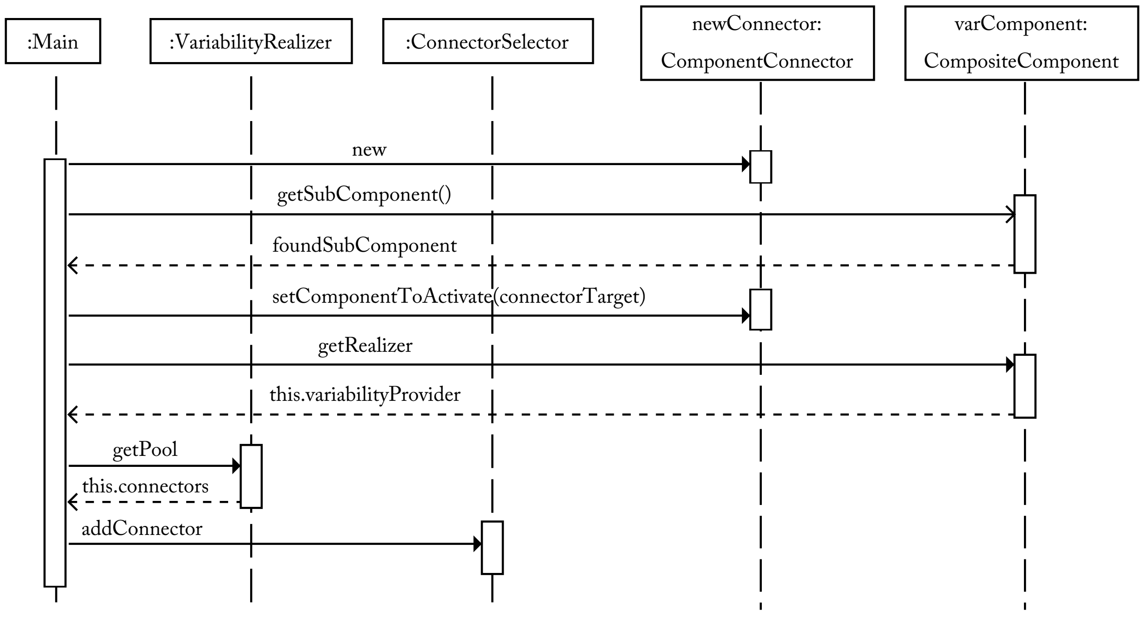


Figure 3.23.: The work flow describing how a *ComponentConnector* has to be added to a *Variability Realizers ConnectorPool*

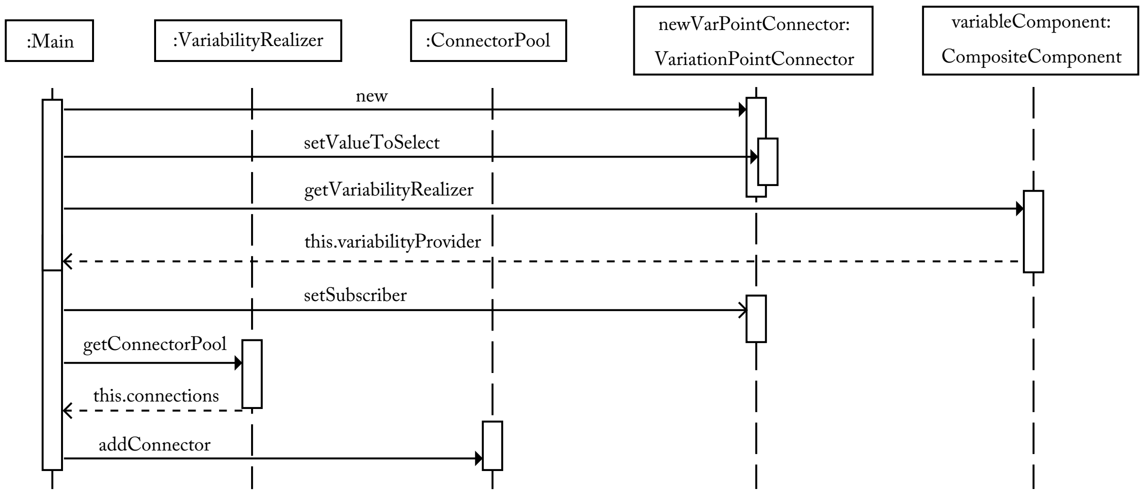


Figure 3.24.: The operation sequence executed to add a new *VariabilityConnector* between a *VariabilityRealizer* and a composed component

Aspects too. Connectors on the other hand serve as selection mechanism for different variants. In context of *Aspects* they provide connections to Aspect Implementers.

VariabilityRealizers store references to all aspects served by the surrounding component directly. To successfully add such an aspect to a *VariabilityRealizer* the following constraint must be fulfilled.

Constraint 10: *An Aspect added to a VariabilityRealizer must be present in the corresponding VPM.*

Figure 3.25 illustrates how an *Aspect* is added to a *VariabilityRealizer*.

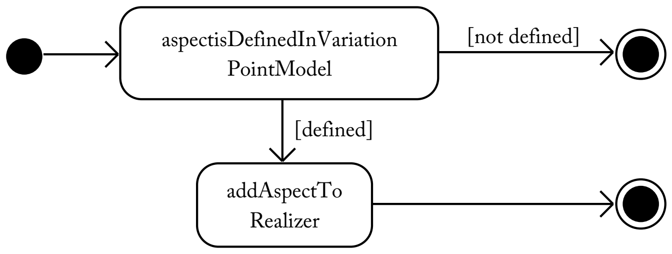


Figure 3.25.: Adding an *Aspect* to be handled by a *VariabilityRealizer*

Connectors reference the *Aspect* that is implemented by the aspect implementer they connect to. Because *VariabilityRealizers* control with aspects are implemented by a component, connectors can only reference those aspects provided by them. Figure 3.26 illustrates how an *Aspect* is set to be referenced by a connector.

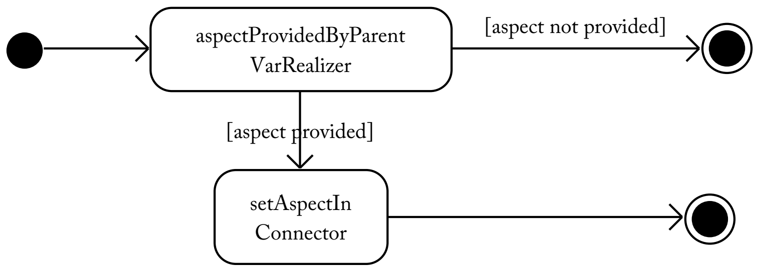


Figure 3.26.: Setting the *Aspect* in a connector that connects to the component that implements it

Technically, Aspect Implementers are equivalent to optional subcomponents. Therefore, like any other component, they can be subject to variability. Selecting variants of Aspect Implementers is only reasonable if the corresponding aspect is active. The *VariabilityRealizer* needs to make sure that variant selection is only applied if this aspect is active. To achieve this, the *VariabilityRealizer* checks before selecting any connector if it depends on an *Aspect*. If such a dependency exits and the corresponding aspect is active, the requested variant can be derived. If the aspect is inactive, i.e. not selected for the particular system, the requested variant is rejected. To safely apply this pattern the following constraint must be fulfilled.

Constraint 11: *All connectors connecting to the same aspect component need to reference the same aspect.*

3.5. Embedding the Variant Component Model into SPLE

After defining the Variant Component Model, this section provides details on how the Variant Component Model can be used inside SPL based software development. This section especially provides information how tasks related to DE and AE need to be done when using the Variant Component Model.

Similar to the situation when using the Variant Block Set to model variability within Simulink and using pure::variants to configure it, the Variant Component Model also needs some tool chain to support it. Furthermore, different tools need different actors and activities in order to produce artifacts exchanged between those tools.

3.5.1. Identifying Roles and Activities

In each Software Development Process different roles and activities assigned to them can be identified. In SPLE two separate processes exist 1) Domain Engineering and 2) Application Engineering.

The following sections describe the different roles identified. Furthermore activities are assigned to that roles. Those activities are mapped to the Variant Component Model and finally, artifacts are described that need to be delivered by executing those activities.

3.5.1.1. Domain Developer

The domain developer has the responsibility to identify features of the domain. According to [Kang et al., 1990] the domain developer categorizes features into different groups as listed in Section 2.2.1 and relates features to each other to describe dependencies between them. By doing this the domain developer develops a Feature Model which can be used by other roles to define the software architecture or to derive concrete variants. The role of the domain developer including his activities is illustrated in Figure 3.27

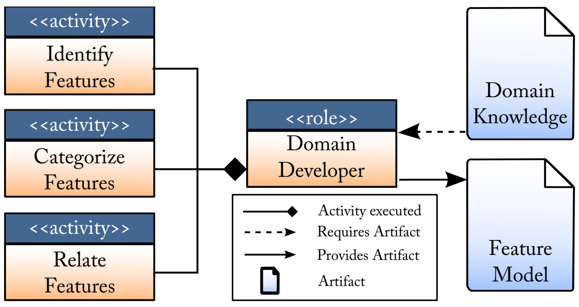


Figure 3.27.: The domain developer providing the Feature Model by identifying and grouping features required by the analyzed domain

The Variant Component Model does not provide abilities to do domain development in the context of feature modeling. The Variant Component Model just references features, provided by a feature model, to assign them to variation points.

3.5.1.2. Variation Point Modeler

The responsibility of the variation point modeler is to define variation points used in the domain model to configure different variants during the variant selection process. The variation point modeler defines the relationship between variation points and features provided by a feature model. Furthermore, the variation point modeler defines dependencies between variation points. The variation point modeler provides the VPM as defined in Section 3.3. Figure 3.28 illustrates the role of the variation point modeler and his activities.

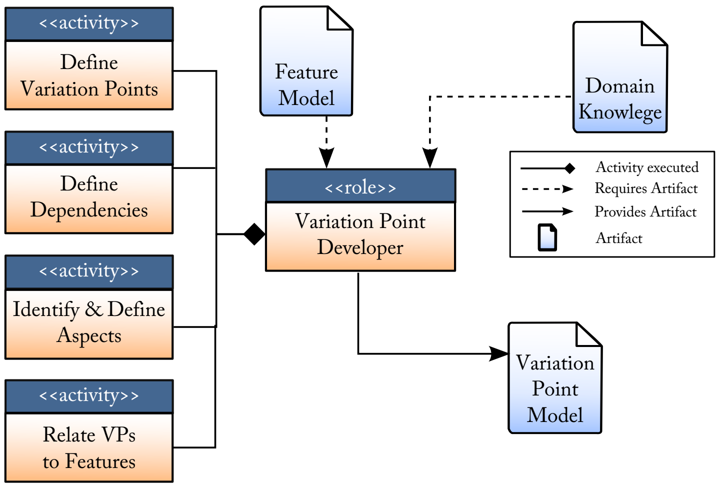


Figure 3.28.: The activities a variation point modeler has to execute

3.5.1.3. Component Developer

The job of the component developer is to deliver components that can be used to develop the system architecture. The component developer selects existing functional components and implements required functionality. After selecting functional components, variability of components needs to be implemented. The VPM provides *Variation Points* to configure those components. The component developer selects functional components according to the identified variability (e.g. alternative components) and composes them into a new component. According to the identified variability, the component developer implements this variability for this component. The component developer delivers a set of components that can be used by the system architect to compose the software architecture. Figure 3.29 shows the different activities of the component developer.

The Variant Component Model provides all necessary items to help the component developer to execute his tasks. *Components* in the Variant Component Model provide the ability to reference implementation details and to store the interfaces a component requires and provides. *VariabilityRealizers* and *Connectors* provide the ability to model and implement variability of components. The set of components the component developer provides consists of variable and non variable components. Section 3.4.1 illustrates the development process of variable components in more detail.

3.5.1.4. System Architect

The system architect has to develop the software architecture that results from the DE development process. The system architect therefore, uses the components delivered by the component developer and connects them. In the next step the system architect connects variation points provided by the VPM to the components to make them configurable. If the variation point is compatible to the configuration interface of the component, as defined in *Definition 8* in Section 3.4.4.1, this can be done directly. If these entities are not compatible the system architect has to define a *VariationPointMapper* (see Section 3.4.4.2). Figure 3.30 illustrates the different activities of the system architect.

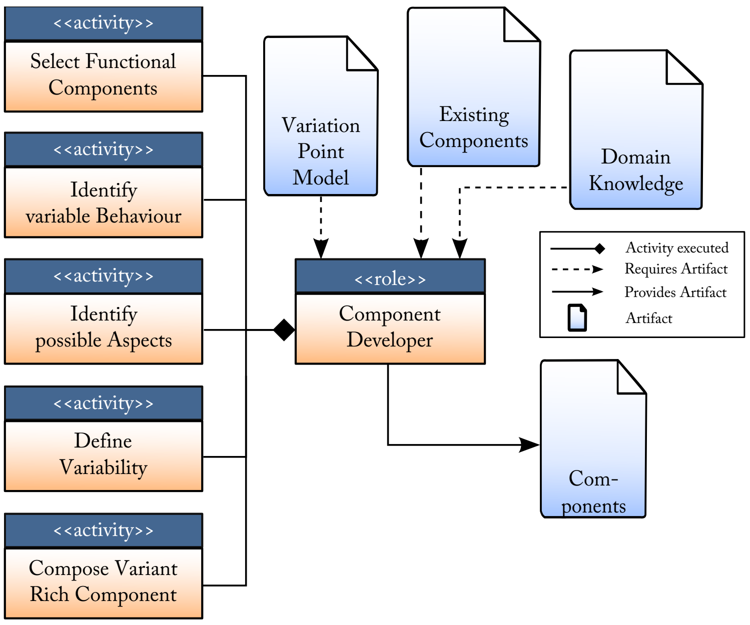


Figure 3.29.: The activities a component developer has to carry out to deliver a set of reusable components

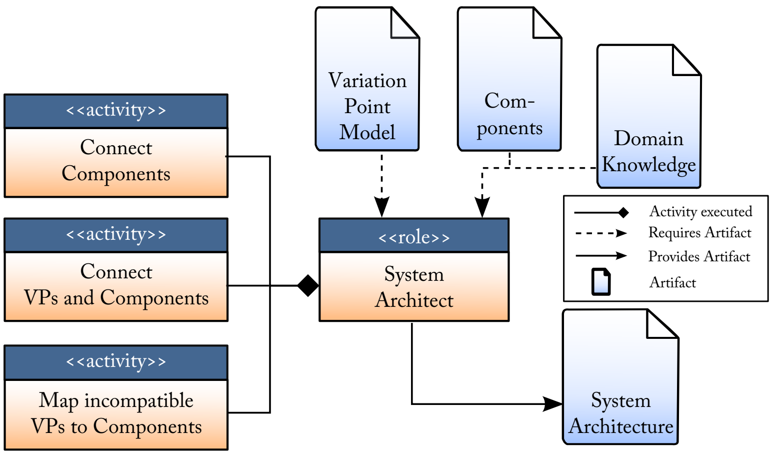


Figure 3.30.: The activities of the system architect to deliver the system architecture

The system architect delivers the system architecture that is used by the variant developer to derive concrete variants from it. If the variant developer detects errors in the system architecture the system architect is responsible to resolve them. Figure 3.31 illustrates the development process shown in this section.

3.5.1.5. Variant Developer

The variant developer is responsible for deriving concrete variants from the generic system architecture provided by the system architect. The main activity the variant developer has to perform is to bind variation points to concrete values and to select aspects to be

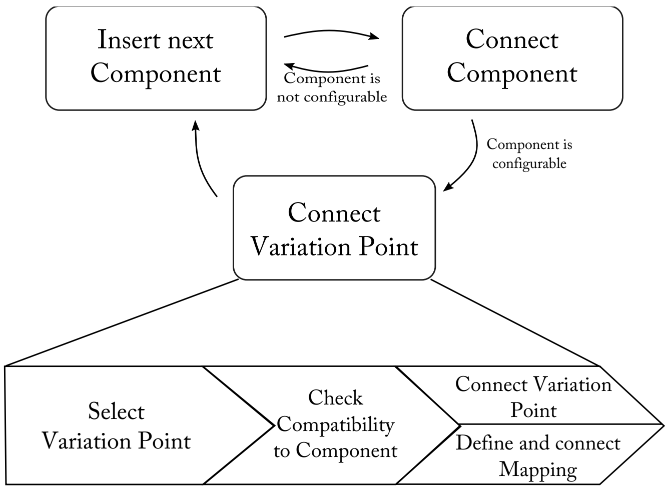

Insert next Component
Connect Component
Component is not configurable
Component is configurable
Connect Variation Point
Select Variation Point
Check Compatibility to Component
Connect Variation Point
Define and connect Mapping

Figure 3.31.: The development process that system architect applies to deliver the system architecture

active or inactive in the concrete variants. Furthermore, the variant developer has to resolve dependencies between different variation points. If behavior exists that is needed by just one variant the variant developer may need to implement it. In terms of software maintenance the variant developer is responsible to propagate changes in the generic system architecture to the different concrete variants. Figure 3.32 illustrates the activities the variant developer has to carry out.

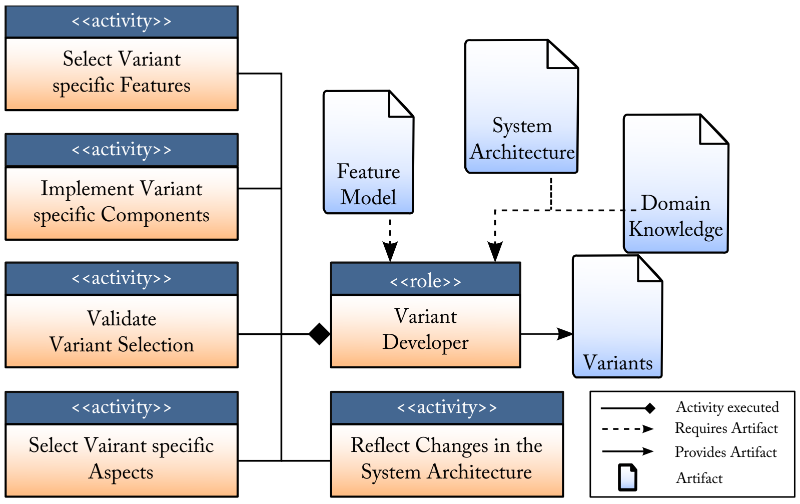


Figure 3.32.: The activities of the variant developer to deliver different variants based on the system architecture

To carry out these tasks the variant developer mainly uses the Feature Model and the system architecture. By selecting features the Variant Component Model can determine how to adjust variation points associated with them. Furthermore, the VPM provides the ability to resolve dependencies between variation points automatically. Possible errors in the system architecture need to be reported to the system architect who is responsible to resolve them.

3.5.2. Tools to support the Activities

Based on the different roles and activities associated to them tools are needed that help those roles to implement their tasks. There are five kinds of tools which are illustrated in Figure 3.33. This tools are needed in order to develop applications that use the SPLE processes described in Section 2.3

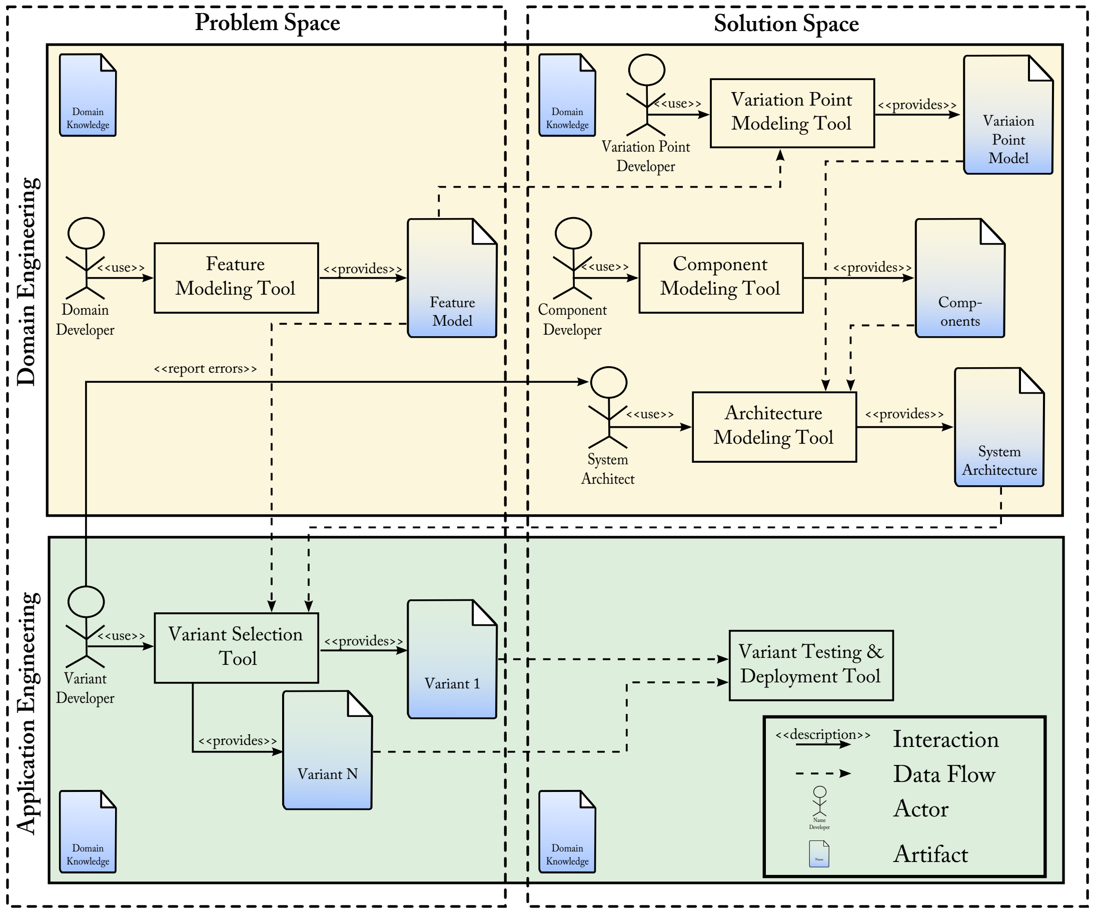


Figure 3.33.: The tools and their relationship needed to develop software according to Software Product Line Engineering

Feature Modeling Tool used by the domain developer to develop the Feature Model.

Variation Point Modeling Tool used by the variation point modeler to define variation points and to relate them to features provided by the Feature Model. Furthermore, this tool provides the ability to define aspects needed throughout the system architecture.

Component Modeling Tool is used by the component developer to model components and their interfaces. Furthermore, this tool provides the ability to define and implement variability of components where needed.

Architecture Modeling Tool is used by the system architect to define interactions between components by connecting them. Furthermore, this tool provides the ability to connect variation points to configuration interfaces provided by components.

Variant Selection Tool is used by the variant developer to derive concrete variants from the generic system architecture. It assembles the Feature Model and the system architecture and adjusts variation points based on the feature selection done by the variant developer.

Variant Testing and Deployment Tool provides the ability to test concrete variants and to deploy them to concrete devices, e.g. ECUs in the context of automotive software. The functionality of this tool is not covered by this theses but it completes the image of SPL based software development.

The tools are based on the different roles identified earlier. Each tool is associated to a specific role. The exception here is the *Variant Testing and Deployment* tool, which has no role assigned in the Variant Component Model. Each tool provides the ability to execute the tasks associated with the role that uses it.

The mentioned tools need not to be viewed separated from each other. There can be tools that unite functionality described in different tools. Simulink, for example, provides the ability to model components and architecture in one single tool. Figure 3.33 illustrates the relationship of these tools and how they are placed in the four quadrants of SPLE described in Section 2.3.2.3.

3.5.3. Deriving Variants

Section 2.2.4 shows possibilities, how variability can be brought to a model based environment. Based on the approach of using dedicated elements to model variability, this section demonstrates how variants can be derived from models implemented using the Variant Component Model. Basically there are two possibilities 1) to generate a new model or 2) to remove elements from an existing model.

To generate a new variant specific model, information needs to be preserved that is needed by the model's environment. Simulink, for example, stores various informations on how and where to display the blocks used within its MDL files. Therefore, a separate entity is needed to remember this information. To actually map components within the Variant Component Model to this information the `codeBehind` member can be used as described in Section 3.4. Figure 3.34 illustrates this approach.

When removing components from an existing model no such *Information Data Base* is required. On the other hand, a separate copy of the source model per possible variant is needed. Therefore, the problem of preserving information needed by the target environment to process the models is moved outside the variant derivation tool.

Which approach to use depends on the actual use case and tool support. For example, if the source model stores structure separated from layout the first approach can be preferred.

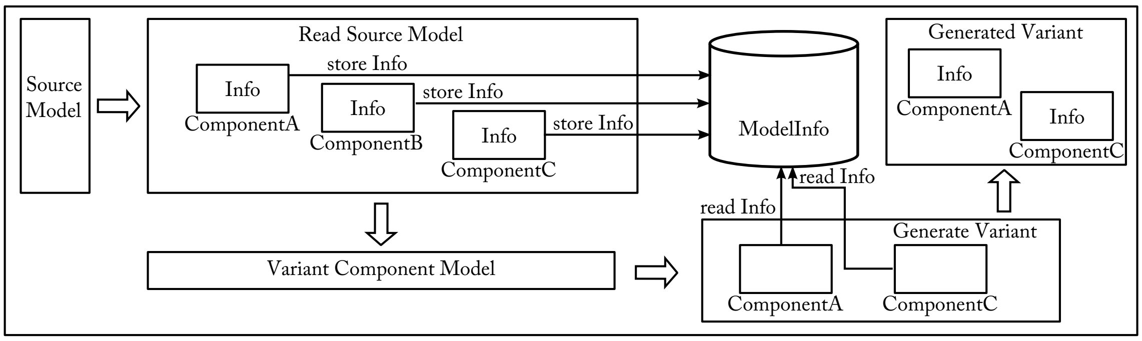


Figure 3.34.: Generating a new model per variant needs a central data base to store information specific to elements of a certain source model

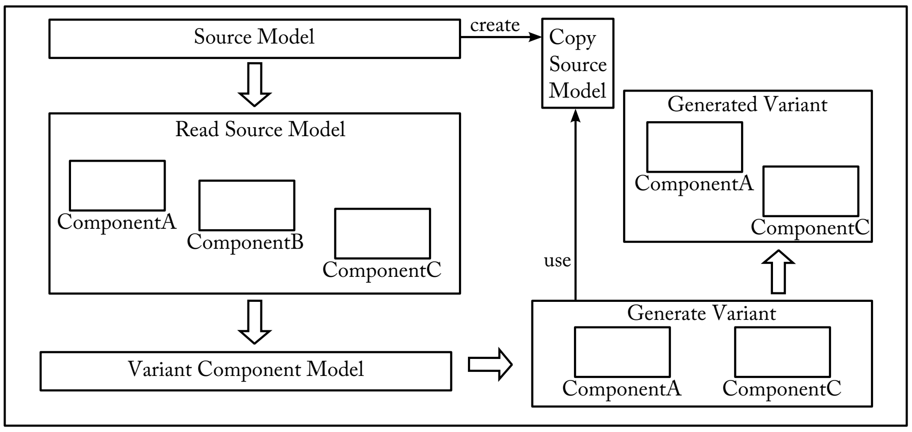


Figure 3.35.: Preserve model specific information by providing a copy of the source model which is adjusted when a variant is generated

If the sources models do not follow that scheme but can be copied easily the second approach can be taken into account.

For the implementation of the first approach a mechanism is required to store the required information per element, e.g. *Component*. The second approach requires a mechanism to automatically generate copies of the processed source models.

4. Implementation

4.1. General Overview

4.1.1. Tool Selection

In Chapter 3 an abstract tool chain is defined to support the Variant Component Model. For a concrete implementation these tools need to be mapped to either already existing tools or to tools that need to be developed. Figure 4.1 illustrates the tools selected to support the Variant Component Model in the implementation of this thesis.

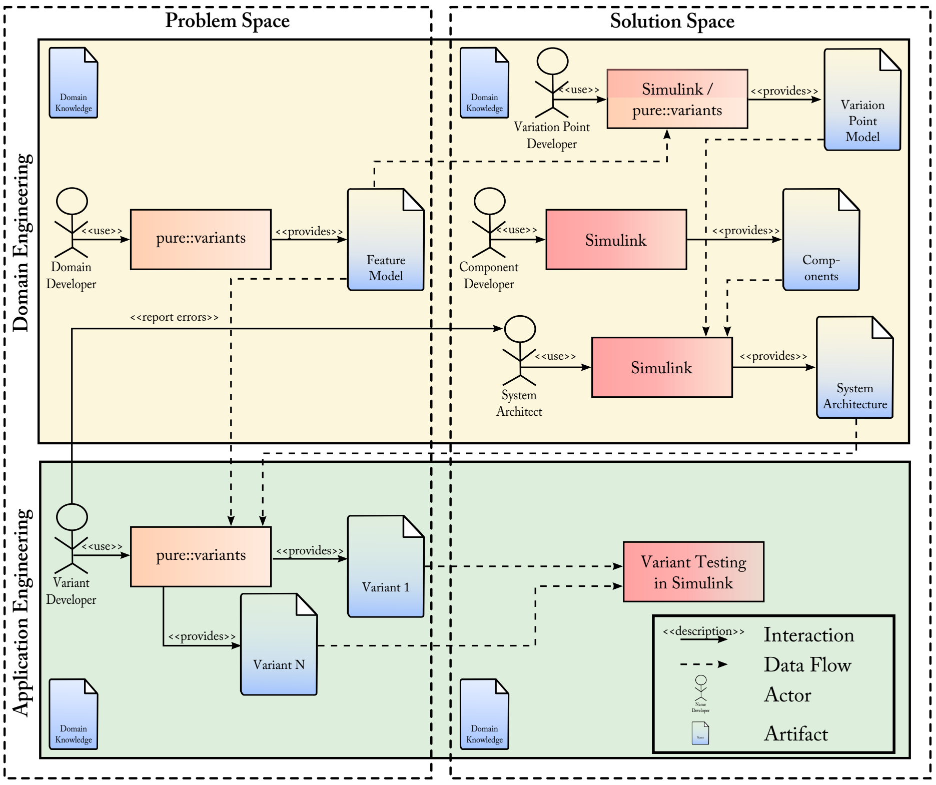


Figure 4.1.: The basic tool landscape consisting of pure::variants and Simulink selected for the implementation

4.1.1.1. Problem Space

As shown in Section 2.3.3.1, pure::variants provides a lot of functionality especially for SPLE. Furthermore, pure::variants is designed to be extensible to make use of existing tools. pure::variants Feature Models and Association Models provide functionality required by the Variation Point Model, defined in Section 3.3. Therefore, pure::variants was selected to be the tool for the problem space related roles, Domain Developer and Variant Developer, as defined in Section 3.5.1.

4.1.1.2. Solution Space

As already stated in Section 3.1.1, Simulink has a dominant role within development of automotive software. Therefore, Simulink is the central part when developing automotive software components that provide variability. The Variant Block Set, discussed in Section 2.3.3.2, gives a first starting point for modeling variability within Simulink. Therefore, Simulink was selected to be source for the implementation of the Variant Component Model. The roles of the Variation Point Developer, Component Developer and System Architect, as defined in Section 3.5.1, are located within Simulink.

4.1.2. Selected Approach

The approach used in this implementation is to convert the Simulink model into an instance of the Variant Component Model. The variation points, defined with means of the Variant Block Set are converted into an instance of the Variation Point Model, using a custom XML based exchange format. These models are then loaded into pure::variants where variants can be derived. Based on the selected configuration a Matlab script is generated that contains information about how the Simulink model needs to be transformed. This script has to be executed within Matlab to generate the desired variant.

The implementation of the Variant Component Model, as presented in Chapter 3, is done in Java independently from pure::variants and is organized into three parts:

Variant Component Model core This part provides central interfaces to decouple the generic models from concrete source models like Simulink. Furthermore, it provides the implementation of the variant building algorithm and the interface to the Family Model that is based on the Variation Point Model and Variant Component Model.

Variant Component Model core models This part provides the implementation for the generic models presented in Section 3.3 and Section 3.4.

Variant Component Model integration This part implements the interface between the component model core and Simulink – converting Simulink models into instances of the Variant Component Model.

This Micro Kernel architecture [Buschmann et al., 1996] makes it possible to plug-in different model sources without affecting parts that make use of them.

Simulink models are converted into an instance of the meta model provided by the Variant Component Model. Therefore, the variant generation process is not specific to Simulink models and can be used for other model based software development approaches as well. Details on these different parts are given in Section 4.3 and Section 4.4.

To overcome shortcomings of the Variant Block Set identified Section 2.3.3.3, some additional blocks and APIs are provided to support the roles related to component development. These parts of the implementation are shown in more detail in Section 4.2.

Last but not least Section 4.5 provides details on the implementation of the the Eclipse plug-in for pure::variants. The task of this plug-in is to provide a hierarchical view on variation points and to provide a graphical interface for importing Variant Component Models and exporting the desired variants into Matlab scripts.

4.2. Simulink API

To be able to model all scenarios defined in the motivation section of this thesis new blocks and guidelines are necessary. Some blocks are reused from the Variant Block Set and are particularly interpreted while loading Simulink models into a Variant Component Model instance. The following sections describe how the different scenarios can be achieved within the implementation provided by this thesis. The newly introduced blocks are combined in a library called *Variant Modeling*. This library provides an API to configure or manipulate these blocks.

4.2.1. Providing Configuration Information

During simulation of the entire system information about the currently active configuration is still provided by the *Variant Constant* block. The usage of this block has been described in Section 2.3.3.2.

4.2.2. Modeling Alternatives

To represent alternatives in Simulink the *Variant Multi Port Switch* block provided by the Variant Block Set has to be used. Alternatively, the *Variant Switch*, analyzed in Section 2.3.3.2 can be used as well. Note that this block can only serve two alternatives and can not be used with subsystems providing a composite signal. To facilitate post processing, some structural guidelines need to be fulfilled. The control input of each switch is required to be either directly connected to a *Variant Constant* block or to an input port of the surrounding subsystem. This way, the connected input port becomes part of the components configuration interface. The number of configuration ports between the switch and the *Variant Constant* block does not matter, there just need to be a path from the switch to the constant. The other structural requirement affects the alternatives themselves. All behavior related to a certain alternative must be modeled inside a single subsystem. On the one hand this reflects the design of variant components (see Section 3.4) and on the other hand makes it easer to identify parts belonging to a certain alternative. Figure 4.2 illustrates the structure required to model alternative implementations.

For automation and ease of usage the Variant Modeling Application Programing Interface (API) provides a custom block that can be used to model alternatives within Simulink. This block serves as a template which provides the basic structure of s subsystem that provides alternatives. This structure consists of a *Variant Multi Port Switch*, a configuration port connected to the control port of this switch and an output port connected to the output port of the switch. To configure the number of alternatives and to select a variation point used to control those alternatives a configuration dialog is provided. Furthermore, the number of input ports for all alternatives can be configured. Based on the number of values the selected variation point defined the configuration dialog generates

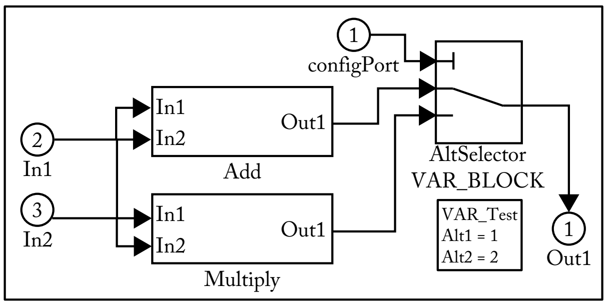


Figure 4.2.: Structural requirements to subsystems that provide alternative implementations

empty subsystems representing the different alternatives. Furthermore, input ports are generated and connected to each of the alternative subsystems, and each alternative sub system is connected to the already present variant switch.

Reconfiguration currently needs to be done manually since the provided configuration dialog is only intended for initial configuration.

4.2.3. Modeling Optional Components

Within the Variant Block Set optional components can be modeled by using *Variant Enabled Subsystem* blocks. For the proposed implementation these blocks are not enough, because they do not provide information about the variant that enables them. Therefore, optional subsystems are required to define three items:

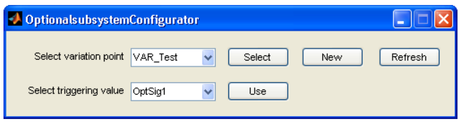
1. A mask parameter called `var_info` providing the name of the variation point that controls the optional subsystem (Note that all variability mechanism blocks defined by the Variant Block Set define this parameter).

2. A mask parameter called `triggeringValue` providing the name of the value that enables the optional subsystem. The used name must be defined by the variation point provided via the `var_info` mask parameter mentioned above.

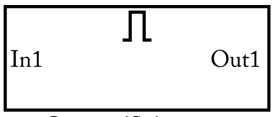
3. An *Enabled Port* used to simulate activation and deactivation of the subsystem. The instance of the *Variant Constant* block providing variability information is connected to this port.

Therefore, no special subsystem block needs to be used, and every subsystem block can be made optional by just following the three requirements enumerated above. According to these requirements the Variant Modeling API provides another custom block that predefines the required mask parameters and a basic structure consisting of the required `Enabled Port`, an `Inport` and an `Outport`. The last two ports are directly connected to each other.

Furthermore, a simple configuration dialog is provided that makes selection of variation points and triggering values for the desired subsystem easier. The provided block and its configuration dialog are shown in Figure 4.3.



(a) Using the configuration dialog



OptionalSubsystem

(b) to configure an optional subsystem

Parameter Name	Parameter Value
var_info	VAR_Test
trigger:	OptSig1

(c) and its required mask parameters

Figure 4.3.: Configuring an optional subsystem

4.2.4. Handling Variable Buses

Optional subsystems may not only be connected to ports, they may also be connected to buses. Buses are used to combine different signal sources into one composite signal. Simulink defines two blocks that interact with such buses: The `BusCreator` and the `BusSelector`. `BusCreators` are used to combine different signals into one composite signal and `BusSelectors` on the other hand are used to extract the single signals from the provided composite signal. If a certain signal source disappears in a certain variant, for example by removing an optional subsystem, buses routing those signals need to be adjusted.

Unlike subsystems, `BusCreators` and `BusSelectors` can not be masked in Simulink. The Variant Block Set also provides no block that addresses this problem. Therefore, the Variant Modeling API provides two variant adapters for buses. One adapter for `BusCreators`, called *VariantBusCreator* and one for `BusSelectors`, called *VariantBusSelector*. Both blocks are simply subsystems that provide four mask parameters:

`var_info` The name of the variation point that configures the block related to the bus.

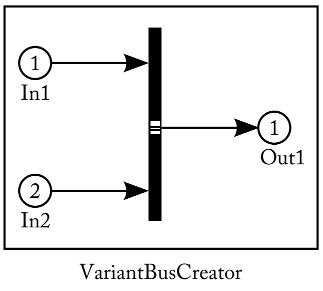
`fixedSignals` The number of signals that are fixed. These signals are part of any variant.

`optionalSignals` The number of optional signals. Each of this signals can be disabled in specific variants.

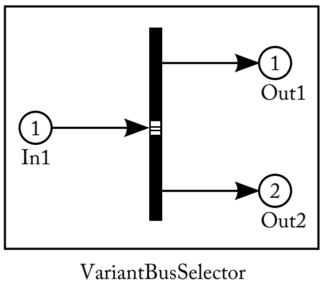
`triggerMappings` The list of mappings from variation point values to ports of optional signals. The list is formated using a colon separated list of assignments from signal ports to variation point values, e.g. `val1=in1;val2=in2`. Each optional signal has to be member of this list.

Since the *Variant Bus Creator* adopts the `Bus Creator` provided by Simulink is has a fixed output port that is connected to the output port of the wrapped `Bus Creator`. The number of input ports the *Variant Bus Creator* defines depends on the number of inputs the `Bus Creator` is ready to consume.

The *Variant Bus Selector* works similarly but has a fixed number of input port and a variable number of output ports, since the number of provided signals depends on the number of signals the underlying bus carries. Figures 4.4a and 4.4b show the structure of variant buses using two signals.



(a) Variant Bus Creator



(b) Variant Bus Selector

Figure 4.4.: Exemplary structure of the *Variant Bus Creator* and *Variant Bus Selector* blocks

Since configuring such buses, especially the required list of mappings, can be complicated, the Variant Modeling API provides a configuration dialog for this type of blocks. The configuration dialog is organized as shown in Figure 4.5a.

To configure a variant bus three steps need to be performed:

Select Variation Point Selecting a variation point affects the number of optional signals the variant bus provides. If no variation point is selected the variant bus acts like a default bus block as provided by Simulink. In case a new variation point is needed, the *Variation Point Explorer* provided by the Variant Block Set can be used.

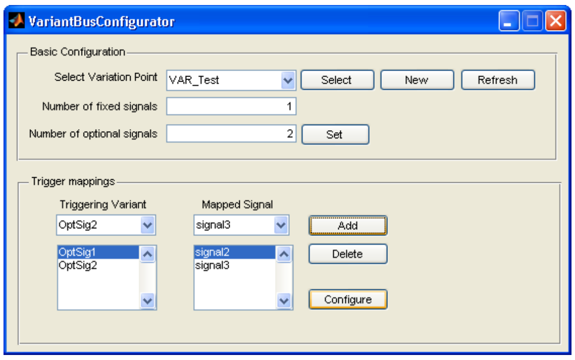
Provide number of fixed signals Fixed signals will never change and are present in all variants.

Define mapping of variation point values to optional signals By selecting items from the two provided drop down lists called `Variation Point Value` and `Optional Signal` mappings can be defined that are used by the variant bus block.

The number of signals provided by variant buses depends on the number of fixed signals and on the number of optional signals. The number of optional signals is determined by the number of values defined by the selected variation point.

The provided configuration dialog can be used for both, the *Variant Bus Creator* and the *Variant Bus Selector* blocks since both blocks need the same the configuration information. In fact, the major difference between those two blocks is the configuration of the wrapped blocks' type. Whereas the `Bus Creator` block just defines a parameter called `Inputs` which represents the number of signals it consumes, the `Bus Selector` defines a parameter called `OutputSignals` where the names of the signals are listed it provides.

Beside the configuration of the wrapped bus handling blocks and filling the required mask parameters as mentioned above, the configuration code generates the required input or output ports and connects them to the corresponding wrapped bus block. For example, if a *Variant Bus Creator* should have one fixed signal and the used variation point defines two values this *Variant Bus Creator* finally has three input ports all connected to the inner `BusCreator` This is illustrated in Figure 4.5. The same is done for *Variant Bus Selectors* just generating output ports and connecting them to the output ports of a *BusSelector*.


VariantBusConfigurator

Basic Configuration

Select Variation Point: VAR_Test

Number of fixed signals: 1

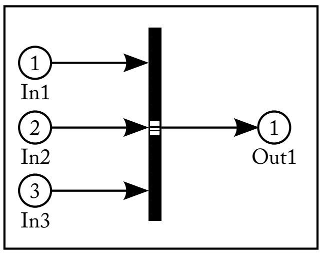
Number of optional signals: 2

Trigger mappings

Triggering Variant: OptSig2 (list: OptSig1, OptSig2)

Mapped Signal: signal3 (list: signal2, signal3)

(a) Using the configuration dialog



(b) to configure a variant bus (bus creator in this example)

Parameter Name	Parameter Value
var_info	VAR_Test
fixedSignals	1
optionalSignals	2
triggerMappings	OptSig1=In2;OptSig2=In3

(c) and its required mask parameters

Figure 4.5.: Configuring a *Variant Bus Creator*

4.2.5. Variant State Machines

Unfortunately, Simulink State Charts are not as flexible as Simulink subsystems. Neither states nor transitions can be annotated to store variability information. Transitions are the only place to store information about variability in a state chart by utilizing the conditions that can be defined for them. To implement variability in state charts the following basic setup needs to be performed:

1. Define a configuration port that is either directly connected to a variant constant block or to another input port that has a path to such a block.
2. Define transitions that do the variant selection. Each of these transitions has to define a condition comparing the actual value of the configuration port against the value that identifies the variant implemented by this transition or its subsequent state.

To clearly identify variant parts in a state machine the following structure is required:

1. All variable states or transition have to be grouped into a sub state having a name with the prefix `Var`.

2. This sub state has to have a junction point that collects all incoming transitions. Incoming transitions have to be free from any conditions and actions.

3. Variant selection is done by transitions subsequencing the mentioned junction point. The conditions of these transitions, as mentioned above, have to clearly define which variant they select. If transitions need to implement behavior, this can be implemented within their actions.

4. States that may be selected by different variants have no restrictions, except their transitions. They may only have one incoming and one outgoing transition.

5. If a transition represents a variant on its own it has to leave the variant sub state immediately.

4.2.5.1. Example on State Machine Variability

This short example shows how variability inside Simulink State Charts has to be defined. The state machine receives an input parameter, called `baseValue` from the surrounding Simulink model and defines two variants:

First Variant provides the result of adding 1 to the input value.

Second Variant provides a result equal to the input value.

The variants themselves are configured via the `configPoint` input port to the state chart. This example is only used to illustrate the required structure so no further actions are done.

Figure 4.6 shows the resulting state machine that defines the required variability.

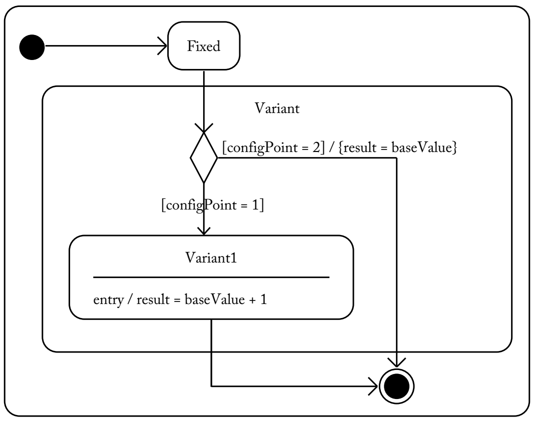

Fixed
Variant
[configPoint = 2] / {result = baseValue}
[configPoint = 1]
Variant1
entry / result = baseValue + 1

Figure 4.6.: An example how variability inside state machines has to be modeled

4.2.6. Additional Functions Provided by the Variant Modeling API

4.2.6.1. Exchanging Variation Points

Variation points can only be stored using Simulink specific, binary based Data Dictionary (DD) files. Because of their binary nature these DD files can not be used by the implementation since they can not be parsed without knowing the underlying structures. This problem was solved by defining a simple XML based exchange format for variation points which is described in more detail in Section 4.4.2. To provide a confident way to export variation points defined within Matlab the *Variant Modeling API* provides a function called `storeVariationPointsToFile`. This function takes a single argument representing the name of the file where the variation points should be stored. The function itself collects all variation points currently defined within the Matlab workspace utilizing the `var_get` function provided by the Variant Block Set. After collecting the variation points they are converted by using Matlabs built in *XML* creation functions.

4.2.6.2. Disabling Bus Signals

Since *Variant Buses* are designed to adopt themselves to different variants it must be possible to adjust them some how. The *Variant Modeling API* provides a function called `disableBusSignal` that is capable to execute this task. This function takes two arguments to identify 1) the *Variant Bus* it self and 2) the index of the signal that should be removed. Since Simulink uses paths rooted at the first level model to identify elements within the model this path is used to identify the *Variant Bus*. The function determines on its own if the provided path to the *Variant Bus* leads to a Bus Creator or Selector, and therefore, can be used for both types of buses. Figure 4.7 illustrates the result of an invocation of this function.

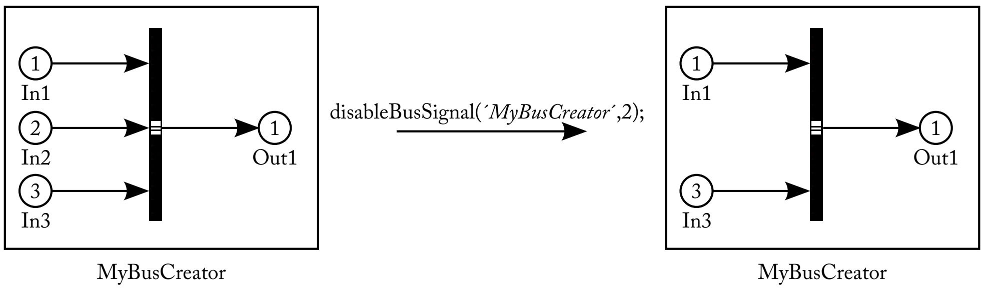


Figure 4.7.: Removing the second signal from a *Variant Bus Creator* by invoking `disableBusSignal`

4.2.6.3. Adjusting State Charts

There are two different APIs provided by Simulink. The one used in this thesis to manipulate Simulink models and another, more "object oriented" API, which has to be used to manipulate state charts. Because this API involves more steps in order to get things done the *Variant Modeling API* provides some functions to automate certain steps. For example, before being able to remove a state 1) its parent state chart must be found and 2) the state itself needs to be found based on its id. To finally remove the state its "destructor" function needs to be called.

All these steps are done in one function call to *deleteStateFromStateMachine* provided by the *Variant Modeling API*. Appendix A provides details for all functions provided by the *Variant Modeling API* to manipulate state charts.

4.3. Variant Component Model Core Implementation

Beside the implementation of the meta models the core of the Variant Component Model implementation consists of two major parts:

Model Providers are responsible to map concrete model sources to the representation defined by the used meta models.

Variant Family Model stores the instances of the Variation Point Model and Variant Component Model and provides access to them. Further it provides the operations to start the variant derivation process.

4.3.1. Abstracting Models from their Sources

To decouple the loading of models from their representation the core implementation provides two basic interfaces called *VariationPointModelProvider* and *ComponentModelProvider*. Implementations of these interfaces are responsible to convert the representation of their specific sources into the respective models. Therefore, *VariationPointModelProvider* implementations have to provide a Variation Point Model and *ComponentModelProvider* implementations have to provide the root component of the imported model.

Model Providers can be created through a factory class called *ModelProviderFactory*. Following the Abstract Factory pattern described in [Gamma et al., 1995] this class provides creation methods for both model providers. To make model providers accessible to the factory they need to be registered. To do this an identifier is required that is used to lookup the provider while creating it.

4.3.2. Variant Family Model

Based on the term used within pure::variants the Variant Component Model provides a Family Model that consists of the Variation Point Model and a variant component acting as root of the used system. The *Variant Family Model*, represented by an equally named class, provides operations for loading models and generating variants from these models. Furthermore, the *Variant Family Model* provides shell methods to bind or unbind variation points in the encapsulated VPM. Table 4.1 lists all operations supported by the *Variant Family Model* along with a short description of each.

To be able to load models from different sources, for example Simulink, the Variant Family Model utilizes the different model providers by requiring an instance of each.

4.3.3. Generating Variants

Generation of specific variants is based on the current configuration selected for the variation points. The variant generation it self is based on three basic steps:

1. Identify variable components.
2. Bind variable components.
3. Generate the variant.

Operation	Operation Description
Bind variation point	Binds a variation point in the stored VPM to a specific value. The variation point and the selected value can be provided via their concrete instances or via their names.
Unbind variation point	Clears the selected values in the specified variation point.
Load models	Loads the models specified by invoking the specified model providers.
Generate variant	Based on the current configuration a variant generated and stored in the provided variant generation script instance.
Get prepared script	Since Component Model Providers can generate a variant generation script too this operation provides access too it.

Table 4.1.: Operation supported by the Variant Family Model

The first two steps are executed recursively on the present component model. This means before reducing a variable component the selected variant, its variable subcomponents are reduced to this variant. Figure 4.8 illustrates this.

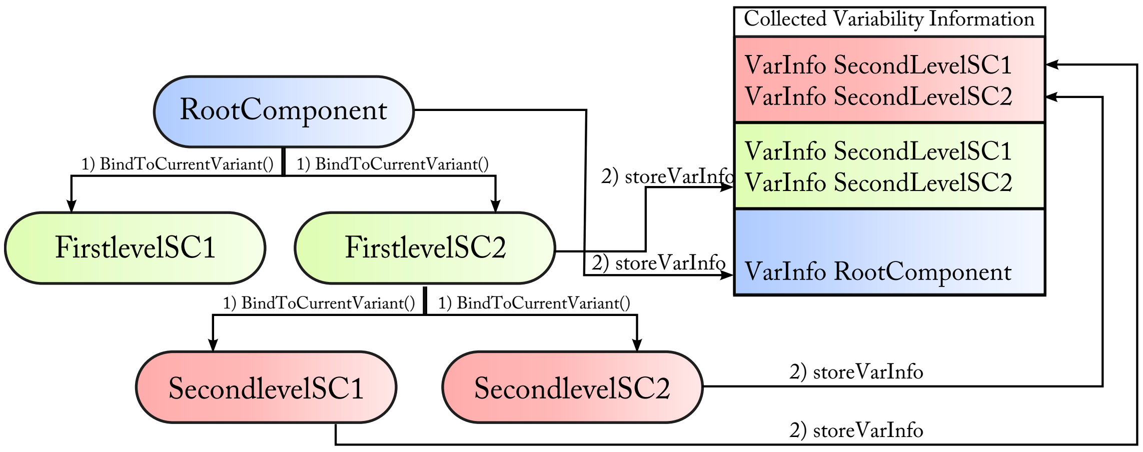


Figure 4.8.: Binding a system with variable components to a specific variant

4.3.3.1. Binding a Component

Binding a component means to remove all parts from it that do not belong to the desired variant. In the Variant Component Model this is done by using the connectors the component's *Variability Realizer* defines. Since each connector defines a callback for handling each of its selection states the variant derivation algorithm just needs to call the appropriate callback method based on this selection state. Therefore, if the connector is exported its `onIncluded` method is called, its `onRemoved` method is called otherwise. This behavior is implemented within the *bindToCurrentConfiguration* method each variant component defines.

Removing Subcomponents Because the implemented approach removes components that are not needed in certain variants, their connections to other components need special attention. Subcomponents are always connected to other components. If a certain component is removed in a particular variant its connections to other components or to its parent component needs to be removed too. Removing connections may imply changes in the parents components interface. This is the case if the port of the parent component only connects to the removed subcomponent. In this case the port can be removed in the specific variant.

Adding new Connections During variant generation it may happen, that port links need to be added that were not part of the source model. An example for this case is given in Section 4.4.1.2. To make this generally possible port links can be marked as *virtual*. Virtual port links indicate that they are currently not present in the source model and need to be added when generating a variant from it.

4.3.3.2. Storing Variant Configurations

All that is needed is a place to store the information the different connectors and their callback methods provide. That's where the *VariantGenerationScript* and *VariantGenerationScriptOperation* classes come into play. *Variant Generation Scripts* are responsible to store instructions that should be executed against the model in order to generate the specific variant. These instructions are implementations of the *VariantGenerationScriptOperation* interface. To automate the instantiation of these instructions, the class implementing the *Variant Generation Script* provides builder methods to create and store instances of the desired instructions. Table 4.2 lists all operations supported by the *Variant Generation Script*.

Operation	**Operation Description**
Add line to model	Adds a port link between two components.
Disable bus signal	Disables a signal identified by its index in a bus component.
Remove component	Removes a certain component from the model.
Remove port	Removes a port from a certain component.
Remove port link	Removes the connection between two components.
Rewrite transition	Rewrites a transition between two states. This can be a change of the source and / or target states and / or a change in the label of the transition.

Table 4.2.: Operations supported by the Variant Generation Script

An instance of this class is used to store the information about items that should be removed or added in a specific variant. Component Model Providers also provide such scripts in order to inform the variant derivation process about items that have been removed from the model when reading it. Such elements need to be removed from the target model in case the variant is not newly created but manipulates an existing model, as done in this implementation.

Since *Variant Generation Scripts* just store information about actions that need to be done, an environment is needed that actually executes these actions. Since these environments differ between model sources an abstraction mechanism is needed. This is done by the *VariantScriptWriter*. A *Variant Script Writer* supports the same set of operations as the *Variant Generation Script*. A *Variant Generation Script* is executed against a specific *Script Writer*. To find the right operation to call inside the *Script Writer* a visitor mechanism, as defined in [Gamma et al., 1995], is used. The listings provided in Listing 4.1 and Listing 4.2 illustrate this mechanism.

```
public void execute(VariantCreator scriptWriter)
{
  for (VariantGenerationScriptOperation current : this.scriptOperations)
    current.acceptCodeGenerator(scriptWriter);
}
```

Listing 4.1: The `execute` method of the `VariantGenerationScript` invokes the different operations by calling their `acceptCodeGenerator` method

```
public void acceptCodeGenerator(VariantCreator generator)
{
  generator.removeComponent(this.toRemove, this.parentOfRemovedComponent);
}
```

Listing 4.2: The invoked operation, in this case `RemoveComponent`, calls the appropriate method on the provided variant generator

All operations listed in Table 4.2 available for *Variant Generation Scripts* are visitable and accept a *Variant Script Writer* as argument of their `acceptCodeGenerator` method. Therefore, the operations themselves can decide which method to call on the *Script Writer* instance.

4.3.4. Implementing the Models

The needed stubs for the classes and methods of the VPM and Variant Component Model were generated from the Unified Modeling Language (UML) models provided in Section 3.3 and Section 3.4.

The implementation of the stubs was directly derived from the rules defined in the mentioned sections but was not generated since the tool used for modeling, BOUML[1], does not support generation of code from UML activity diagrams. To make instantiation of members of the different models easier, item factories [Gamma et al., 1995] for both models are provided:

VariationPointModelItemFactory A class providing methods to create instances of model elements, e.g. *VariationPoint*s, of the Variation Point Model.

ComponentModelItemFactory A class providing method to create instance of model elements, e.g. `Components`, of the Variant Component Model.

[1] `http://bouml.free.fr/`

Both factories are implemented using static methods which assign the provided parameters to the appropriate fields and return the newly created instance of the desired model element. Listing 4.3 provides an example on how such factory methods are implemented.

```
public static FunctionalComponent createFunctionalComponent(
  String componentName, Port... ports) throws InvalidOperationException
{
  FunctionalComponent result = new FunctionalComponent();
  result.setName(componentName);

  for (Port current : ports)
    result.addPort(current);
  return (result);
}
```

Listing 4.3: A factory method that creates a new instance of a `FunctionalComponent` that defines the specified ports

4.4. Variant Component Model Simulink Interface

4.4.1. Simulink Model Provider

The task of the Simulink Model Provider is to read Simulink models represented as text based MDL files and to convert them into an instance of the Variant Component Model. The actual parsing of MDL files is handed over to a library provided by the ConQAT[2] project. This library reads MDL files and converts them into a Java object representation that directly maps the Simulink Object Model to Java objects.

Therefore, the Simulink model provider needs to convert the object model provided by this library into an instance of the Variant Component Model.

4.4.1.1. Mapping Simulink Blocks to Components

In the provided implementation the set of blocks Simulink offers is divided into two major groups: 1) Blocks that carry relevant information 2) Block with no special information. Relevant information in the context of the Variant Component Model is information that can be used to either form a new component or that represents variability information of a certain component. Table 4.3 lists all blocks that are mapped to concepts of the Variant Component Model when reading the Simulink model.

Beside these blocks only the *Variant Constant* block located within the Variant Block Set gets special treatment. If the binding time of the associated variation point is set to `PreBuild`, instances of this block are always removed from the model.

All other blocks are treated uniformly by converting them into so called *PlaceHolder-Components*. *PlaceHolderComponents* preserve the blocks interface but give no special meaning to their type. To get a clue on the real type of the converted Simulink block *PlaceHolderComponents* store the name of the block type. Figure 4.9 illustrates how an instance of a *PlaceHolderComponent* looks like when used to represent a Simulink `Mux` block.

[2] `http://conqat.in.tum.de/index.php/ConQAT`

Simulink Block	Mapped Concept
Subsystem	Composite Component
Variant Switch	Variability Realizer
Variant Multi Port Switch	Variability Realizer
Variant Bus Creator	BusComponent
Variant Bus Selector	BusComponent
InPort	Port
OutPort	Port
Line	Port Link

Table 4.3.: Simulink blocks that have corresponding concepts within the Variant Component Model

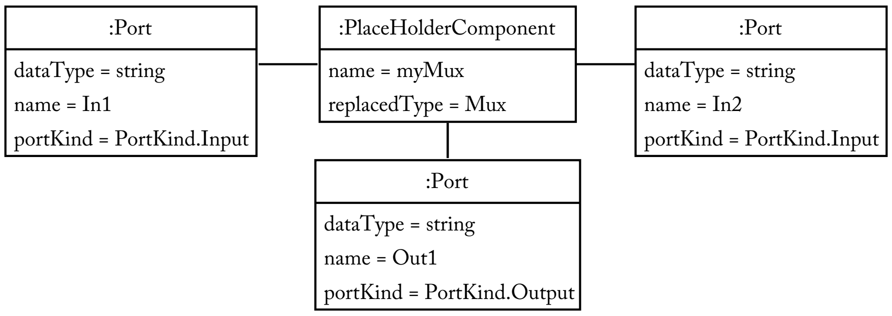


Figure 4.9.: The representation of the Simulink block `Mux` in the Variant Component Model

To abstract away logic for processing different Simulink blocks from logic determining the type of the Simulink block a Builder [Gamma et al., 1995] approach is used. For each Simulink bock listed in Table 4.3 a separate handler is implemented. All of these handlers implement the common `SimulinkBlockHandler` interface as shown in Listing 4.4.

```
public interface SimulinkBlockHandler
{
  public void handleBlock(SimulinkModelProvider callingProvider,
    CompositeComponent parent,
    SimulinkBlock blockToHandle)
      throws SimulinkModelException;
}
```

Listing 4.4: The `SimulinkBlockHandler` interface

The `handleBlock` method of this interface defines the following parameters:

`parentComponent` The parent component of the block to convert. A missing parent, i.e. the parameter is set to `null`, indicates that the block to handle is placed inside the root component, i.e. in Simulink terms the model, of the system,

`callingProvider` The *Simulink Model Provider* that has invoked the handler. Since

Simulink models are build recursively, (using blocks that store other blocks) this can be used to invoke the appropriate handlers of such nested blocks.

`blockToHandle` The actual Simulink block object, provided by the used ConQAT library, that should be converted.

Since Simulink stores all non built-in blocks as references to their respective libraries they are located in, instances of ConQATs `SimulinkBlock` class provide a fully qualified block name. This fully qualified name formated as `library.BlockType` is used to invoke the appropriate handler. This secondary lookup is implemented within a separate Simulink block handler called `ReferenceHandler`. It obtains the referenced type and forwards this type along with the provided `SimulinkBlock` instance and the parent component back to the Simulink Model Provider that invoked the reference handler.

The `SimulinkModelProvider` itself just stores a mapping of block types to their appropriate handlers. Based on the types of nested blocks the `SimulinkModelProvider` selects the appropriate handler. If no handler was found for the give type a default block handler invokes the generation of *PlaceHolderComponent*s as described earlier. Using this scheme handlers only need to be registered for blocks that need special treatment. When reading a Simulink model, the *Simulink Model Provider* instantiates a new composite component, names it according to the name of the provided model and invokes the process of reading this component. Therefore, Simulink models are treated like ordinary subsystems – as composite components.

4.4.1.2. Reading Components

As mentioned in the previous section, Simulink subsystems are treated as composite components. Subsystems may define variability by either using variant switches or nested optional subsystems. The process of reading such components is done in three steps:

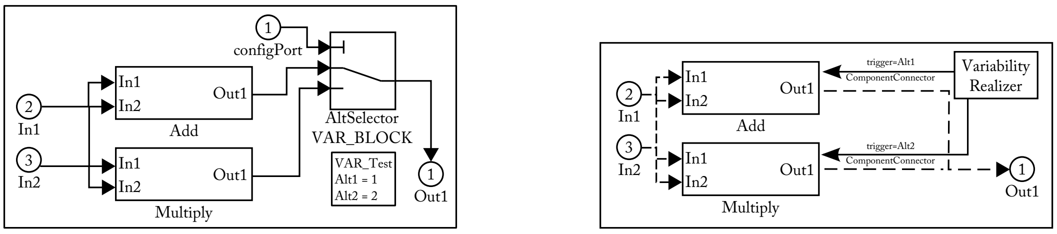
1. Read the components interface.
2. Read child components.
3. Read connections to surrounding components.

The component interface is derived from the name of the subsystem and the In- and Outports it defines. Child components are handed back to the *Simulink Model Provider* to decide how to handle these blocks. Reading connections to other components is a bit more complex. Since the ConQAT library exactly maps the representation of Simulink lines to Java `SimulinkLine` objects follow the scheme `SourceBlock/PortIndex -> TargetBlock/PortIndex`. Since *PortLinks* in the Variant Component Model reference ports defined by the appropriate components, these ports need to be looked up. Because not all subcomponents can be read at once it may happen that referenced subcomponents, including their ports, are not converted yet. Therefore, each subcomponent stores information about connections referencing it in the model provider. This information is used by the handler of the parent component to establish all required port links after reading all of its subcomponents.

4.4.1.3. Handling Variability

The Variant Component Model requires *Variability Realizers* that store connectors to parts of the component that depend on certain variation points. Therefore, the blocks that define variability within the Simulink models need to be mapped to this concept. The general process is to determine which variation point configures the component and which value of this variation point triggers a certain sub component. There are six blocks that may provide variability: *Variant Switch, Variant Multiport Switch, Optional Subsystem, Variant Bus Creator* and *Variant Bus Selector*. Except the two switches all blocks are provided by the Variant Modeling Library. All these blocks define a mask parameter called `var_info`. This parameter stores the name of the variation point that provides the configuration information. Since all `ComponentModelProviders` have access to the used VPM this information can be used to retrieve the actual variation point. By adding all values this variation point defines, the *Variability Realizer* of the component in question can be configured.

Mapping Variant Switches Both kinds of variant switches can be handled similarly. The difference between these two is just the count of alternatives that can be modeled with these blocks. The *Variant Switch* only supports two data inputs whereas the *Variant Multiport Switch* supports any number of data inputs and, therefore, any number of alternatives. The triggering variation point value is determined by the index of each data line. If the variation point used is called `myVariationPoint` the expression `myVariationPoint.getPossibleValues().get(indexOfDataLine)` provides the required variation point value. This mechanism was used because variant switches provide no ability to map signal lines to proper variation point values within Simulink. Therefore, the ordering of alternatives has an direct impact on variant selection of the given component. Figure 4.10 illustrates this a mapping process.



(a) Mapping from a Simulink variant switch

(b) to a Variant Component Model *Variability Realizer*

Figure 4.10.: Mapping a variant switch to a *Variability Realizer*

Mapping Optional Subsystems Beside the `var_info` parameter optional subsystems also have to define a parameter called `trigger` as mentioned in Section 4.2. Therefore, the required trigger for the component connector to this optional component can be uniquely identified. When identifying an optional component by its required structure, a component connector is created that activates this component with the identified trigger. This connector is then added to the *Variability Realizer* of the parent component.

Mapping Variant Bus Creators and Variant Bus Selectors From the component models perspective *Variant Bus Creators* and *Variant Bus Selectors* can be treated similarly. They are both buses either providing or consuming a variable number of signals. Therefore, they are both modeled using a special component called `BusComponent` which is a specialization of a `FunctionalComponent` provided by the Variant Component Model. The structure of the *Bus Component* is illustrated in Figure 4.11.

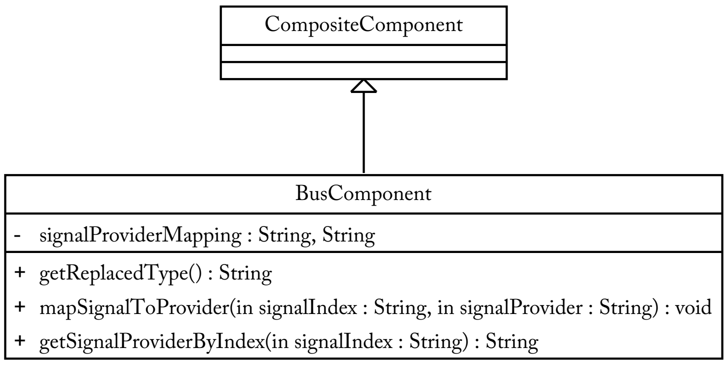
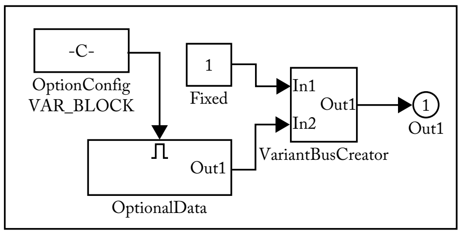
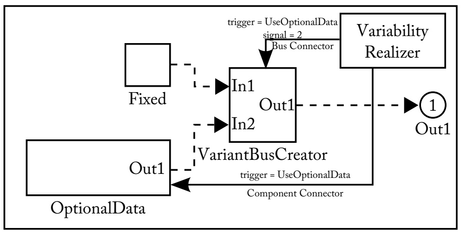

CompositeComponent
BusComponent
- signalProviderMapping : String, String
+ getReplacedType() : String
+ mapSignalToProvider(in signalIndex : String, in signalProvider : String) : void
+ getSignalProviderByIndex(in signalIndex : String) : String

Figure 4.11.: The structure of `BusComponent`

Bus Components provide a mapping from indices of signals to variation point value names in their `triggerMapping` parameter. This information can be directly used to generate connectors to this *Bus Component*. To reduce complexity an additional connector for *Bus Components* is implemented too. This connector, called `BusConnector`, stores the index of the signal that should be disabled in case the corresponding variant was not selected. Therefore, a *Bus Connector* generates a `DisableBusSignal` operation parametrized with the stored signal identifier when its `onRemoved` method is invoked. Figure 4.12 illustrates the mapping process of models containing buses and optional subsystems.



(a) The Simulink model to map



(b) The resulting Variant Component Model *CompositeComponent*

Figure 4.12.: Mapping a Simulink model containing a Bus (and optional subsystems) to a Variant Component Model *Composite Component*

4.4.2. XML Variation Point Provider

As shown in Section 4.2.6.1, variation points from Simulink are exported in an XML based format. On the Java side the task is to read these files and convert their content into an instance of the Variation Point Model.

4.4.2.1. The Used Exchange Format

The format used for exchange of variation points is based on an XML Schema. The schema was designed using a visual designer provided as an plug-in for Eclispe. All elements of this schema are located inside the `tns` XML namespace. The structure used in the XML Schema directly reflects the structure of Variation Point Model *Variation Points*.

Inside the root element, called `VariationPoints`, elements of type `VariationPoint` can be defined. Table 4.4 lists the required content of such elements.

Element Name	Description	Type	Possible Values
name	The name of the variation point	xsd:string	Non empty
enabled	The enabled state of the variation point	xsd:boolean	true | false
BindingTime	The latest possible binding time of the variation point	xsd:string	PreBuild | Post-Build
Values	The collection of value this variation point defines	tns:VariationPointValue	At least one value needs to be defined

Table 4.4.: The structure of files to exchange variation point information

The values a variation point defines need to be specified using the `value` element. The required contents of such elements are shown in Table 4.5. `value` elements do not define any child elements only attributes are used.

Attribute Name	Description	Type	Possible Values
name	The name of the variation point value	xsd:string	Non empty
value	The actual value the variation point value stores	xsd:string	Non empty
enabled	Indicates whether or not this value is accessible	xsd:boolean	true | false, defaults to true

Table 4.5.: The structure of variation point values in the used exchange format

Listing 4.5 shows an example that uses the specified format to define variation points. Note that namespace declarations are removed from the example.

```
<?xml version="1.0" encoding="utf-8"?>
<tns:VariationPoints>
  <tns:VariationPoint>
    <tns:name>VAR_Test</tns:name>
      <tns:enabled>true</tns:enabled>
      <tns:BindingTime>PreBuild</tns:BindingTime>
      <tns:Values>
        <tns:Value name="Variant1" value="1"/>
        <tns:Value name="Variant2" value="2"/>
    </tns:Values>
  </tns:VariationPoint>
</tns:VariationPoints>
```

Listing 4.5: Exemplary content of a variation point descriptions file

4.4.2.2. Parsing Variation Point Descriptions

Since an XML Schema was used to design the used file format it was possible to generate the parser for this format. Java has built-in support for XML via its Java API for XML Binding (JAXB) package. To generate the actual parser a tool provided by this package needs to be invoked. The *EclipseLink*[3] Eclipse plug-in provides a confident graphical interface for invoking the code generator.

The code generator itself generates a number of classes, one for each element defined in the schema, and the actual parser for the XML files. Parsing of files can then be done by invoking the generated `Unmarshaller` class as shown in Listing 4.6.

```
@Override
public VariationPointModel loadVariationPointModel()
  throws ModelProviderException
{
  try
  {
    JAXBContext myContext = JAXBContext.newInstance(
      "org.iti.vcm.core.vpm.xml");
    Unmarshaller variationPointReader =
      myContext.createUnmarshaller();
    VariationPoints readVarPoints = (VariationPoints)
      variationPointReader.unmarshal(new File(this.xmlFileName));
    // generate the variation point model
  }
  catch (JAXBException exc)
  {
    throw new ModelProviderException(
      "Could␣not␣read␣XML␣data␣for␣variation␣points", exc);
  }
  // catch exceptions occurring during model generation
}
```

Listing 4.6: Using the Unmarshaller to parse an XML file into a Java object model

The `Unmarshaller` returns an instance of the class representing the root element. After parsing the XML file the objects generated by the parser need to be converted into objects required by the VPM. JAXB would provide a mechanism to automate this, but the used model is small enough to convert it manually by simply looping through the elements generated by the parser and instantiating the corresponding model elements with the provided data.

[3] `http://www.eclipse.org/eclipselink/`

4.5. pure::variants Integration

Extensions to pure::variants can be developed using techniques provided by the Eclipse plug-in infrastructure. To let custom models interact with pure::variants models so called *External Models* are provided. The necessary classes and interfaces are provided via pure::variants' core plug-in. pure::variants can use these External Models to display them in a hierarchical structure, known from Feature Models, and to use them as source for Variant Description Models.

Since Simulink models can not be edited directly only the Variation Point Model is loaded into pure::variants for display.

4.5.1. Importing the VPM into pure::variants

4.5.1.1. Mapping to an External Model

Since both models, the Variation Point Model and the External Models are structured hierarchically, both models can be mapped directly. The API provided for External Models provide different elements for different depths inside the model. Table 4.6 lists the elements used for the VPM import.

Model Element	**Used for**
External Element	Variation Points and Variation Point Values
External Property	All members defined for Variation Points and Variation Point Values, e.g. their names
External Constant	For the actual values the different properties store.

Table 4.6.: Elements used from the External Model to import the VPM

Each External Model has to have a root element specifying the type of the model. Possible types for this root elements are limited to `Family Model (CCFM)` and `Feature Model (CFM)`. For the Variation Point Model the decision was made to represent it as a Family Model because variation points are technical realizations of features. Because not all models get imported into a pure::variants representation the root element defines two properties called `simulinkFile` and `variationPointsFile` which hold the file names of the Simulink model and the Variation Point Model. The latter is needed to implement synchronization.

Below the root element the variation points are located. Each variation point is converted into an External Element defining properties for its binding time and enabled state. Since all elements within pure::variants have to have a name, the one defined for variation points can be used here. External Elements representing variation points define child elements representing variation point values. These elements again define properties for each of the members defined for variation point values.

The import process itself is started by loading the provided XML file using the `XmlVariationPointProvider` class shown in Section 4.4.2. The Variation Point Model provided is then converted into an External Model. The final import into pure::variants is done using the `ModelGenerator` provided by the pure::variants SDK. The Model Generator reads the External Model and transforms it into a real pure::variants model and adds this model to the selected project.

4.5.1.2. User Interface for Import

The user interface to start the import of a VPM is implemented using a custom wizard. The wizard is plugged into Eclipse using the `com.ps.consul.eclipse.ui.pvimport.-VariantImportWizards` extension point provided by pure::variants. The real user interface has to be developed in form of classes that are derived from `WizardPage`. *Wizard Pages* can define their user interface as needed using elements provided by the Eclipse SWT framework but always have the required wizard buttons (Next, Back, Finish) and a field to display information about the currently active page. The Wizard Page developed for importing variation points requests the following pieces of information:

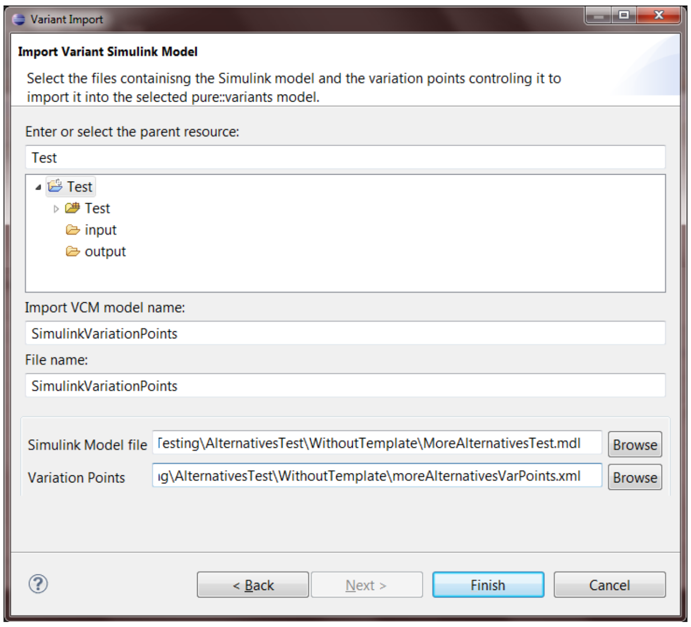
Target model name is the name of the model that should be generated inside pure::variants, thus the model displayed in the project tree.

Target model file is the name of the actual file where the target model should be stored.

Source Simulink model is the file that stores the Simulink model processed by this implementation.

Source Variation Point Model is the XML file containing the variation points.

Figure 4.13 demonstrates the look and feel of the developed import wizard.


Variant Import

Import Variant Simulink Model

Select the files containisng the Simulink model and the variation points controling it to import it into the selected pure::variants model.

Enter or select the parent resource:

Test

- Test
  - Test
  - input
  - output

Import VCM model name:

SimulinkVariationPoints

File name:

SimulinkVariationPoints

Simulink Model file: Testing\AlternativesTest\WithoutTemplate\MoreAlternativesTest.mdl

Variation Points: ıg\AlternativesTest\WithoutTemplate\moreAlternativesVarPoints.xml

Figure 4.13.: The wizard page requesting the needed information from the user in order to import variation points into pure::variants

After importing the required models, pure::variants presents the view on the newly created Family Model shown in Figure 4.14.

Currently it is not possible to change variation points. Though it is possible to change their contents inside pure::variants, changes will not affect the selected sources. Therefore, the usage of the imported VPM is limited to include them in Configuration Spaces to derive concrete variants.

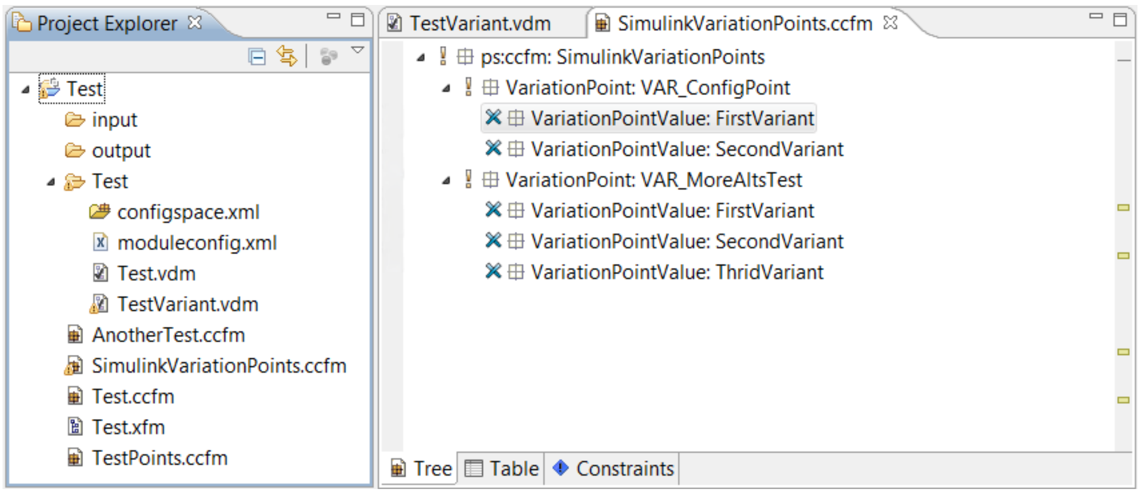

Project Explorer
Test
input
output
Test
configspace.xml
moduleconfig.xml
Test.vdm
TestVariant.vdm
AnotherTest.ccfm
SimulinkVariationPoints.ccfm
Test.ccfm
Test.xfm
TestPoints.ccfm
TestVariant.vdm
SimulinkVariationPoints.ccfm
ps:ccfm: SimulinkVariationPoints
VariationPoint: VAR_ConfigPoint
VariationPointValue: FirstVariant
VariationPointValue: SecondVariant
VariationPoint: VAR_MoreAltsTest
VariationPointValue: FirstVariant
VariationPointValue: SecondVariant
VariationPointValue: ThridVariant
Tree
Table
Constraints

Figure 4.14.: The imported VPM integrated into pure::variants

4.5.2. Exporting Variants

Within pure::variants, variants are designed using Variant Description Models as explained in Section 2.3.3.1. Therefore, the best place to get the selected configuration to be used for the Matlab scripts is such a Variant Description Model.

4.5.2.1. Extracting the Selected Configuration

Internally VDMs are organized in so called *Configuration Targets* which are stored by *Variant Elements*. These targets reference the corresponding elements in the source models. Such references include 1) the model containing the configured elements, 2) the elements that are configured and 3) the selected configuration for those elements.

Since configuration targets only reference elements by their identifiers, these elements need to be resolved in order to obtain their data. To automate this task the core plug-in of pure::variants provides the so called *TargetResolver*. The purpose of this resolver is to lookup the path of a given target and provide the corresponding element in the referenced pure::variants model.

The resolved elements are now used to generate the desired configuration script. Because the needed models are not present to this point[4], they now need to be loaded. After obtaining the *Variant Family Model* the selected configuration is propagated to this model. With this variation point selection the desired configuration script for Matlab is generated.

4.5.2.2. Implementing the Export Wizard

As user interface for exporting the desired Matlab script again a wizard was chosen. This time the `com.ps.consul.eclipse.ui.pvexport.VariantExportWizards` extension point has to be used to plug-in the implemented export wizard. This extension point requires the wizard implementing class to implement the `IVariantsExportWizard` interface provided

[4]The VPM is only loaded during import but not stored to reference it later because it is not, as the Variant Component Model, physically compatible to pure::variants models

by the pure::variants SDK. This interface provides the ability to filter model types that can be handled by the export wizard. The method used is called `isApplicable` and requires an instance of `ModelInfo` as its only argument. Since the implemented wizard can only handle Variant Description Models, it checks the type of the selected model provided in the `ModelInfo` argument to be of `VDM_TYPE` as shown in Listing 4.7.

```
@Override
public boolean isApplicable(ModelInfo info)
{
  return(info.getType().equals(ModelConstants.VDM_TYPE));
}
```

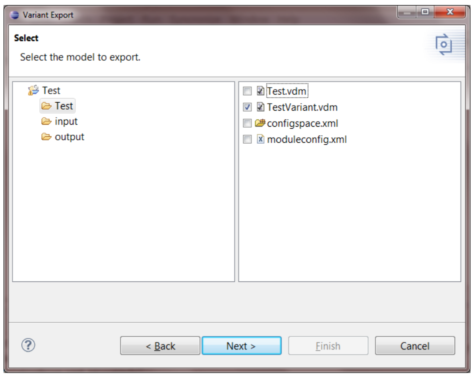
Listing 4.7: Checking if the selected model is a VDM

In the second method required by the `IVariantsExportWizard` called `setup`, initialization stuff for the wizard can be implemented. The version for the Matlab export wizard does two things in its `setup` method: 1) Extract the Family Models referenced by the selected VDM 2) Initialize and add the wizard page used by this wizard.

The Family Models are needed because a VDM can reference an arbitrary number of them and each of them needs a separate configuration.

In its `performFinish` method the tasks described in Section 4.5.2.1 are executed for all Family Models referenced by the selected VDM using the provided information gathered by the UI of this wizard.

The User Interface The user interface of the wizard is also implemented using wizard pages. Figure 4.15 shows the appearance of the implemented UI.


Variant Export

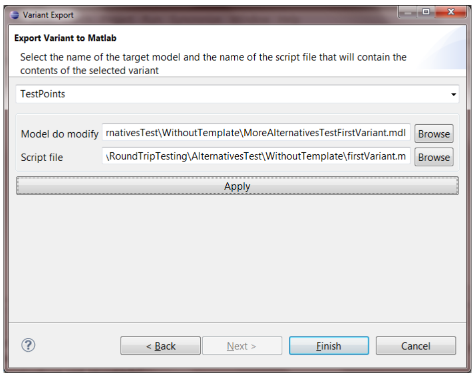
Select

Select the model to export.

- Test
  - Test
  - input
  - output

- [ ] Test.vdm
- [x] TestVariant.vdm
- [ ] configspace.xml
- [ ] moduleconfig.xml

(a) Select the VDM to export


Variant Export

Export Variant to Matlab

Select the name of the target model and the name of the script file that will contain the contents of the selected variant

TestPoints

Model do modify: rnativesTest\WithoutTemplate\MoreAlternativesTestFirstVariant.mdl

Script file: \RoundTripTesting\AlternativesTest\WithoutTemplate\firstVariant.m

(b) Configure target model and script file name

Figure 4.15.: The user interface to export a VDM into Matlab configuration scripts

To successfully generate a configuration script two pieces of information are needed: 1) the name of the script file itself and 2) the name of the model the generated script should manipulate.

Because an arbitrary number of Family Models can be referenced by VDMs it must be possible to generate an arbitrary number of configuration scripts. To offer this the implemented wizard page provides a drop down element to associate the different Family

Models with their respective target files. The wizard itself is blocked from being finished until all Family Models are associated with the required information.

4.6. Testing

This section deals with testing of the provided implementation. It does not deal with testing of the different variants that can be generated from different models. Most of the test cases are implemented using JUnit[5] tests. This framework is tightly integrated within Eclipse making it possible to run and debug the tests within the IDE. Test data was obtained by either utilizing processing rules or by models developed within Simulink. Integration testing has to be done manually, because it requires to run generated scripts within Matlab.

4.6.1. Test Strategies

4.6.1.1. Using Fixtures to Remove Complexity from Tests

Test Fixtures [Beck, 2002] provide the ability to use the same test data in multiple test cases. Therefore, the complexity of setting up the test is moved away from the test case and, much more important, is not reimplemented once per test. The test framework is responsible to invoke the generation of the fixtures and the test case itself can just use the data. There are two possible ways to define such fixtures: Fresh Fixtures and Shared Fixtures [Meszaros, 2007]. The first implies that the fixture is generated once per test, whereas the latter states that the same instance of the test fixture is used for all tests within the test suite. The tests implemented for the Variant Component Model implementation make heavy use of fresh fixtures, because a lot of tests manipulate the models that are created within the fixtures and therefore, every test needs a new fresh instance of the model to run independently.

4.6.1.2. Factories for Test Data

Test fixtures remove complexity from the test cases, but how to implement those fixtures? The core of the Variant Component Model implementation, as described in Section 4.3, already provides factories for the different elements defined by its models. Apart from the required testing of these factories, they can be used to setup test fixtures too. By composing multiple calls to these factories into one method more complex objects can be created. Using such a *Delegated Setup* [Meszaros, 2007] makes it simpler to setup the different fixtures by removing the complexity of building the composition of different model elements from the setup method of those fixtures. Utilizing the *Standard Fixture* pattern, which can be found in [Meszaros, 2007] too, further simplifies the generation of test fixtures, by reducing required parameterization when invoking the test data factories.

For example, if the standard fixture requires input ports to be named `in` and output ports `out` followed by an index starting at `1` the generation method for such components just needs to know the number of input and output ports. With no such standard fixture

[5]`http://www.junit.org`

the same method needs to take two lists representing the desired names for input and output ports.

4.6.2. Testing the Variant Component Model Core

4.6.2.1. Testing the Models

The test data used to test the Variant Component Model core is derived from the processing rules and constraints defined for it. Chapter 3 defines various constraints that imply valid models. All this constraints where taken to test if the corresponding methods 1) work as expected when those constraints are fulfilled and 2) fail as expected when those constraints are not fulfilled.

Constraint 3 for example requires variation points to be named uniquely within the VPM. This constraint implies two test cases:

Adding a new variation point This should work properly since the variation point does not yet exist.

Adding two variation points with the same name This action should fail by throwing an exception.

Based on this schema test cases against all constraints and processing rules defined for the two core models, the Variation Point Model and Variant Component Model, were implemented.

4.6.2.2. Testing Variant Generation

To test if variants are generated accurately, the number and contents of `VariantGenerationScriptOperation` instances need to be checked. In their fixture the different test cases store an instance of the *Variant Family Model* which aggregate an instance of the VPM and an instance of a composite component which represent the variant model. Each test case defines an instance of `VariantGenerationScript` with the expected operations. These operations are generated by invoking the methods the `VariantGenerationScript` provides manually. After setting up the expected result each test binds a certain variation point and invokes the variant generation algorithm by calling the *Variant Family Models* `generateVariant` method. After that, the two scripts are compared. The result of this comparison indicates whether the test was successful or not.

4.6.3. Testing the Simulink Model Provider

Test data for this tests are real Simulink models. Table 4.7 lists the different test cases developed within Simulink with their test objectives.

For each test case described in Table 4.7, a separate test suite exists. Each of these test suits defines a fixture containing the hand coded versions of the models that should be imported. This fixture definitions use the techniques shown in Section 4.6.1. Furthermore testing of variant derivation from the given models is tested within these suites using the same approach as shown in Section 4.6.2.2. Therefore, the defined fixtures can be reused directly.

Test Case Name	**Test Objective**
AlternativesTest	Test a model that uses alternative subsystems
AlternativesWithTemplateTest	Test a model that uses the template for alternative subsystems presented in Section 4.2.2
BusTest	Test a model that uses variant buses as presented in Section 4.2.4 and optional subsystems as presented in Section 4.2.3
StateMachineTest	Test a model that uses a variable state machine using the scheme defined in Section 4.2.5.

Table 4.7.: Test cases and their objectives for the Simulink model provider

4.6.4. Integration Testing

Since integration testing requires to execute scripts within Matlab, it can not be done fully automatically. Though done manually the approach of doing a *Four Phase Test* [Meszaros, 2007], also used by JUnit, is still applied. During the *Setup Phase* the test models are prepared, parsed and the required Matlab scripts are generated. In the *Exercise Phase* the generated scripts are executed within Matlab. The *Verification Phase* is executed by comparing the model generated by Matlab against a prebuilt model that represents the desired variant. The *Tear Down Phase* is accomplished by reverting the manipulated model to its state before executing the test. The models used for comparison remain the same during all phases of the tests. Each test uses a *Shard Fixture* consisting of the source model, variation points exported from this source model and one model per variant that can be derived from the source model. Therefore, each variant generated from the source model represents a separate test case. The test data used here is the same as used for testing the Simulink model provider as shown in Section 4.6.3. This process, illustrated in Figure 4.16, is applied for all models individually.

To support the test process two tools were developed for the *Setup* and *Verification* phase. The first, shown in Figure 4.17a, provides the ability to load and parse the source models and to generate the scripts to be executed within Matlab. The second tool, shown in Figure 4.17b provides the ability to compare two models. This tool is used to compare the variants generated by the scripts against those variants generated manually.

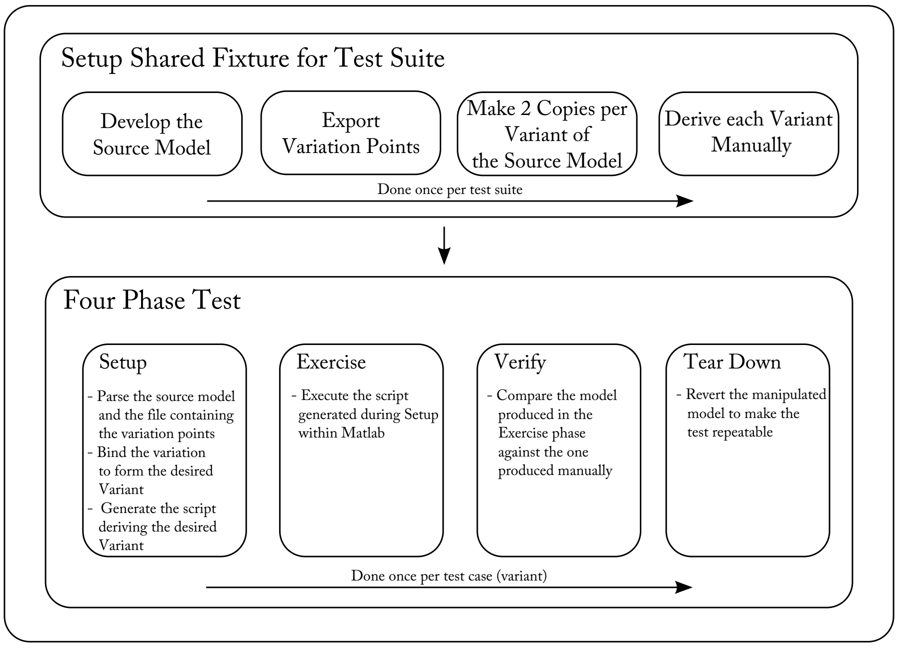
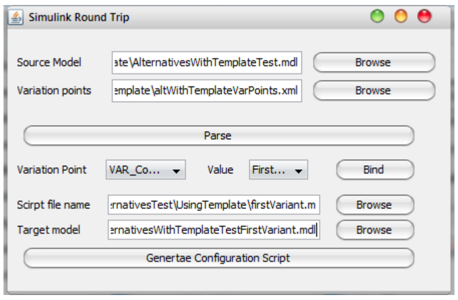
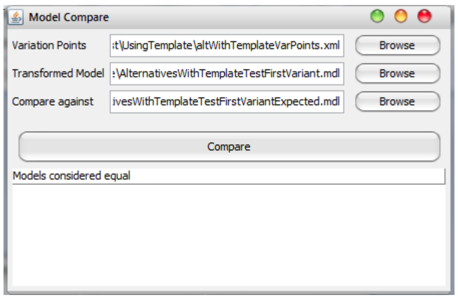

Setup Shared Fixture for Test Suite
Develop the Source Model
Export Variation Points
Make 2 Copies per Variant of the Source Model
Derive each Variant Manually
Done once per test suite
Four Phase Test
Setup
- Parse the source model and the file containing the variation points
- Bind the variation to form the desired Variant
- Generate the script deriving the desired Variant
Exercise
- Execute the script generated during Setup within Matlab
Verify
- Compare the model produced in the Exercise phase against the one produced manually
Tear Down
- Revert the manipulated model to make the test repeatable
Done once per test case (variant)

Figure 4.16.: The adoption of *Four Phase Testing* applied during integration testing



Source Model: ate\AlternativesWithTemplateTest.mdl
Variation points: emplate\altWithTemplateVarPoints.xml
Variation Point: VAR_Co... Value: First...
Scirpt file name: rnativesTest\UsingTemplate\firstVariant.m
Target model: ernativesWithTemplateTestFirstVariant.mdl

(a) Tool to setup the different test cases



Variation Points: t\UsingTemplate\altWithTemplateVarPoints.xml
Transformed Model: \AlternativesWithTemplateTestFirstVariant.mdl
Compare against: ivesWithTemplateTestFirstVariantExpected.mdl

Models considered equal

(b) Tool to compare test- and expected results

Figure 4.17.: The UIs of the tools used in the *Setup* and *Verify* phases

5. Results and Evaluation

5.1. Applying the Scenarios

The component model developed in this thesis is motivated by some scenarios provided in Section 1.2. Until now the implementation of them with means of Simulink and pure::variants is not possible. This section gives an overview how this scenarios can now be implemented using the Variant Component Model. The following sections provide details on the different scenarios in terms of 1) modeling them inside Simulink, 2) representing them using the Variant Component Model and 3) how the resulting variants that can be achieved.

5.1.1. Selection of Different Alternatives

The key aspect of this scenario is that a component can provide different alternatives from which one has to be chosen. As mentioned in Section 4.4.1.3 modeling alternatives in Simulink is achieved with *Variant Switches*. Figure 5.1 illustrates a model providing two alternative calculation methods (Add and Multiply) modeled using the mentioned mechanism.

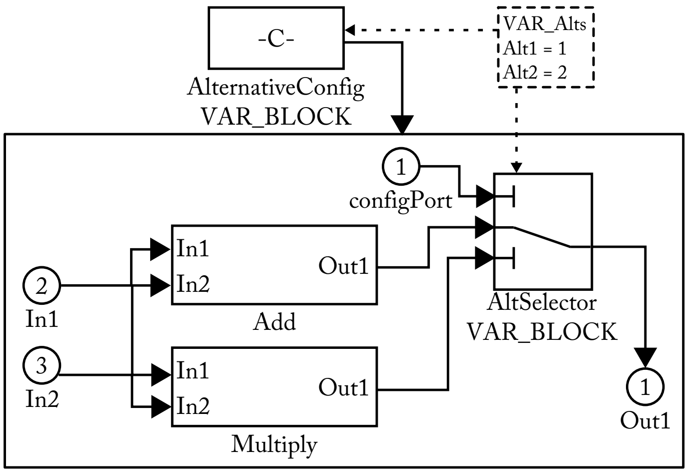


Figure 5.1.: Modeling alternatives in Simulink with the Variant Block Set

To be able to choose between different alternatives a *Variation Point* is needed that provides the configurations for the different alternatives. In the given example this *Variation Point* consists of two values `Alt1` and `Alt2`. These values are mapped to the port indices of the corresponding *Variant Multi Port Switch*. The "`AlternativeConfig`" constant provides the currently selected configuration during simulation of the variant rich model.

When minimal variants are needed the Simulink system needs to be converted into the representation defined by the Variant Component Model. A converted version of Figure 5.1 is given in Figure 5.2.

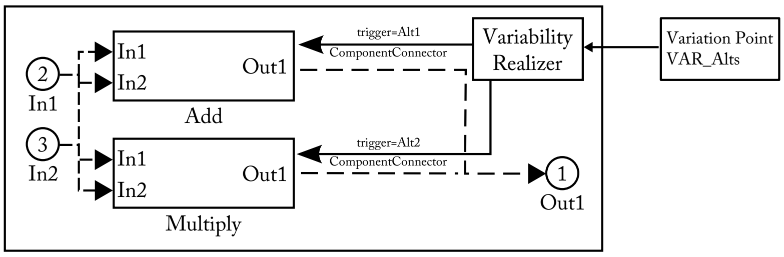


Figure 5.2.: Converted representation of the model shown in Figure 5.1

The *Variability Realizer* is used instead of the *Variant Multi Port Switch*. The switch itself and the *Variant Constant* block are removed from the model. Instead of the *Variant Constant* the variation point `VAR_Alts` is used as the desired configuration provider.

For each alternative one variant can be derived from this model. Each of these variants is represented as a separate Simulink model. Listing 5.1 gives an example of the scripts used to generate variants from Simulink models.

```
load_system('AlternativesWithTemplateTestFirstVariant');
delete_line('AlternativesWithTemplateTestFirstVariant/Subsystem',
    'Add/1', 'AltSelector/2');
delete_line('AlternativesWithTemplateTestFirstVariant/Subsystem',
    'Multiply/1', 'AltSelector/3');
delete_line('AlternativesWithTemplateTestFirstVariant/Subsystem',
    'configPort/1', 'AltSelector/1');
%(...)
delete_block( 'AlternativesWithTemplateTestFirstVariant/configPoint');
delete_block(
    'AlternativesWithTemplateTestFirstVariant/Subsystem/Subtract');
delete_block(
    'AlternativesWithTemplateTestFirstVariant/Subsystem/Multiply');
delete_block(
    'AlternativesWithTemplateTestFirstVariant/Subsystem/AltSelector');
add_line('AlternativesWithTemplateTestFirstVariant/Subsystem',
    'Add/1', 'Out1/1');
save_system('AlternativesWithTemplateTestFirstVariant');
close_system('AlternativesWithTemplateTestFirstVariant');
```

Listing 5.1: Excerpt from the generation script for the first variant selecting the `Add` – alternative from the model shown in Figure 5.1

Listing 5.1 shows how subsystems are removed including all their connections to other components. Figure 5.3 shows the different systems that can be derived based on the used *Variation Point*.

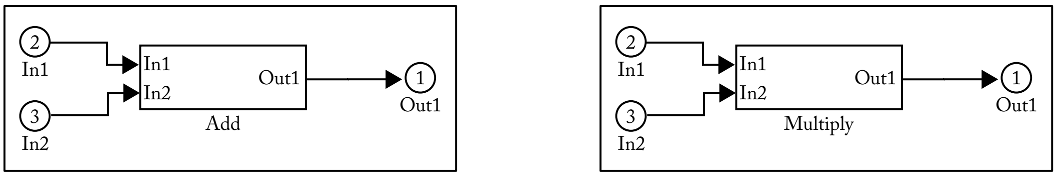


Figure 5.3.: Possible variants based on the system provided in Figure 5.1

5.1.2. Implementing Optional Behavior

As already stated in Section 1.2, implementing optional behavior is very similar to implementing alternatives. The key essence of this scenario is to provide variability on Simulink buses. Indeed, alternatives may also be connected to a bus. As shown in Section 4.2.4, two special blocks called *Variant Bus Creator* and *Variant Bus Selector* are provided by the Variant Modeling API. Figure 5.4 shows a system that uses both of these blocks each in combination with an optional subsystem.

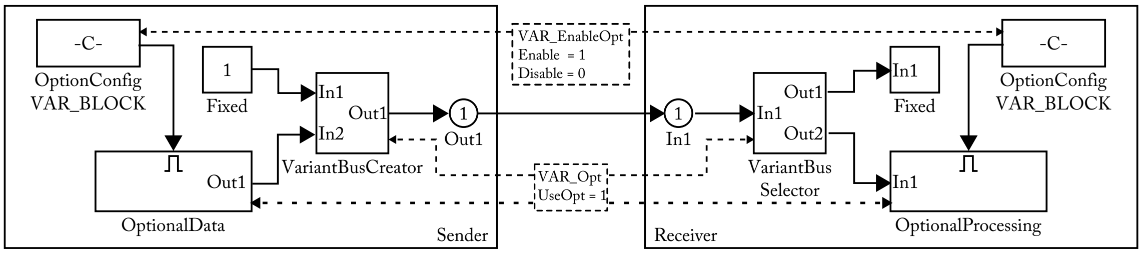


Figure 5.4.: Example on modeling optional subsystems and variable buses in Simulink with the Variant Modeling API and the Variant Block Set

As mentioned in Section 4.4.1.3, the mapping process of models that use buses involves the utilization of a special *Connector* called *BusConnector*. The model, shown in Figure 5.4, defines a *Variation Point* whose single value indicates whether or not to use the optional subsystems called `OptionalData` and `OptionalProcessing`. Note that the two *Variant Constant* blocks are only needed for simulation purposes.

Figure 5.5 shows the converted representation of the Simulink model provided in Figure 5.4. This illustrates how all non component related blocks are removed and how variant behavior is implemented using the *Variability Realizer* and the connectors provided by the Variant Component Model[1].

For the variant that includes the optional subsystems no changes in the model are necessary. Therefore, the variant that disables the optional subsystem is shown in Listing 5.2.

```
load_system('BusTestFirstVariant');
delete_line('BusTestFirstVariant', 'OptionConfig/1',
    'OptionalDisplay/enable');
delete_line('BusTestFirstVariant', 'OptionConfig/1',
    'OptionalData/enable');
delete_line('BusTestFirstVariant', 'OptionalData/1',
    'VariantBusCreator/2');
```

[1]And indeed the custom connector for buses

```
delete_line('BusTestFirstVariant', 'VariantBusSelector/2',
    'OptionalDisplay/1');
delete_block('BusTestFirstVariant/OptionalData/trigger');
delete_block('BusTestFirstVariant/OptionalDisplay/trigger');
disableBusSignal('BusTestFirstVariant/VariantBusCreator', '2');
disableBusSignal('BusTestFirstVariant/VariantBusSelector', '2');
delete_block('BusTestFirstVariant/OptionConfig');
delete_block('BusTestFirstVariant/OptionalData');
delete_block('BusTestFirstVariant/OptionalDisplay');
save_system('BusTestFirstVariant');
close_system('BusTestFirstVariant');
```

Listing 5.2: The script adjusting variant buses and removing optional sub systems

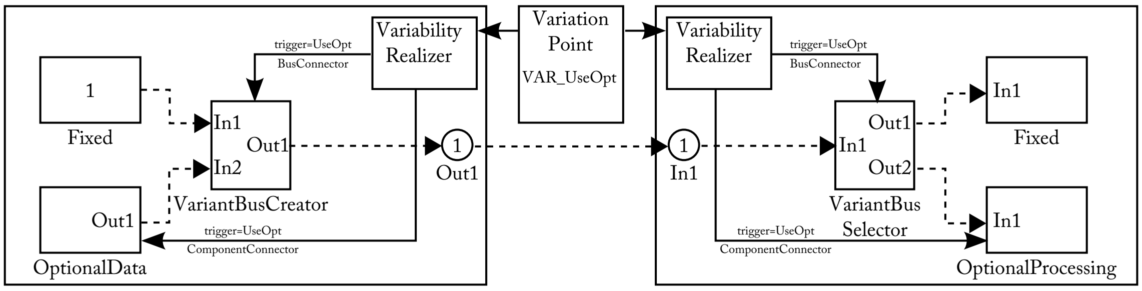


Figure 5.5.: The representation of the system provided in Figure 5.4 using the Variant Component Model

Listing 5.2 contains calls to the *disableBusSignal* function mentioned in Section 4.4.1.3. This function is responsible for adjusting *Variant Bus* blocks. Figure 5.6 illustrates the two variants that can be derived from the sample model provided in Figure 5.4.

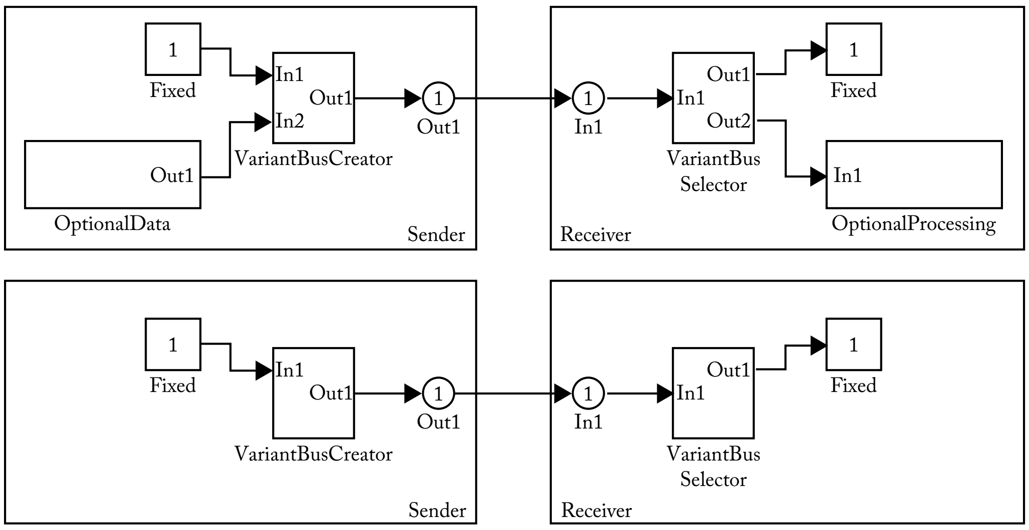


Figure 5.6.: Variants that can be derived from the system provided in Figure 5.4

5.1.3. Variable State Machines

As already mentioned in Section 1.2.3 optional states within a state machine imply two variability mechanism at once: 1) Enabling/Disabling an optional state 2) Selection of alternative transitions.

In order to separate variable from non variable parts Section 4.2.5 introduced a structure that makes it possible to identify variable parts in a certain state chart. Figure 5.7 shows a state chart defining variability.

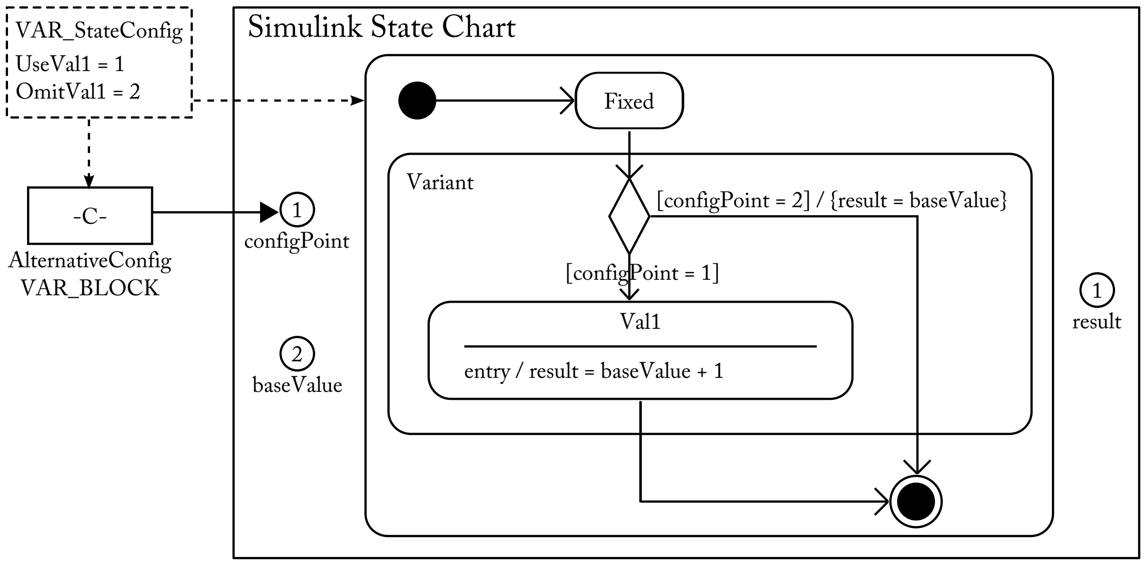


Figure 5.7.: A variant state chart that can be processed in order to extract minimal variants from it

The state chart in Figure 5.7 shows that the concept pair of a *Variarbility Mechanism / Variability Control Block*, as it is used by the Variant Block Set[Dziobek et al., 2008](see Section 2.3.3.2), is applied here. The decision point(called Junction within Simulink State Charts) acts as an indicator that variability is applied, whereas the different transitions control which variant is selected. As the most right transition in Figure 5.7 shows a transition can actually implement a variant on its own using its action. In the given example this is done by assigning `baseValue` to `result`. Figure 5.8 illustrates how the state chart given in Figure 5.7 can be implemented using the Variant Component Model.

This model shows that the *Connector* concept implemented in Section 3.4.5 can also be applied to state machines. The usage of specialized components is needed because states, for example, may only reference other states or transitions, but, for example, must not reference a multiplexer component.

Based on the model provided in Figure 5.8 two different variants can be generated. Listing 5.3 shows the most relevant parts of the script generated to derive the variant including the optional state.

```
load_system('StateflowTestFirstVariant');
delete_line('StateflowTestFirstVariant', 'AlternativeConfig/1',
    'VAR_MyChart/2');
```

```
delete_line('StateflowTestFirstVariant/VAR_MyChart', 'configPoint/1',
    '␣SFunction␣/2');
reRouteTransition('VAR_MyChart', 31, 25, 30)
rewriteTransitionLabel('VAR_MyChart', 31, '');
delete_block('StateflowTestFirstVariant/VAR_MyChart/configPoint');
deleteTransition('VAR_MyChart', 29);
deleteTransition('VAR_MyChart', 11);
delete_block('StateflowTestFirstVariant/AlternativeConfig');
deleteStateFromStateMachine('VAR_MyChart', 6);
save_system('StateflowTestFirstVariant');
close_system('StateflowTestFirstVariant');
```

Listing 5.3: The script that generates the variant that includes the optional state defined in Figure 5.7

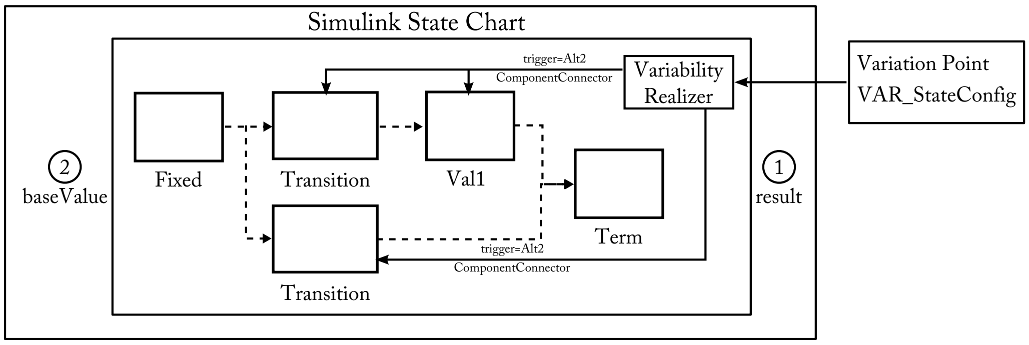


Figure 5.8.: Implementing the state machine from Figure 5.7 using the Variant Component Model

The numbers used as arguments in the various calls to adjust the state chart correspond to the `SSIDNumber` parameters generated by Simulink State Charts for states and transitions. Note the calls to `reRouteTransition` and `rewriteTransitionLabel` are used to bypass elements removed from the model that deal with variant selection, i.e. the `Junction` illustrated as *Desicion* element in Figure 5.7.

Figure 5.9 show the minimal state machine variants that can be achieved.

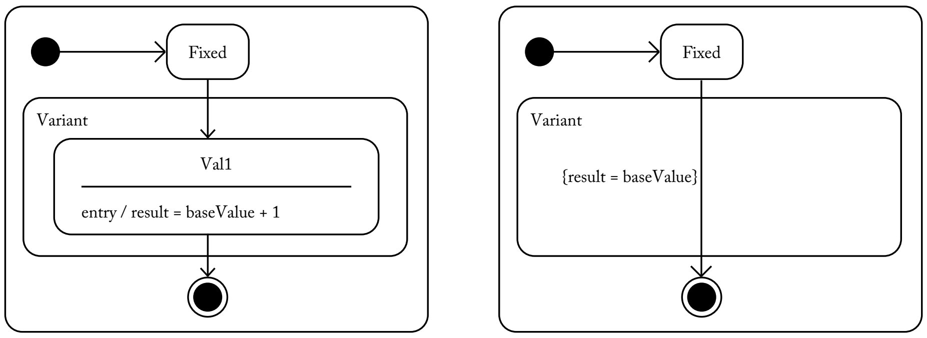


Figure 5.9.: The variants that can be derived based on the model from Figure 5.8

5.2. Applying the Variability Patterns

To evaluate the applicability of the patterns provided in [Reiser et al., 2009](see Section 2.2.6) the concept needs to be separated from the implementation. Some patterns are not applicable for Simulink models although they can be achieved using the concept provided in Chapter 3. Therefore, the next section provides details on how the patterns can be applied to the Variant Component Model, and Section 5.2.2 provides details on patterns that can be applied to Simulink models.

5.2.1. Variant Component Model

5.2.1.1. Plain Propagation

Plain Propagation[Reiser et al., 2009](see Section 2.2.6.1) can be achieved by utilizing the *Variability Connector*. Imagine there are two components, `Parent` and `Child` with `Child` defining two variants, `VarOne` and `VarTwo`. These two variants can be added to the *Variability Realizer* of `Parent`. To propagate the selected variant back to `Child`, `Parent` defines two `Variability Connectors`, one for each variant, connecting `Parent's` *Variability Realizer* and `Child`. Each *Variability Connector* sets the same value that is triggering it, therefore plainly propagating the configuration selected for `Parent`. Additionally, the *Variability Realizer* of `Parent` defines both variants provided by `Child`. Figure 5.10 illustrates this.

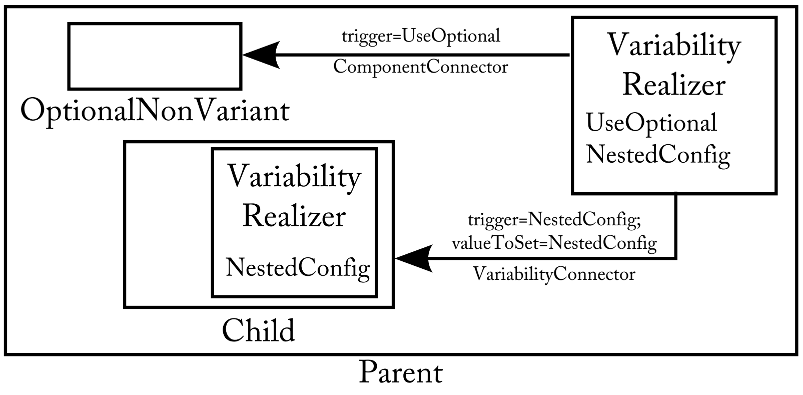


Figure 5.10.: Applying *Plain Propagation* to Variant Component Model components

If a *VariationPoint* is connected to `Parent` and provides a variant that needs to be propagated to `Child` the registered *Variability Connector* informs the *Variability Realizer* of `Child` about the selected variants, as shown in Section 3.4.5.2.

5.2.1.2. Direct Binding

Within the Variant Component Model *Direct Binding*[Reiser et al., 2009](see Section 2.2.6.2) can be achieved through a special *Variation Point* that always provides the same value. Although the Variation Point Model presented in Section 3.3 does not directly provide such a variation point, its constraints only require that each *Variation Point* provides at least one variant(see Constraint 5). A “constant” *Variant Point*, always providing the same variant is therefore accepted by the model.

5.2.1.3. Orthogonal Propagation

Orthogonal Propagation [Reiser et al., 2009] (see Section 2.2.6.3) is interesting, because it supports the mapping from a variant accepted by a component to a variant accepted by another component, therefore promoting reusability of variant components. Components which are part of the Variant Component Model support this again by using *Variability Connectors*. Since multiple connectors can be defined for the same trigger, different variants can also be merged into one, as shown in [Reiser et al., 2009]. Take again our Components `Parent` and `Child`. This time we don't want to export variant behavior of `Child` into the interface of `Parent` but let certain variants defined for `Parent` configure certain variants for `Child`. This time the used *Variability Realizers* do not just propagate the trigger, they define a certain variant out of the set provided by `Child`. Using this scheme, variants selected for *Parent* are converted to the one expected by *Child*. Figure 5.11 illustrates this design.

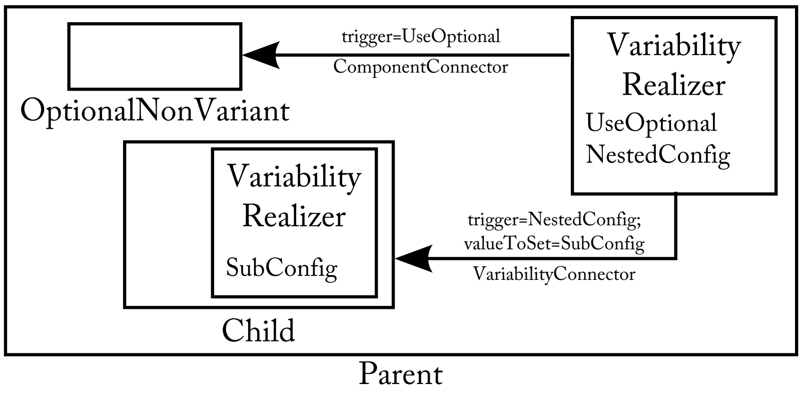


Figure 5.11.: Orthogonal propagation of variability inside the Variant Component Model

5.2.1.4. Top Level Propagation

There is no direct connector in the Variant Component Model that supports *Top Level Propagation* as defined in [Reiser et al., 2009] (see Section 2.2.6.4). Using connectors for this kind of propagation would make little sense either, because this would imply some form of, at least, *Plain Propagation.* To achieve *Top Level Propagation* in the Variant Component Model the *Variability Realizer* of any component can directly subscribe to a *Variation Point* no matter to which hierarchy level the component is assigned to.

5.2.1.5. Global Features

Global Features (see Section 2.2.6.5) are directly supported by the Variant Component Model via the *Aspect* concept shown in Section 3.4.6. *Aspects* in the Variant Component Model are defined globally and components that want to contribute to this aspects connect their contributing parts to the desired aspect.

5.2.2. Implementation

To analyze the implementation it is first important to analyze the abilities provided by Simulink. Simulink does not provide the ability to model all patterns. Variation points within Simulink can only be defined globally. Simulink subsystem can not define their "private Feature Model" as would be required. Essentially, variation points within Simulink are more related to *Global Features* than to the local Feature Models required by most of the patterns. Therefore, every subsystem no matter on which hierarchy level consumes the same "Feature Model", in terms of Simulink variation points. There is just one tweak here: The required usage of *Constant Block* to "propagate" the selected configuration to *Variability Mechanisms* as shown in Section 2.3.3.2. This can be used to implement a *Direct Binding* facility, by not using the *Variant Constant* block from the Variant Block Set but the built-in *Constant* block built into Simulink.

5.3. Discussion

Section 5.1 has shown the application of the Variant Component Model on the scenarios provided in Section 1.2. Furthermore, the proposed concept supports all patterns for variability propagation(see Section 2.2.6). Unfortunately, the implementation can not come up with such a support. The reason is, that the implementation can only support patterns that are applicable to its source models. Since Simulink only supports a subset of the patterns, as shown in Section 5.2.2 only this subset is supported by the implementation. Using another model source other patterns would be supported, for example models based on [de Jonge, 2004] would support *Orthogonal Propagation*. Table 5.1 summarizes the results provided in this chapter.

Objective	**Concept**	**Implementation**
Scenario		
Selecting Alternatives	✓	✓
Optional Components	✓	✓
Optional States	✓	✓
Variability Patterns		
Plain Propagation	✓	-
Direct Binding	✓	✓
Orthogonal Propagation	✓	-
Top Level Propagation	✓	✓
Global Features	✓	✓

Table 5.1.: Scenarios and patterns supported by the Variant Component Model and its Simulink based implementation

As the scenario part of Table 5.1 indicates the Variant Component Model now defines the needed *ModelConfiguration* binding time as described in [Beuche and Weiland, 2009] for Simulink models. Simulink models can now be reduced to specific variants. By utilizing concepts of the Variant Block Set and implementing an explicit meaning to certain Simulink blocks this reduction step is now explicit and is not hidden behind code generator options as shown in Section 2.3.3.2.

Beside the scenarios, the provided component model and its implementation fulfill almost all requirements defined in Section 3.1. The analysis of the scenarios has shown, that the Variant Component Model is capable of generating minimal and valid variants based on a provided configuration as required by Requirement 1. Reusability as required by Requirement 2 is achieved through separating configuration providers from configuration consumers, i.e. components, on one hand, and on the other by providing the ability to define mappings between physically incompatible, yet logically compatible, configuration sets. The *Variant Modeling* library for Simulink provides seamless integration for modeling variability in Simulink based models as required by Requirement 3. The VPM, as a submodel of the Variant Component Model, provides explicit mechanisms do define dependencies between variation points. By providing *Aspects* as explicit model element inside the Variant Component Model crosscutting effects as required in Requirement 5 can be explicitly modeled. The only exception to fulfilled requirements is Requirement 6 which requires interoperability to AUTOSAR. But since the Variant Component Model is decoupled from concrete model sources the fulfillment of this requirement can be achieved by implementing a model provider that is capable of reading AUTOSAR based system descriptions. Section 6.1 provides more details on this issue.

6. Summary and Outlook

The goals of this thesis were to 1) provide a component model that provides the ability to model variability of components and 2) to integrate this component model in Simulink.

Section 1.2 has introduced three scenarios, based on Simulink, how variability may occur in model based software development. These are:

1. Selecting between alternative components.
2. Enabling / Disabling of optional components.
3. Selecting alternative / optional states and / or transitions in state machines.

Based on these scenarios, this thesis provides a component model called *Variant Component Model* that provides 1) a hierarchical structure for component composition and 2) explicit model elements to implement variability of components. With the Variant Component Model it is also possible to implement different propagation patterns for variability.

The implemented model is divided into two parts:

- The Variation Point Model and
- the actual components that provide variability.

This separates the configurable entities, called *Variability Consumers*, from the entities that provide the actual configuration, called *Variation Points* and makes it possible to reuse *Variability Consumers* independently from the used Variation Point Model.

Based on this component model the prototype, shown in Chapter 4, has been developed. First, this prototype integrates Simulink models into the Variant Component Model and second, provides mechanisms to integrate variability into Simulink models. Integration of Simulink models is done by mapping certain concepts from Simulink to concepts provided by the Variant Component Model, e.g. *Simulink Subsystems* to Variant Component Model *Composite Components*. Section 3.5.3 proposes two alternative ways how variants from Simulink models can be derived. The implemented prototype uses the second alternative, which means to remove items from a copy of the variant rich source model. Chapter 5 shows that the goals of this thesis have been reached by showing how the different scenarios can be implemented and how variability information can be propagated between Variant Component Model components based on the appropriate patterns mentioned earlier.

6.1. Future Work

6.1.1. Further Integration into pure::variants

In a productive environment it is reasonable that variation points may be adjusted inside pure::variants. Currently this has to be done inside Simulink. The VPM needs to be regenerated to synchronize it with the data currently present within pure::variants. The Variant Block Set provides an API to manipulate and create variation points within Simulink programmatically. Therefore, this API only needs to be instrumented using a similar mechanism used in this thesis to remove blocks from Simulink models.

6.1.2. Further Automate Generation of Variants

The implemented variant derivation process requires the *Variant Developer* (see Section 3.5.1.5) to provide a copy of the source model which can be manipulated. Although the step of creating these copies may be automated it is more reasonable that the variant derivation process creates a new variant specific model on its own, as described in Section 3.5.3.

6.1.3. Interface to AUTOSAR

Currently no implemented interface between the Variant Component Model and AUTOSAR is provided. To provide such an interface two steps need to be carried out:

1. Map AUTOSAR variability concepts to those provided by the Variant Component Model
2. Map AUTOSAR component descriptions to those of the Variant Component Model

Providing such mappings enables usage of AUTOSAR system descriptions whithin the Variant Component Model. Furthermore, is gets possible to directly export variants derived from Variant Component Model into an AUTOSAR system description.

6.1.4. Translation of Component Description Languages

Having a meta model that can mediate between different representations of the same information makes it indirectly possible to convert between these representations. The Variant Component Model is such a meta model and component description languages, such as AUTOSAR and Simulink do provide similar information on the implemented system. Therefore, it may be possible to read a model in format A and write it in format B, e.g. read Simulink and write AUTOSAR.

6.1.5. Configuration of Parameters

Blocks in Simulink provide the ability to define parameters for them. Such parameters are grouped in so called *Masks*. For example, `Constants` get their concrete value from a parameter called `Value`. It is reasonable that different variants may define different values for certain parameters, e.g. different values for `Constants`. To provide this ability components may store certain parameters in their definition. Additional connectors to these definitions may than provide the variant specific value for them.

A. State Chart API

This chapter provides details on the API provided with the *Variant Modeling* library to manipulate state charts. This API is utilized by Matlab scripts that need to reduce state machines to specific variants.

A.1. Delete States from State Charts

To delete a state from a state chart the function `deleteStateFromStateMachine` is provided. The state chart is identified by its name and the state by its `SSIDNumber`. Both values are provided as arguments to the mentioned function (see Table A.1). The function itself first tires to find the state chart with the given name and then the state itself. If both are found, the state is deleted by calling its destructor function provided by Simulink. In case the state or the state chart are not found a appropriate error message is prompted.

Function Name:	`deleteStateFromStateMachine`	
Returns:	nothing	
Error conditions:	State chart not found State not found	
Arguments		
Name	**Type**	**Description**
stateMachineName	String	The name of the state chart.
stateId	int	The `SSIDNumber` of the state that should be removed

Table A.1.: Synopsis of `deleteStateFromStateMachine`

A.2. Delete Transitions from State Charts

Deleting transitions is similar to deleting states as described above. The function `deleteTransition` provides this scheme for transitions. Again the name of the state chart the a `SSIDNumber` identifying the transition have to be provided. Table A.2 provides the complete synopsis of this function.

A.3. Rerouting Transitions

Rerouting transitions means to change the source and / or the target state of a certain transition. The function `reRouteTransition` implements this behavior. It always requires

Function Name:	`deleteTransition`	
Returns:	nothing	
Error conditions:	State chart not found Transition not found	
Arguments		
Name	**Type**	**Description**
stateMachineName	String	The name of the state chart.
transitionId	int	The `SSIDNumber` of the transition that should be removed

Table A.2.: Synopsis of `deleteTransition`

the identifiers of both the source and the target state. If a state remains the same the unmodified identifier of this state can be used to indicate that it should be kept. Table A.3 provides the complete synopsis of this function.

Function Name:	`reRouteTransition`	
Returns:	nothing	
Error conditions:	State chart not found Transition not found New Source State not found New Target State not found	
Arguments		
Name	**Type**	**Description**
stateMachineName	String	The name of the state chart.
transitionId	int	The *SSIDNumber* of the transition that should be removed.
newTransitionSource	int	The *SSIDNumber* of the new source state.
newTransitionTarget	int	The *SSIDNumber* of the new target state.

Table A.3.: Synopsis of `reRouteTransition`

A.4. Changing Labels of Transitions

The function `rewriteTransitionLabel` provides the ability to change the label of a certain transition identified by its `SSIDNumber`. Note that it is not possible to just change parts of the label, e.g. the condition. If parts from the old label should be kept, they need to be part of the newly provided label. The function performs no validation on the new label. Table A.4 provides the complete synopsis of this function.

Function Name:	`rewriteTransitionLabel`	
Returns:	nothing	
Error conditions:	State chart not found Transition not found	
Arguments		
Name	**Type**	**Description**
stateMachineName	String	The name of the state chart.
transitionId	int	The *SSIDNumber* of the transition that should be removed.
newLabel	String	The new label the given transition should use.

Table A.4.: Synopsis of `rewriteTransitionLabel`

B. Refactoring Simulink Models for Variability

Section 4.2 provides the required structure for different representations of variability in Simulink. This chapter provides details on how existing models can be refactored to meet these structural requirements. A simple pattern language [Buschmann et al., 2007] is used to define those refactorings. This language consists of four parts:

- A problem description,
- a short description of the solution,
- a graphical workflow showing how to achieve the solution and
- a description of the different steps.

All refactorings have a common precondition: A *Variation Point* has been defined that covers the desired variants.

B.1. Introducing Optional Subsystems

The Problem

Parts of the model need to be part of certain variants but have to be omitted in others.

The Solution

Introduce an optional subsystem that contains all of these parts in order to disable it in variants where it is not needed.

The Workflow

Figure B.1 illustrates the steps to achieve an optional subsystem from an existing model.

Description of the Different Steps

Introduce new Subsystem

All optional parts need to be grouped into a single subsystem. If such a subsystem is already present, this step can be omitted.

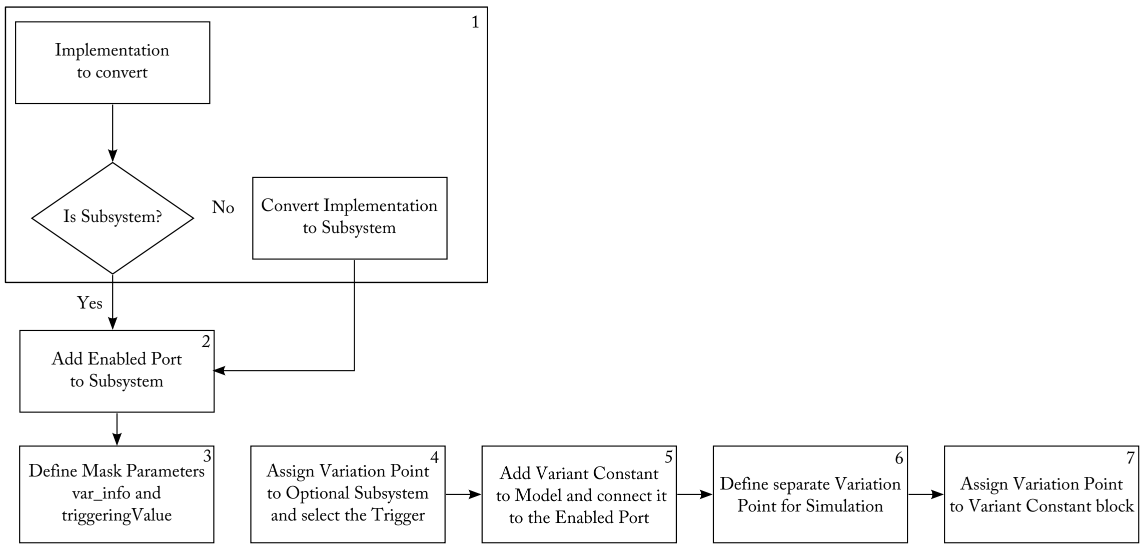

1
Implementation to convert
Is Subsystem?
No
Convert Implementation to Subsystem
Yes
2
Add Enabled Port to Subsystem
3
Define Mask Parameters var_info and triggeringValue
4
Assign Variation Point to Optional Subsystem and select the Trigger
5
Add Variant Constant to Model and connect it to the Enabled Port
6
Define separate Variation Point for Simulation
7
Assign Variation Point to Variant Constant block

Figure B.1.: Workflow to convert parts of a model into an optional subsystem

Make the Subsystem Toggleable

In order to be able to simulate different variants within the whole solution family, an `Enabled Port` is needed. This makes it possible to switch off the optional subsystem without generating variants for small test runs. Therefore, such a port has to be added to the optional subsystem.

Define Mask Parameters and Assign Concrete Values to them

As shown in Section 4.2.3, optional subsystems need to define at least two mask parameters called `var_info` and `triggerValue` that provide the configuring variation point and the value that enables the optional subsystem. Therefore, these two parameters need to be defined.

After defining those parameters the desired values need to be assigned to them.

Add a Variant Constant

To make use of the previously added `Enabled Port`, a block is needed that provides configuration to it. Therefore, a new `Variant Constant` block has to be added to the model. Since this block will provide the configuration during simulation of the solution family it has to be connected to the `Enabled Port` of the optional subsystem.

Define Variation Point for Simulation

Since `Enabled Ports` only accept two values (0 means disabled, >0 means enabled) a variation point is needed that can provide these values. Since it is not feasible to mix variation points meant to control the model with variation points only needed for simulation, a separate variation point for that purpose has to be defined. It has only to store two values indicating whether or not the optional subsystem should be enabled during simulation.

This newly introduced variation point has to be assigned to the `Variant Constant` block introduced in the last step.

B.2. Introducing Alternative Implementations

The Problem

Parts of the model need to be handled differently in different variants.

The Solution

Collect different implementations for different variants into dedicated subsystems and switch them according to the variant they implement, thus making these implementations alternative.

The Workflow

Figure B.2 illustrates the workflow, showing how different existing implementations can be refactored into alternative subsystems.

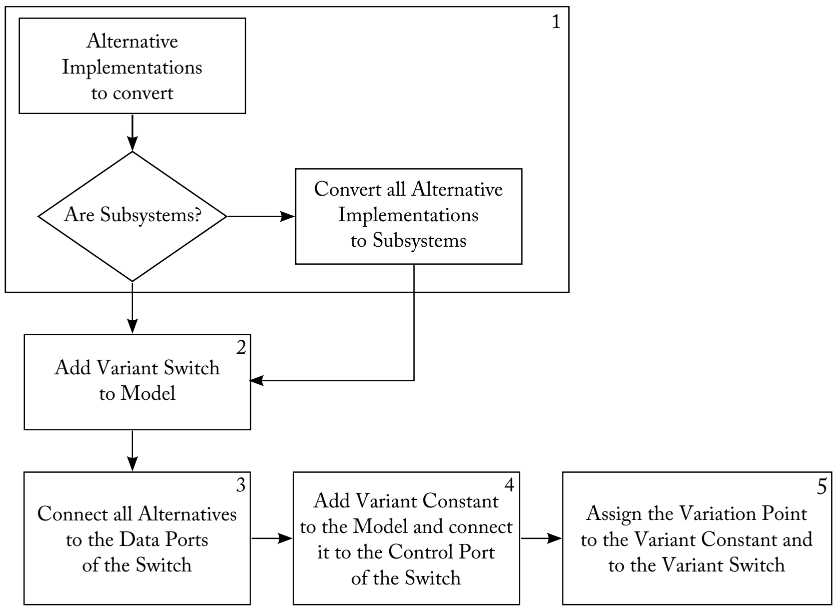

1
Alternative Implementations to convert
Are Subsystems?
Convert all Alternative Implementations to Subsystems
2
Add Variant Switch to Model
3
Connect all Alternatives to the Data Ports of the Switch
4
Add Variant Constant to the Model and connect it to the Control Port of the Switch
5
Assign the Variation Point to the Variant Constant and to the Variant Switch

Figure B.2.: Workflow how to refactor existing implementations to alternatives

Description of the Different Steps

Convert Implementations to Subsystems

Each alternative implementation has to be represented as a separate subsystem. Therefore, all parts belonging to an alternative have to be grouped into a subsystem. This step can be omitted if such subsystems already exist.

Add a Variant Switch

Section 4.2.2 states that a *Variant Switch* has to be used to model the selection of alternative subsystems. Although both types, `Variant Switch` and `Variant Multiport Switch` are supported it is reasonable to always use the `Variant Multiport Switch` because it is more flexible.

Connect the Alternatives

All alternative subsystems have to be connected to data ports of the *Variant Switch* added in the last step. Note, that the ordering of the alternatives on the *Variant Switch* is important, as described in Section 4.2.2.

Add a Configuration Provider

For simulation purposes an explicit configuration provider is necessary. The workflow illustrated states to use a `Variant Constant`. Alternatively, a dedicated configuration port can be used which is connected to such a constant. In any case, the used configuration provider has to be connected to the control port of the used `Variant Switch`.

Finally the variation point controlling which alternative to use has to be assigned to the used *Variant Constant* and to the used *Variant Switch*.

Bibliography

[Autosar, 2009a] Autosar (2009a). AUTOSAR BSW & RTE Conformance Test Specification Part 1: Background. http://www.autosar.org/download/R4.0/AUTOSAR_PD_BSWCTSpecBackground.pdf.

[Autosar, 2009b] Autosar (2009b). AUTOSAR BSW & RTE Conformance Test Specification Part 2: Process Overview. http://www.autosar.org/download/R4.0/AUTOSAR_PD_BSWCTSpecProcessOverview.pdf.

[Autosar, 2009c] Autosar (2009c). AUTOSAR BSW & RTE Conformance Test Specification Part 3: Creation & Validation. http://www.autosar.org/download/R4.0/AUTOSAR_PD_BSWCTSpecCreationValidation.pdf.

[Autosar, 2009d] Autosar (2009d). AUTOSAR BSW & RTE Conformance Test Specification Part 4: Execution Constraints. http://www.autosar.org/download/R4.0/AUTOSAR_PD_BSWCTSpecExecutionConstraints.pdf.

[Autosar, 2009e] Autosar (2009e). Software Component Template. http://www.autosar.org/download/R3.2/AUTOSAR_RS_SoftwareComponentTemplate.pdf.

[Autosar, 2009f] Autosar (2009f). Specification of RTE. http://www.autosar.org/download/R3.2/AUTOSAR_SWS_RTE.pdf.

[Autosar, 2009g] Autosar (2009g). Applying Simulink to AUTOSAR. http://www.autosar.org/download/AUTOSAR_SimulinkStyleguide.pdf.

[Beck, 2002] Beck, K. (2002). Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Berg et al., 2005] Berg, K., Bishop, J. and Muthig, D. (2005). Tracing software product line variability: from problem to solution space. In Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries SAICSIT '05 pp. 182–191, South African Institute for Computer Scientists and Information Technologists, Republic of South Africa.

[Beuche, 2003] Beuche, D. (2003). Composition and Construction of Embedded Software Families. PhD thesis, Otto-von-Guericke-Universität Magdeburg.

[Beuche and Weiland, 2009] Beuche, D. and Weiland, J. (2009). Managing Flexibility: Modeling Binding-Times in Simulink. In Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications ECMDA-FA '09 pp. 289–300, Springer-Verlag, Berlin, Heidelberg.

[Broy, 2006] Broy, M. (2006). Challenges in automotive software engineering. In Proceedings of the 28th international conference on Software engineering ICSE '06 pp. 33–42, ACM, New York, NY, USA.

[Buschmann et al., 2007] Buschmann, F., Henney, K. and Schmidt, D. C. (2007). Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. Wiley.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996). Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley.

[Clements and Northrop, 2001] Clements, P. and Northrop, L. (2001). Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Crnkovic, 2002] Crnkovic, I. (2002). Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA.

[Dauenhauer et al., 2009] Dauenhauer, G., Aschauer, T. and Pree, W. (2009). Variability in Automation System Models. In Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering ICSR '09 pp. 116–125, Springer-Verlag.

[de Jonge, 2004] de Jonge, M. (2004). Multi-level Component Composition. In 2nd Groningen Workshop on Software Variability Modeling (SVM'04), (Bosch, J., ed.), number 2004-7-01.

[Dziobek et al., 2008] Dziobek, C., Loew, J., Przystas, W. and Weiland, J. (2008). Functional Variants Handling in Simulink Models. last visited: 20.02.2011.

[Eisemann et al., 2009] Eisemann, U., Stichling, D. and Stroop, J. (2009). Successful AUTOSAR Migration. Online version, german: `http://www.elektroniknet.de/automotive/technik-know-how/test-entwicklungstools/article/1629/0/Erfolgreiche_AUTOSAR-Migration/`, last visited: 10.10. 2011.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E. and Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[Harald et al., 2004] Harald, H., Klaus-Peter, S., Helmut, F., Jürgen, B., Lennart, L., Jean, L., Jean-Luc, M., Kenji, N. and Thomas, S. (2004). AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures. In Convergence International Congress & Exposition On Transportation Electronics pp. 325–332,.

[Haugen et al., 2008] Haugen, O., Moller-Pedersen, B., Oldevik, J., Olsen, G. and Svendsen, A. (2008). Adding Standardized Variability to Domain Specific Languages. In Software Product Line Conference, 2008. SPLC '08. 12th International pp. 139 –148,.

[Haugen et al., 2010] Haugen, O., Moller-Pedersen, B., Olsen, G. K., Svendsen, A., Fleurey, F. and Zhang, X. (2010). Model driven development of highly configurable embedded Software intensive Systems.

[Kajtazović, 2011] Kajtazović, N. (2011). Evaluation of variant management capabilities of automotive software engineering tools. Master's thesis Technische Universität Graz Austria.

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report Software Engineering Institute Carnegie Mellon University.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997). Aspect-oriented programming. In ECOOP'97 — Object-Oriented Programming vol. 1241, of Lecture Notes in Computer Science pp. 220–242. Springer-Verlag Berlin/Heidelberg.

[Kum et al., 2008] Kum, D., Park, G.-M., Lee, S. and Jung, W. (2008). AUTOSAR migration from existing automotive software. In International Conference on Control, Automation and Systems, 2008. ICCAS 2008. pp. 558 –562,.

[Lacouture and Aniorté, 2008] Lacouture, J. and Aniorté, P. (2008). CompAA : A Self-Adaptable Component Model For Open Systems. In Fifteenth IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2008) pp. 19 –25,.

[McRitchie et al., 2004] McRitchie, I., Brown, T. J. and Spence, I. T. (2004). Managing Component Variability within Embedded Software Product Lines via Transformational Code Generation. In Software Product-Family Engineering vol. 3014, of Lecture Notes in Computer Science pp. 98–110. Springer Berlin / Heidelberg.

[Meszaros, 2007] Meszaros, G. (2007). XUnit Test Patterns: Refactoring Test Code. Addison-Wesley.

[Object Management Group, 2006] Object Management Group (2006). Meta Object Facility (MOF) Core Specification Version 2.0. `http://www.omg.org/spec/MOF/2.0/PDF/`.

[Pohl et al., 2005] Pohl, K., Böckle, G. and van der Linden, F. J. (2005). Software Product Line Engineering: Foundations, Principles and Techniques. Springer.

[pure-systems GmbH, 2009] pure-systems GmbH (2009). pure::variants User's Guide. `http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf`. last visited: 20.02.2011.

[Reiser et al., 2009] Reiser, M.-O., Kolagari, R. and Weber, M. (2009). Compositional Variability - Concepts and Patterns. In 42nd International Conference on System Sciences, 2009. HICSS '09. Hawaii pp. 1 –10,.

[Schmidt, 2006] Schmidt, D. (2006). Guest Editor's Introduction: Model-Driven Engineering. Computer *39*, 25 – 31.

[Sora et al., 2004] Sora, I., Cretu, V., Verbaeten, P. and Berbers, Y. (2004). Automating Decisions in Component Composition Based on Propagation of Requirements. In FASE vol. 2984, of Lecture Notes in Computer Science pp. 374–388, Springer.

[Szyperski, 1997] Szyperski, C. (1997). Component Software: Beyond Object-Oriented Programming (ACM Press). Addison-Wesley Professional.

[Voelter and Visser, 2011] Voelter, M. and Visser, E. (2011). Product Line Engineering using Domain-Specific Languages. In 14th International Conference on Software Product Lines (SPLC 2011), Proceedings CPS.

[Webber and Gomaa, 2004] Webber, D. L. and Gomaa, H. (2004). Modeling variability in software product lines with the variation point model. Sci. Comput. Program. *3*, 305–331.