PESOA

Process Family Engineering in Service-Oriented Applications

BMBF-Project

Variability Mechanisms for Process Models

Authors:
Frank Puhlmann
Hasso-Plattner-Institut
Arnd Schnieders
Hasso-Plattner-Institut
Jens Weiland
DaimlerChrysler
Research and Technology
Mathias Weske
Hasso-Plattner-Institut

PESOA-Report No. TR 17/2005
June 30, 2005

Abstract

This report describes the representation of variant rich process models and the derivation of concrete process models using variability mechanisms. Therefore based on a study of existing variability mechanisms, architecturally relevant variability mechanisms are identified and their transfer first onto generic processes and then onto UML Activity Diagrams, UML State Machines, BPMN, and Matlab/Simulink is described. A number of practical examples demonstrates the application of the approach.

PESOA is a cooperative project supported by the federal ministry of education and research (BMBF). Its aim is the design and prototypical implementation of a process family engineering platform and its application in the areas of e-business and telematics.
The project partners are:

- DaimlerChrysler AG
- Delta Software Technology GmbH
- ehotel AG
- Fraunhofer IESE
- Hasso-Plattner-Institute
- University of Leipzig

PESOA is coordinated by
Prof. Dr. Mathias Weske
Prof.-Dr.-Helmert-Str. 2-3
D-14482 Potsdam

www.pesoa.org

Table of Contents

1 Introduction

One of the main objectives of the PESOA project consists in investigating an approach for the development of families of process oriented software. One key concept of product family oriented software development is that reuse shall take place on any stage of the software development process, i.e. not only code shall be reused to a maximum, but potentially also any other software development artifact, like architecture or design models. For the optimal reuse of software development artifacts so called variability mechanisms play a crucial role. Variability mechanisms allow for the derivation of artifact variants from generic artifacts. While the derived artifact variant is typically specific for a concrete member of the product line, the generic artifact has features, which are common for more than one member of the product line.

Up to now existing variability mechanisms mostly target at the static aspects of a software system's model, while approaches for process oriented software as being dealt with in process family engineering have been neglected. Therefore, the intention of this report is to analyze thoroughly how variability mechanisms for process models can be acquired.

For this we will access the considerable number of variability mechanisms, which have already been published. These variability mechanisms can be classified according to the point in the product line lifecycle at which they perform the resolution of the variability, which is also referred to as the binding time. In our case only those variability mechanisms are relevant which resolve variability at the architecture or design model level or which at least have a visible impact on the architecture or design of a system. For the sake of simplicity we will call these variability mechanisms *architecturally relevant variability mechanisms*. Our approach for obtaining variability mechanisms for process models is to investigate the transferability of architecturally relevant variability mechanisms to process models. We take this approach as we assume that only architecturally relevant variability mechanisms can be transferred to process models since also process models of process oriented systems only contain architecture or design related information.

Our proceeding therefore is first to identify architecturally relevant variability mechanisms and secondly describe their usability in process oriented architecture models. We thereby neither claim that the described variability mechanisms for process oriented architecture models form a complete or minimal set.

This report is structured as follows: Section 2 gives an overview of existing variability mechanisms and divides them into architecturally and non-architecturally relevant variability mechanisms. Section 1 describes funda-

mental process concepts based on which the architecturally relevant variability mechanisms from section 2 will be described for generic processes. Section 4, 5, 6, 7 describe how these variability mechanisms can be represented in UML Activity Diagrams, UML State Machines, BPMN, and Matlab/Simulink. Section 8 summarizes the main contents of this report and gives an outlook to future research.

2 Survey and Categorization of Existing Variability Mechanisms

As already mentioned one of the main goals of product line oriented software development is to maximize the reuse of software development artifacts (like design models, code, etc.) for the members of the product line. For this purpose the product line utilizes generic artifacts. We refer to these artifacts as generic since they have properties, which are common to more than one member of the software product line. But usually the generic artifacts cannot be used as is for a concrete product line member. The reason for this is that typically the requirements of the members of the product line, which require the generic artifact, differ slightly. Therefore, a generic artifact normally has to be adapted to the specific requirements of a concrete product in order to be usable by that product. For being adaptable to the requirements of a concrete member of the product line, a generic artifact disposes of variation points, which specify the parts in which the requirements of the concrete products vary. This allows the generic artifact for being adapted to the requirements of a concrete product by binding variants, which suit the needs of the concrete product, to every variation point of the generic artifact.

For the adaptation of the generic artifacts so-called variability mechanisms are required. According to [JGJ97] variability mechanisms are techniques for specializing abstract components. [SvB03] more generally denotes variability mechanisms as “techniques available for introducing variability into the software product line”. Since we focus on processes in this report we don’t consider the first definition appropriate, while the second one can be concretized more. Therefore we suggest the following definition of variability mechanisms:

Variability mechanisms denote techniques for the derivation of process model variants from existing process models.

The careful selection of an appropriate variability mechanism for specializing an artifact is important since the selection of an unsuitable variability mechanism may lead to the problem that a generic artifact cannot be adapted when required. Because of their importance for the product line oriented software development, variability mechanisms have been investigated extensively, leading to the identification of a great number of variability mechanisms [AnG00, Bos00, CIN02, JGJ97, SvB03]. However, for our purpose to define variability mechanisms for process architecture models only “architecturally relevant” variability mechanisms are of interest. A variability mechanism is considered to be architecturally relevant, if it has a visible impact on the architecture of a system. In addition to variability mechanisms on the architecture level, we differentiate between variability mechanisms on the product

and on the runtime level. Product time comprises those types of variability which are resolved during the implementation of a single product, while the runtime level refers to resolution of variability at runtime. In 2.13 variability mechanisms for the product and runtime level are described. In 3 we then describe the usability of the architecturally relevant variability mechanisms in process oriented architecture models.

In the remainder of this section we will give a brief overview of some of the most important architecturally relevant variability mechanisms.

2.1 Information Hiding

Information hiding as described by [Gom04, Gom05] means that different versions of a component facilitating different members of a product line can be encapsulated by means of a common interface. *Information hiding* is also referred to by [GBS01] as the utilization of black box components.

2.2 Inheritance

Inheritance together with the related concept Polymorphism represents an important variability mechanism that serves as the basis for several other variability mechanisms like *Extensions* and *Design Patterns* besides being usable in isolation. Due to its relevance for product family engineering *inheritance* is addressed in most publications dealing with variability mechanisms like in [Bos00, Gom05, JGJ97, SvB03]. *Inheritance* is used as a variability mechanism on the model level as well as on the code level.

2.3 Parameterization

According to [BaB01, Gom05, JGJ97, SvB03] using *parameterization* component variants are generated by configuring the generic components with a set of parameter values. The prerequisite for this is that all possible variants are provided in the component's code. Parameterization is used typically if there are many small variation points, which causes minor changes to the system for a variant feature [JGJ97]. Following the definition of [ClN02] features are user-visible aspects or characteristics of a system and are typically organized in tree-structures during the domain analysis.

2.4 Templates

Templates [Bos00, JGJ97, SvB03] are a technique that allows for postponing the decision on which type a process shall work, until the time the process is performed instead of the time of its implementation.

2.5 Null-Classes

The optional parts of a component's behavior can be sourced out into a separate class. Now, if the optional behavior shall be omitted a null class can be generated that acts as a placeholder for the class containing the optional behavior [SGB03].

2.6 Interface Separation

As already mentioned, variability in product line architectures can be realized by replacing components. Thereby, the replacing component variant can have a different interface than the replaced component. In order to be able to restrict the variability to the product architecture derivation without having to make adaptations at the code level later, both the provided interface of the varying component as well as the required interface of the component importing the varying component can be sourced out into separate classes. The configuration management tool can then decide which interface classes to use together with which component variants. This *interface separation* technique is described in [SGB03].

2.7 Design Patterns

Certain *Design Patterns* are frequently referred to as a variability mechanism [Bos00, Gom05, SvB03]. "Gang of Four" *Design Patterns* [GHJ95] used as variability mechanisms are the "*Adapter*", "*Strategy*", "*Template Method*", "*Factory*", "*Abstract Factory*", and "*Builder*" pattern. Moreover, the "*Broker Pattern*" [BJM96] can be used as a variability mechanism, as well as the "*Single Adapter*", "*Multiple Adapter*", and "*Option*" Pattern [KeM99]. However, according to [Sch97], except for a small number of *Design Patterns* any *Design Pattern* provides a way to implement variability.

2.8 Replacement of Components

In static models entire components can be replaced by other components, which is described by [ClN02]. In general, if a component is replaced by another component it has to be considered that related components may not be compatible to the interface of the replacing component any more.

2.9 Omission of Components

Variations within a static model can be realized by omitting components entirely, as described by [ClN02]. The question arises what happens to the relations of the components connected to the component to be omitted, i.e. with the required and provided interfaces of the related components? They also have to regard the absence of the component they referred to originally.

2.10 Extensions and Extension Points

Extensions and *Extension Points* [Bos00, ClN02, JGJ97, SvB03] are used if a component can be extended at a certain predefined point by additional behavior selected from a set of possible variants.

2.11 Addition of Components

Components can also be added to static structures at arbitrary points in the diagram. One question is how the new component is connected to the remaining diagram. The difference between extensions/extension points and the addition of components is that for extensions in contrast to the addition of components a placeholder (the extension point) is provided for the component to be added, while this isn't the case if components are added. The addition of components is outlined as a variability mechanism by [ClN02].

2.12 Delegation/Aggregation

As described in [ClN02] the functionality of an object can also be extended by delegating the calculations the object cannot perform on its own to another object encapsulating the (varying) functionality for performing the respective calculations. Alternatively, the invoked object can also be aggregated by the invoking object.

2.13 Further Variability Mechanisms

In addition to variability mechanisms, which focus on the product architecture derivation phase, there is a great number of variability mechanisms being clearly targeted at other points in time during the lifecycle of the product line and therefore will not be considered here. Variability mechanisms at product time [ClN02] are *automatic generation* [ClN02, JGJ97, SvB03], *conditional compilation* [BaB01, ClN02, SvB03], *frames* [ClN02], *static libraries* [ClN02], *scripting* [GBS01], *configuration* [JGJ97, SGB03, SvB03], *if-statements* [SGB03], and *binary replacement* [GBS01]. Variability mechanisms applied at runtime are, for example, the *dynamic binding* of components at runtime [ClN02, GBS01, SGB03] and *reflection* [ClN02]. Additionally

there are variability mechanisms which don’t fit in any of these binding time categories, like for example *Use Case inheritance* [JGJ97].

3 Variability Mechanisms for Generic Processes

This section starts with an introduction of the fundamental process concepts based on which we can describe variability mechanisms for process models generically. Our intention thereby is rather to introduce the terminology we will use, than to define a process meta model.

Next, we describe the transfer of variability mechanisms identified to be relevant on an architecture model level on generic process models as described previously. Thereby, we group the variability mechanisms in basic and composite variability mechanisms. Basic variability mechanisms do not rely on other variability mechanisms, while composite variability mechanisms require the application of basic variability mechanisms.

3.1 Fundamental Process Concepts

A process can consist of subprocesses, which can be encapsulated hiding the details of the encapsulated subprocess behind a subprocess interface. Thereby, subprocesses can be combined as single execution steps. Moreover, subprocesses can be invoked synchronously and asynchronously from within the process either receiving return values or not. Subprocess steps can have incoming and outgoing control and data flow edges and contain process execution steps, which are interconnected by control and data flow edges. Control and data flow edges describe the control and data flow within the process. The data flow is characterized by the data types exchanged between the processing steps in the process. The behavior of an execution step can be parameterized as well as the control and data flow edges of a process. Parameterization of the control flow leads to changes in the routing within the process while parameterization of the data flow leads to variation in the processed data types. The control flow can also depend on the data types forwarded in the data flow. The behavior of subprocesses can also depend on their input data types. Also data storages can be represented in a process model. Data storages can only contain data of a certain type.

3.2 Basic Variability Mechanisms

3.2.1 Encapsulation of Varying Subprocesses

Assuming that subprocesses in process models are the analog concept to components in component models, subprocess interfaces can be defined, which hide details concerning the internal structure of the subprocess. This

allows for the insertion of different subprocess variants hidden by the invariant interface.

3.2.2 Addition, Replacement, Omission of Encapsulated Subprocesses

In process models encapsulated subprocesses can be added, replaced or omitted at potentially any place in the process model. Thereby, it has to be paid attention that the addition, replacement or omission of an encapsulated subprocess doesn't lead to a structurally incorrect process description. If a subprocess is added the respective data flow and control flow edges are interrupted by the newly added subprocess. If a subprocess is replaced by another subprocess especially the compatibility of the interface of the replacing subprocess with the preceding and succeeding elements of the replaced subprocess has to be regarded. If an encapsulated subprocess is omitted this means that the respective execution step is ignored upon execution of the process. The control flow is therefore continued at the successors of the omitted execution step. Concerning the data flow the potential absence of data transformations originally performed by the omitted execution step has to be tolerable by subsequent process model elements. In subsequent execution steps the behavior depending on the transformed data has to be omitted as well.

3.2.3 Parameterization

Through parameterization behavioral variants which are integrated in the process can be activated by configuring the process with corresponding parameter values. Theoretically, by parameterization variability in the control flow as well as in the processed data or the behavior of single execution steps can be controlled.

3.2.4 Variability in Data Types

In process models variability in data types reflects in the dataflow exchanged between subprocesses. Also the control flow within a process can depend on the type of data being forwarded. The type dependency of calculations whose details are hidden on the process model level, can also be represented in process models. If data storages are represented in the process model, variations in the type of data they store can be represented.

3.3 Composite Variability Mechanisms

3.3.1 Inheritance

Specialization of subprocesses in process models corresponds to the specialization of components or classes in static diagrams. Inheritance allows for the replacement or addition of model elements in the derived process diagram.

3.3.2 Design Patterns

Generally speaking, design patterns based on information hiding and inheritance like the Strategy design pattern can be represented in processes using encapsulation of varying subprocesses and process inheritance.

3.3.3 Extensions/Extension Points

A process shall be extendible at certain places by encapsulated subprocesses, whether the extending subprocess has been predefined during the product line infrastructure development or not. The place where the process can be extended is referred to as extension point. Thus a variant-rich process model should provide means for inserting encapsulated optional subprocesses at these extension points. An extending encapsulating subprocess must have a compatible interface in order to be integrable into the process at the corresponding extension point.

Extensions/extension points can be used together with Null-Subprocesses, which are integrated into a variation point if the optional behavior shall be omitted. The Null-subprocess has the same interface as the process to be omitted but doesn't contain any visible behavior.

Processes can also be extended by delegating functionality to external subprocesses. Delegation can be realized by invoking external subprocesses synchronously or asynchronously. Moreover, optional return values may be processed subsequently. Encapsulated processes can be aggregated by means of extensions.

4 Variability Mechanisms for UML Activity Diagrams

This section analyzes the transfer of the variability mechanisms for generic processes outlined in 3 onto UML Activity Diagrams [OMG03, Pen03].

Concerning the regarded part of the Activity Diagram specification, we will concentrate on the IntermediateActivities package and will use only modeling elements from other packages if it's unavoidable for modeling the respective variability mechanism. The concentration on the IntermediateActivities package leads to a reduction of complexity for the definition of variability mechanisms for Activity Diagrams. On the other hand, this reduction is not that critical since Intermediate Activities comprise modeling elements, which are absolutely sufficient for many applications.

4.1 Basic Variability Mechanisms

4.1.1 Encapsulation of Varying Subprocesses

Assuming that Actions are the analog concept for components in Activity Diagrams, the variability mechanism information hiding applies to varying Activities hidden behind the invariant interface of the invoking CallBehaviorAction. The interface of the CallBehaviorAction is represented by the number, types and ordering of its Input and OutputPins as well as the classifier assigned to the CallBehaviorAction as its context. The invoked Activity variants must have the same interface as the invoking CallBehaviorAction. An Action-Interface can be applied for realizing alternative as well as optional behavior. The latter can be realized by invoking a *Null-Activity*.

4.1.2 Adding Actions

The *addition* of Actions in Activity Diagrams corresponds to the addition of execution steps in generic process models. Theoretically, the addition of Actions can happen at arbitrary places in the process flow. However, due to the layout guidelines for Activity Diagrams, the addition of Actions has to be restricted to the insertion between two ActivityNodes. Insertions between multiple ActivityNodes are not allowed. If an Action is added into an Activity Diagram the rules for adding Actions depend on the point in the Activity Diagram where the Action shall be added. If the preceding and succeeding ActivityNode is an Action, an Action can be added whose input and output interfaces are compatible to the interfaces of the surrounding Actions. If only

one of the surrounding ActivityNodes is an Action, the Input and OutputPin of the added Action have to be compatible to the adjunctive preceding or succeeding Action. If neither the predecessor nor the successor is an Action the Input and OutputPin of the inserted Action have to be compatible to the object type transported by the ActivityEdge connecting the preceding and succeeding ActivityNode.

4.1.3 Replacing Actions

The *replacement* of an execution step in process models corresponds to the replacement of an Action Act by an Action Act-R in Activity Diagrams. During the replacement of an Action by another Action in Activity Diagrams the compatibility of the replaced Actions has to be regarded.

4.1.3.1 Interface Compatibility

In order to define the replacement of Actions in Activity Diagrams, interface compatibility of Actions has to be defined. This is described in the following.

Type-compatibility of two input pins

An InputPin IP_1 is *type-compatible* to an InputPin IP_2, if the type of IP_1 is equal to or a supertype of the type of IP_2. IP_2 can then be replaced by IP_1.

An OutputPin OP_1 is *type-compatible* to an OutputPin OP_2, if the type of OP_1 is equal to or a subtype of OP_2. OP_2 can then be replaced by OP_1.

Type-compatibility of an input pin to an output pin and vice versa

An OutputPin OP is type-compatible to an adjunctive InputPin IP, if the type of OP is equal to or a subtype of the type of IP.

An InputPin IP is type-compatible to an adjunctive OutputPin OP, if the type of IP is equal to or a supertype of OP.

Compatibility of the input interfaces of two Actions

Given:

- an Action Act1 with an ordered number of input pins
  IP_ACT1 = { IP_1^1, …, IP_m^1 }

- an Action Act2 with an ordered number of input pins
  IP_ACT2 = { IP_1^2, …, IP_n^2 }

The input interface of the Action Act1 is compatible to the input interface of the Action Act2, if

- Act1 and Act2 have the same number of input pins:
  $|IP\_ACT1| = |IP\_ACT2|$

- Every input pin of Act1 is type compatible to the corresponding input pin of Act2:
  $\forall IP_x^1 \in IP\_ACT1 \wedge \forall IP_y^2 \in IP\_ACT2 : x = y \rightarrow$ IP_x^1 is type-compatible to IP_y^2

Compatibility of the output interfaces of two Actions

Analogue to the compatibility definition for the input interfaces of two Actions.

Compatibility of the output interface of an Action to the input interface of another Action

Given:

- an Action Act1 with an ordered number of output pins
  OP_ACT1 = { OP_1^1, …, OP_m^1 }

- an Action Act2 with an ordered number of input pins
  IP_ACT2 = { IP_1^2, …, IP_n^2 }

The output interface of an Action Act1 is compatible to the input interface of an Action Act2, if

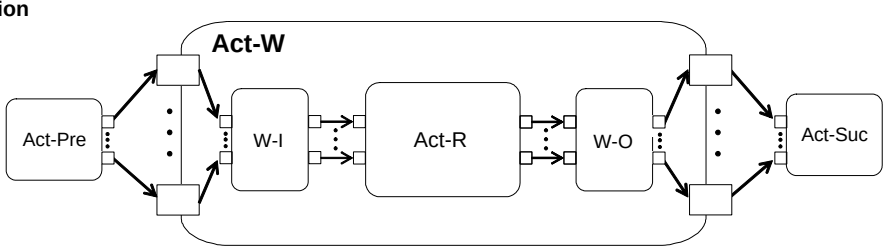
- Act1 has just as many output pins as Act2 has input pins:
  $|OP\_ACT1| = |IP\_ACT2|$

- Every output pin of Act1 is type compatible to the corresponding input pin of Act2:
  $\forall OP_x^1 \in OP\_ACT1 \wedge IP_y^2 \in IP\_ACT2 : x = y \rightarrow OP_x^1$ is type-compatible to IP_y^2

4.1.3.2 Rules for replacement of Actions

If Act-R has a compatible interface to Act it can replace Act offhand. If the interface of Act-R is incompatible to the interface of Act, a wrapper Action Act-W can be wrapped around Act-R that provides an invariant interface compatible to the interface of Act by means of information hiding as shown in Figure 1. In Activity Diagrams the application of a wrapper Action requires that Act has only one preceding and one succeeding Action, since the outgoing arcs of several Actions could else meet in Act-W or the ingoing arcs of many successor-Actions could go out from Act-W.

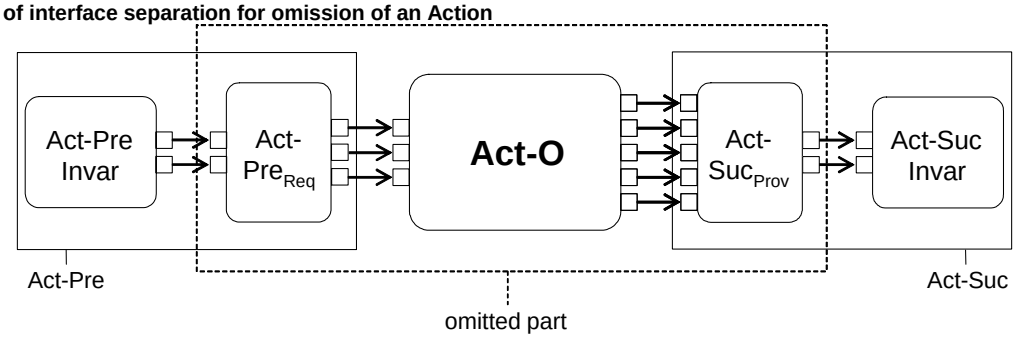
Figure 1: Wrapper Action



Alternatively, required and provided interfaces of the Actions Act-Pre and Act-Suc preceding and succeeding an Action Act-R, which invokes a varying subprocess, are encapsulated each in separate Actions Act-Pre_{Req} and Act-Suc_{Prov} and are replaced together with Act-Rep. Thereby, Act-Pre_{Req} and Act-Suc_{Prov} provide an invariant interface to Act-Pre and accordingly to Act-Suc using *information hiding*. Likewise the separated required and provided interfaces of the preceding and succeeding Actions of the Action to be omitted can be omitted together with the Action.

The application of interface separation is illustrated in Figure 2. In this example the Action Act-O shall be omitted. In order to retrieve a syntactically correct Activity Diagram the required interface of Action Act-Pre and the provided interface of the Action Act-Suc are separated and deleted together with Act-O.

Figure 2: Application of interface separation for omission of an Action



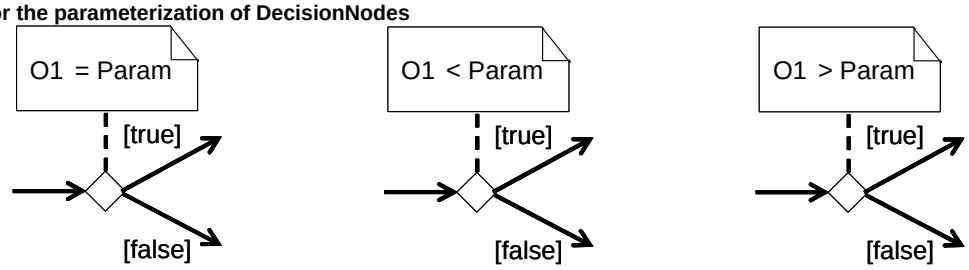
4.1.4 Omitting Actions

The omission of Actions in Activity Diagrams shall be restricted to the case that the Action to be omitted has exactly one preceding and succeeding Action. More sophisticated cases could require complicated adaptations of preceding ForkNodes and succeeding DecisionNodes and JoinNodes, which can depend on the output data types of the Action. Cases like this should be better handled by *parameterization*. For the case that the output interface of the Action Act-Pre preceding Act-O is compatible to the input interface of the Action Act-Suc succeeding Act-O, Act-O can be omitted offhand. If this is not the case, the required interface of Act-Pre or the provided interface of Act-Suc, or both, can be sourced out using interface separation and omitted together with Act-O as shown in Figure 2.

4.1.5 Parameterization

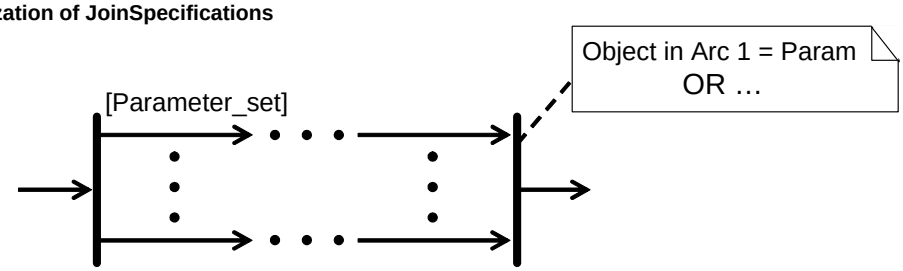
Parameterization for Activity Diagrams means that variations provided in an Activity Diagram must be enactable by setting parameter values. If a parameter value can be selected from a set of several possible parameter values, this corresponds to the realization of a “range variation point” as described in [BBG05]. Theoretically, by parameterization variants in the control flow, the processed data and the behavior of single Actions of a process can be activated. One way for realizing variations in the control flow is to parameterize decisionInputBehaviors (DecisionNodes) as shown in Figure 3. In the cases depicted here the upper path is only taken if an input object has a value that equals to the parameter “Param” (left DecisionNode), smaller (DecisionNode in the middle) or bigger than the parameter (DecisionNode on the right).

Figure 3: Examples for the parameterization of DecisionNodes



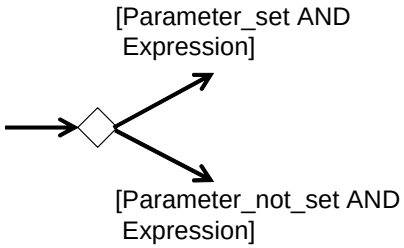
Alternatively, variations in the control flow can be realized by parameterizing the JoinSpecifications of JoinNodes, which is indicated in Figure 4. This allows for the parameterization of the conditions under which a JoinNode will issue a token. In this case, the JoinNode will fire once there is a token in Arc 1 containing an object whose value equals to “Param” or another condition comes true which is not expressed here explicitly. Since a JoinSpecification is normally not displayed in an Activity Diagram, in Figure 4 it is made visible using an UML comment.

Figure 4: Parameterization of JoinSpecifications



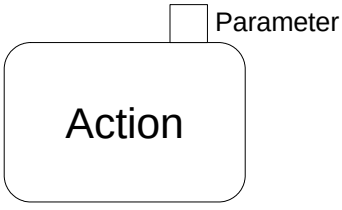
Variations in the control flow can also be realized by assigning guard expressions to ActivityEdges, whose value depends on the parameter value, which is illustrated in Figure 5. A token entering the DecisionNode will take the upper branch in case the Variable “Parameter_set” contains the value “true” and the lower branch if it contains the value “false”.

Figure 5: Parameterization of ActivityEdges using parameterized guards



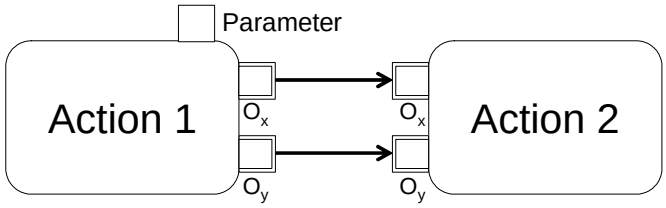
Actions can be parameterized using ValuePins. Thus it is possible to parameterize entire processes, which are invoked by the parameterized Action in case the parameterized Action is a CallBehaviorAction. This is illustrated in Figure 6.

Figure 6: Parameterization of Actions using ValuePins



The data forwarded between two Actions is parameterizable by applying ParameterSets and ValuePins. In the example in Figure 7 there can be two alternative data flows between “Action 1” and “Action 2”. Object “O_x” will be forwarded if “Parameter” has a certain value and “O_y” if it has another value. How “Action 1” realizes that a token is issued via one or the other ParameterSet in dependence of its configuration is transparent at this level of abstraction and can be specified by a separate Activity invoked by the Action.

Figure 7: Example for parameterizing the data flow between two Actions



4.1.6 Variability in Data Types

On a process model level the utilization of templates is reflected in variations of the data types passed between the Actions processing them. Also the control flow may depend on the type of data being forwarded. The dependency of the type of data an atomic Action performs its calculations can also be depicted. Data storages represented in the process may contain different types of data subject to their configuration. The difference between the variability mechanisms *parameterization* and *templates* is that using *parameterization* the data values in an Activity Diagram can be configured as described in 4.1.5, while using templates the respective data types can be adapted to the needs of a certain process variant. Thus, the same elements being subject to parameterization can also be subject to type change. Additionally, the type of an ObjectNode can be changed, as well as the types of Pins, ActivityParameterNodes and CentralBufferNodes.

4.2 Composite Variability Mechanisms

4.2.1 Inheritance

According to the Activity Diagram Inheritance definition in [ScP05], a subactivity CA inherits from its superactivity PA according to the following schema: $CA = PA \oplus \Delta CA$. ΔCA comprises elements that shall be newly added or that are already present in PA and shall be overwritten. $\oplus$ designates the combination of PA with ΔCA that adds the new elements and replaces existing ones which are subject to modification. These transformations being feasible during the derivation of subactivities using Activity Diagram Inheritance are shown in Figure 8 and Figure 9. Simple Activity Diagram elements can be replaced by simple Activity Diagram elements or subprocesses. Likewise, Activity Diagram subprocesses can be replaced by simple elements or subprocesses. Some basic rules for the addition and replacement of encapsulated subprocesses are described in 4.1.2 and 4.1.3. Also simple Activity Diagram elements as well as subprocesses can be added during derivation. The respective transformations described in

[ScP05] also guarantee syntactical and structural correctness for the derived Activity Diagrams.

Figure 8: Activity Diagram inheritance replacement transformations

replaces	Simple Element	Subprocess
Simple Element	X	X
Subprocess	X	X

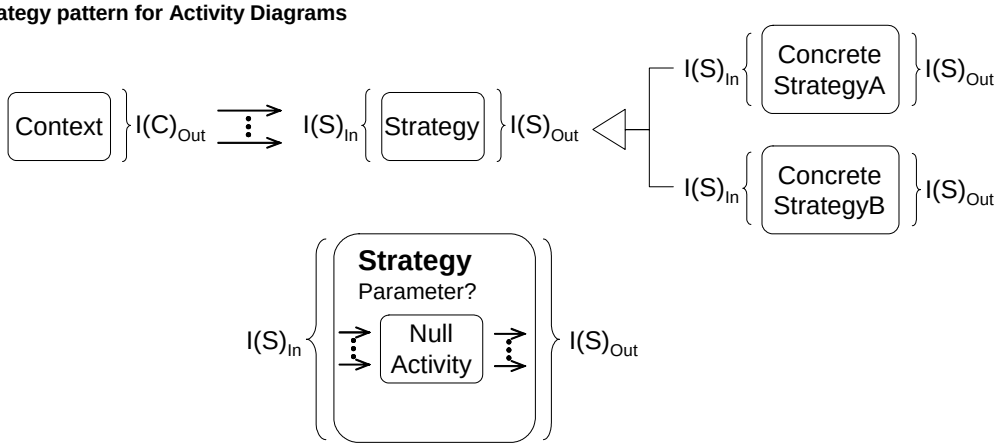
Figure 9: Activity Diagram inheritance addition transformations

is added	
Simple Element	X
Subprocess	X

4.2.2 Design Patterns

Here we will concentrate on the "*Strategy Pattern*" as one of the Design Patterns referenced most frequently in the context of Process Family Engineering (PFE). In Activity Diagrams the "*Strategy Pattern*" is realized by employing a Strategy-Action that contains a *Null-Action* and provides an invariant interface using *information hiding*. Utilizing *Activity Diagram Inheritance* different variants of the Strategy-Action can be derived by applying the required Activity Diagram Inheritance transformations on the *Null-Action* contained in the Strategy-Action. The strategy pattern is illustrated in Figure 10.

Figure 10: Strategy pattern for Activity Diagrams

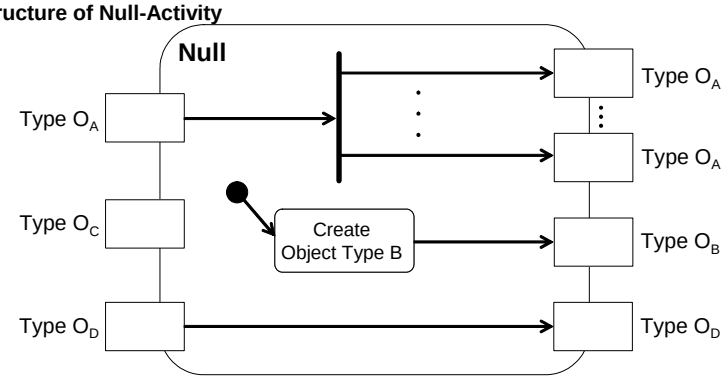


4.2.3 Extensions/Extension Points

In Activity Diagrams extension points can be realized using a CallBehaviorAction that invokes either a *Null-Activity* or an Activity containing an extending subprocess, which is shown in Figure 12. The Null-Activity performs no processing and has the same interface as the Action invoking the subprocess to be omitted. The structure of a Null-Activity is displayed in Figure 11. In case an input object is expected to be doubled by subsequent Actions as

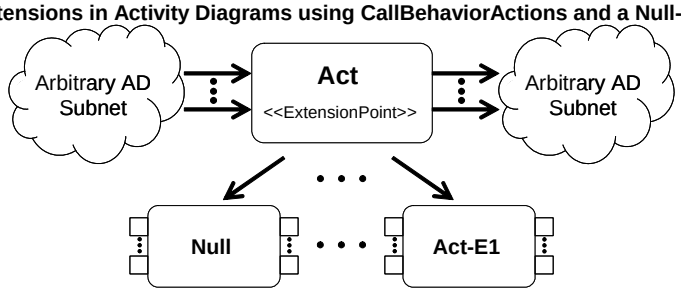
this is the case for object of type O_A, a ForkNode performs the duplication. If an object is not required in the subsequent process any more, it is discarded by the Null-Activity. This is shown exemplarily for the object O_A. Alternatively, objects can also be forwarded without any modification, which is shown for object O_D. Apart from a Null-Activity implementation the variability mechanism *information hiding* is required for providing Null-Activities.

Figure 11: **Structure of Null-Activity**



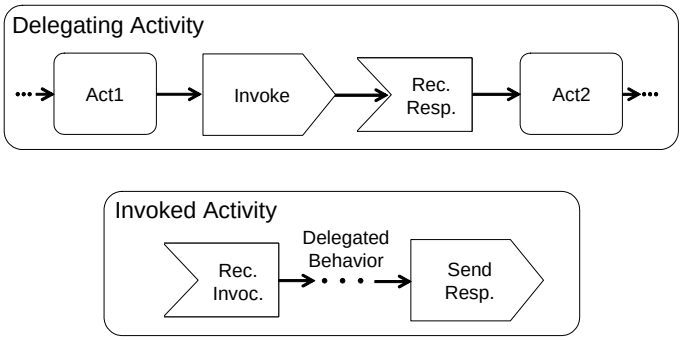
The CallBehaviorAction invoking the extending subprocess uses *information hiding* for encapsulating the possible extensions. Alternatively the *Strategy Pattern* (see 4.2.2) can be used for realizing extensions in Activity Diagrams.

Figure 12: **Extensions in Activity Diagrams using CallBehaviorActions and a Null-Activity**



Activity Diagrams can also be extended by aggregation or delegation. Delegation can be realized in Activity Diagrams by invoking external Activities synchronously or asynchronously using SendSignalActions possibly processing return values of the invoked Activities. The synchronous invocation of an external Activity is shown in Figure 13. The external Activity is invoked and the processing proceeds with “Act2” only after the delegating Activity has received the return value.

Figure 13: **Synchronous invocation of an external Activity**



In contrast to the synchronous invocation of external Activities the processing can continue right away after the external Activity has been invoked. The asynchronous invocation of an external Activity without subsequently processing a return value is shown in Figure 14.

Figure 14: **Asynchronous invocation of an external Activity without receipt of a return value**

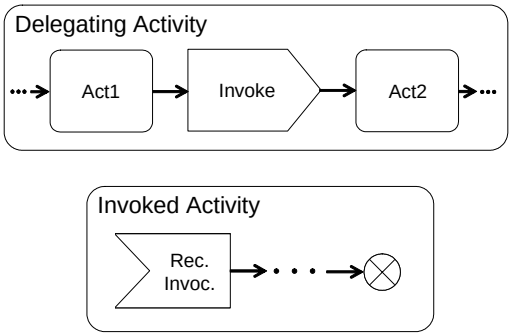
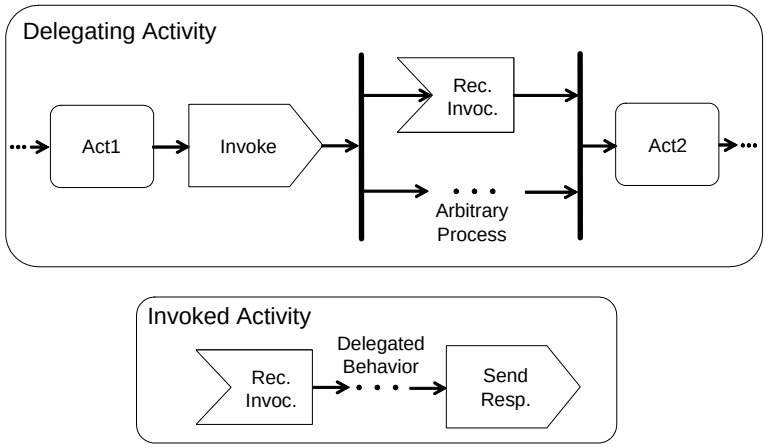


Figure 15 illustrates how the asynchronous invocation of an external Activity and the subsequent processing of a return value can be modeled. The delegating Activity invokes the external Activity. After the invocation the delegating Activity in parallel waits for the return value and continues with arbitrary processing. These two parallel flows are joined once the delegating Activity requires the return value for subsequent calculations. In this case at the latest Act2 requires the return value of the invocation.

Figure 15: **Asynchronous invocation of an external Activity without receipt of a return value**



Optional delegation of functionality to Activities and optional aggregation of Activities can be realized using the variability mechanisms *extensions* or the variability mechanism *adding* Actions.

4.3 Notation for Variability Mechanisms in Activity Diagrams

After having described a set of variability mechanisms for Activity Diagrams, a notation is required for relating the variation points within an Activity Diagram to the corresponding variants and the variability mechanism to apply for binding the variants to their variation points. Moreover, the product features have to be linked to the corresponding variants in order to be able to resolve the variability within the variant-rich process model according to the product features to be regarded by the resulting process model.

In order to highlight product line specific variability in UML Activity Diagrams and to separate this kind of variability from non product line specific variability, the variation points will be marked using the stereotype <<VarPoint>>. Additionally, they dispose of a tagged value with the key “id” that assigns them a unique variation point identification number. The variants belonging to a variation point are included into the Activity Diagram by connecting them to the respective variation point using UML Dependencies as suggested by [Cla01] for generic variability types. A stereotype added to the Dependency relation indicates the variability mechanism to use for binding the variant to the variation point. The variants have a stereotype that links them to the respective feature, an approach also suggested in [RBS00]. An example for the notation of variability in Activity Diagrams is shown in Figure 17. In this case “Action 1” is the variant which can be bound to the variation point represented by a Null-Activity. For binding “Action 1” to the variation point the variability mechanism “Extensions” is used. This is indicated by the respective stereotype assigned to the Dependency relation, connecting the

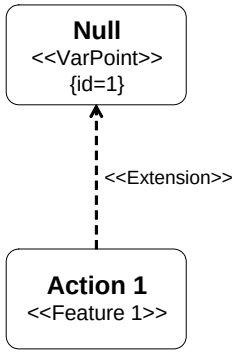
variation point with the variant. Figure 16 lists the stereotypes to be assigned to the Dependency relation for any variability mechanism.

Figure 16: Stereotypes for identification of variability mechanisms in UML diagrams

Variability Mechanism	Stereotype
Information Hiding	<<Implementation>>
Addition of Components	<<Addition>>
Replacement of Components	<<Replacement>>
Omission of Components	<<Omission>>
Parameterization	<<Parameterization>>
Inheritance	<<Inheritance>>
Strategy Pattern	<<StrategyPattern>>
Extension/Extension Points	<<Extension>>

“Action 1” is used to implement “Feature 1” as suggested by its stereotype.

Figure 17: Example for notation of variability in Activity Diagrams

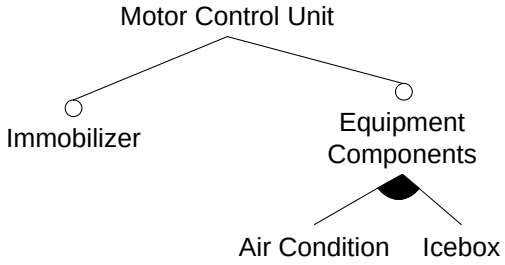


All in all, for the integration of variability mechanisms in UML diagrams only lightweight UML extension mechanisms will be applied in order to be integrateable with small effort into existing UML tools.

4.4 Example

This section gives an example for the configuration of an extract of a motor control unit process family. The FODA [CoN98] feature model in Figure 18 shows that the motor control unit can optionally check an immobilizer and be responsible for the control of the additional equipment components air condition and icebox. For the sake of simplicity it shall here be assumed that the motor control unit is either capable of controlling an icebox or an air condition.

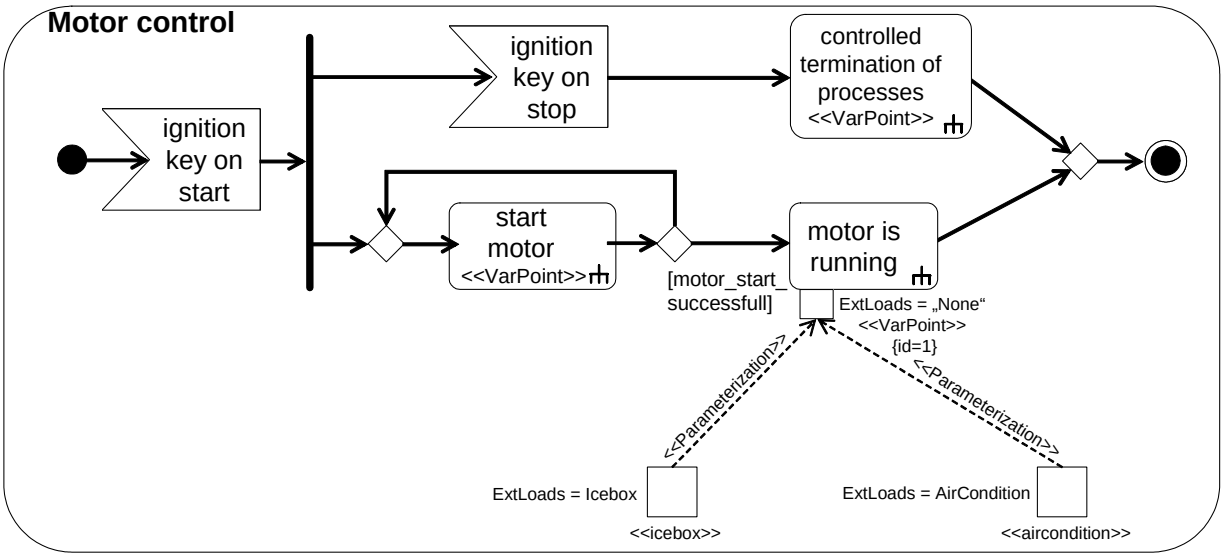
Figure 18: Feature model for motor control unit



4.4.1 Equipment Components

Figure 19 shows the high-level motor control process family. The process has three variation points: “start motor”, “controlled termination of processes”, and the ValuePin “Param_ExtLoads” of the “motor is running” Action. However, only the “Param_ExtLoads” variation point is bound at this level using the variability mechanism *parameterization*. Depending on whether the derived process shall handle an optional icebox and air condition a respective value is set for the ValuPin “ExtLoads”. The default value is “none”.

Figure 19: Motor control unit high-level process



The impact of the configuration on the process is depicted in Figure 20 and Figure 21. Figure 20 shows that the data passed between “monitoring of loads” and “calculation overall load” in Activity “Motor running” depends on its configuration as well as the optional control of the two equipment components (“air condition control” and “icebox control”). The three parameter sets serving as output of “monitoring of loads” and “calculation of overall load” realize the variation of the data flow. The required torque is always forwarded to “calculation overall load”, while the air condition load and icebox load are optional.

Figure 20: "Motor running" subprocess

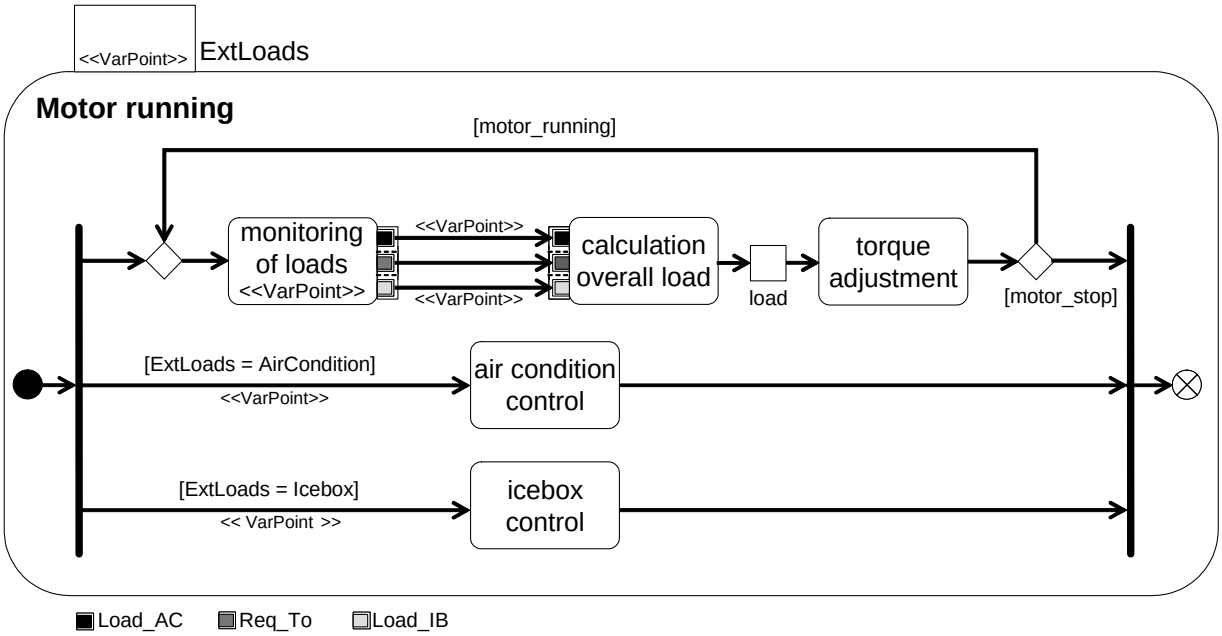
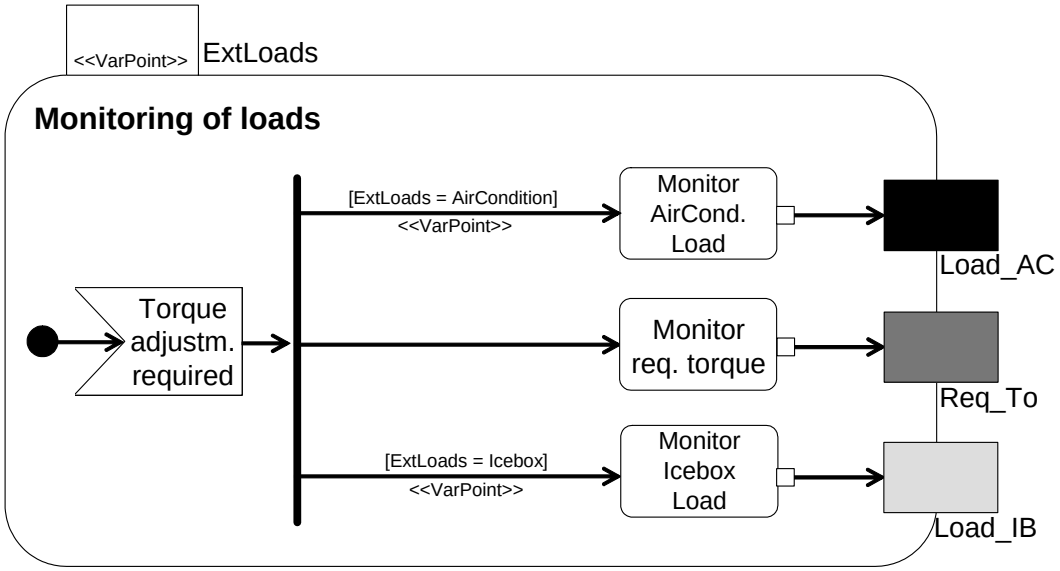


Figure 21 shows the Activity "Monitoring of loads" in detail. Depending on the assignment of the "ExtLoads" ValuePin the "air condition load" and "ice-box load" optionally need to be retrieved in addition to the requested torque depending on the handling of the car by the driver.

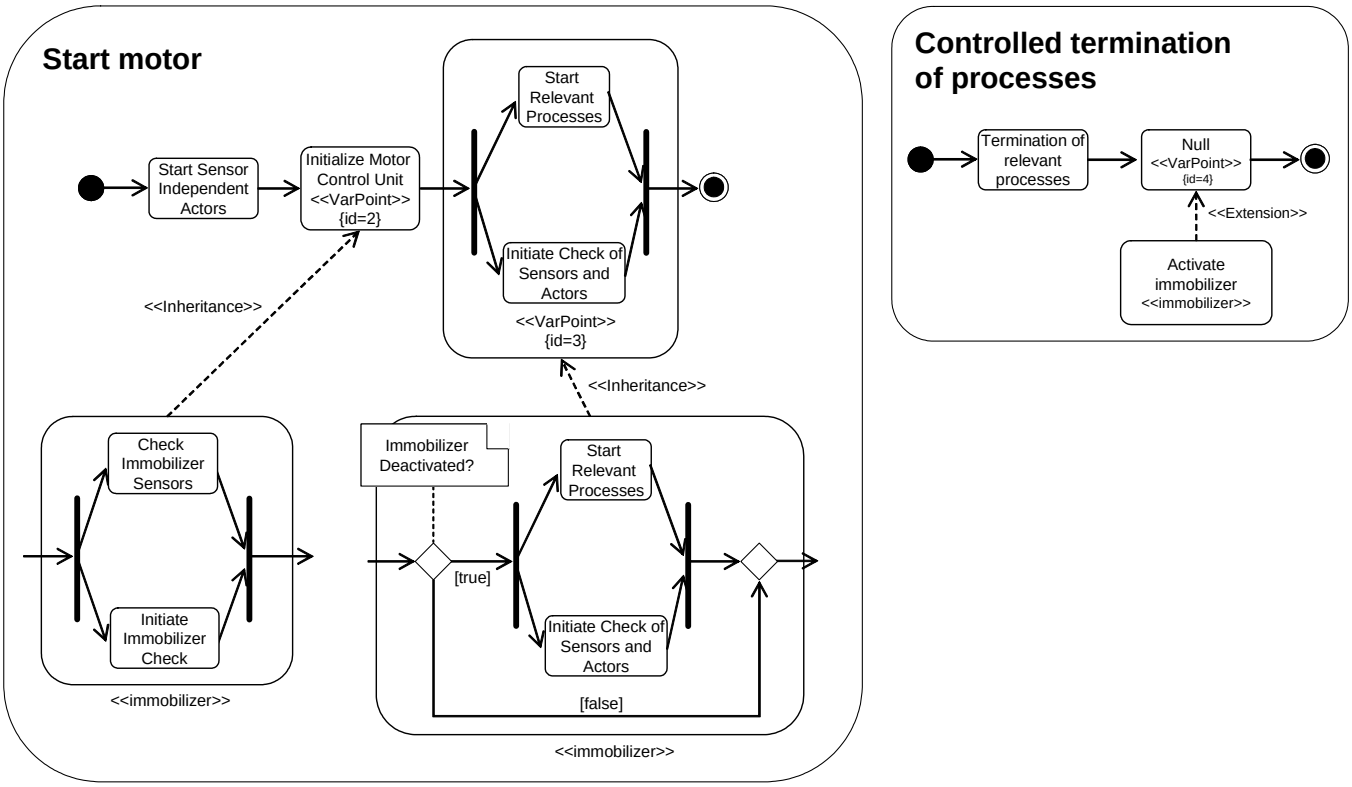
Figure 21: "Monitoring of loads" subprocess



4.4.2 Immobilizer

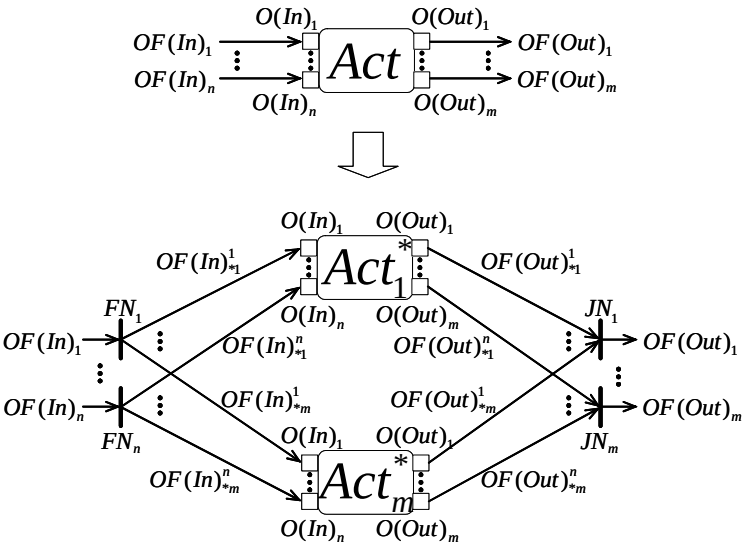
In order to include the processes required for the handling of an immobilizer, the Actions “start motor” and “controlled termination of processes” shown in Figure 22 need to be configured correspondingly. For the optional reactivation of the immobilizer during the shutdown of the motor control unit in “controlled termination of processes”, the variability mechanism *extensions* is applied.

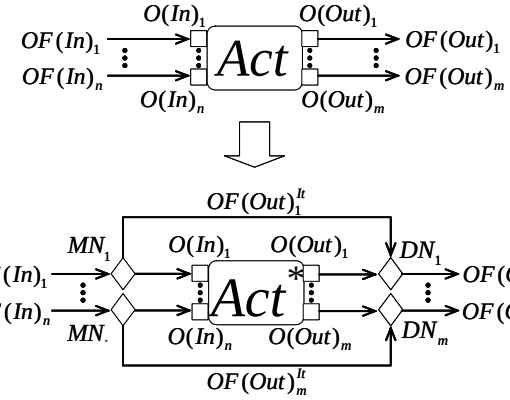
Figure 22: Subprocesses with variability depending on the presence of an immobilizer



For retrieving the appropriate variant of the “start motor” subprocess the variability mechanism *Activity Diagram Inheritance* can be applied using the transformation rules shown in Figure 23.

Figure 23: Activity Diagram Inheritance substitutions applied in Figure 22





5 Variability Mechanisms for UML State Machines

This section analyzes the representation of the variability mechanisms described in 3 for generic processes in UML State Machines.

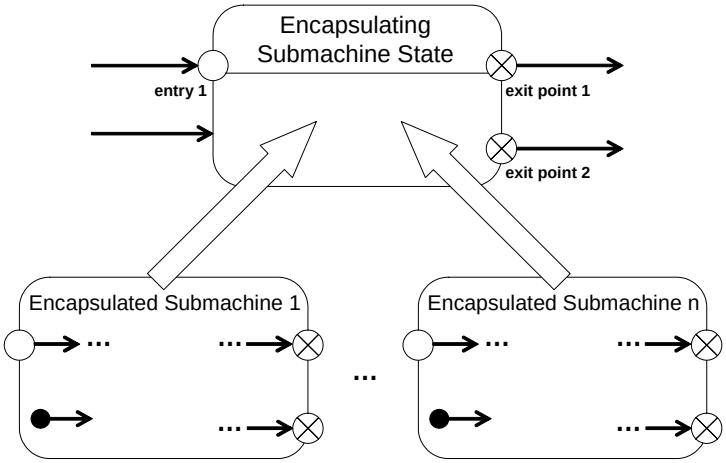
Since UML Activities can be referenced in various parts of an UML State Machine, the application of the variability mechanisms on the Activity representation in State Machines has to be discussed as well.

5.1 Basic Variability Mechanisms

5.1.1 Encapsulation of Varying Subprocesses

In State Machines subprocesses can be encapsulated either in Activities (using CallBehaviorActions) or States (using composite or submachine states). Subprocesses encapsulated in states can best be modeled using submachine states for which varying submachine implementations compliant to the submachine state interface can be inserted. The interface of a submachine state consists of entry and exit states. Additionally, the submachine state may dispose of arcs meeting in and running out of the state's edge. However, arcs meeting in or running out of the state's edge can be neglected since they only lead over to the default starting point of the submachine and are activated if one of the final states of the state machine is reached respectively. On the other hand, deferrableTriggers and information from which state a state has been derived are part of the interface of the state.

Figure 24: Encapsulation of varying subprocesses in State Machines



5.1.2 Addition, Replacement, Omission of Encapsulated Subprocesses

In State Machines encapsulated subprocesses are represented as submachine states referencing a submachine which contains the encapsulated subprocess. Now encapsulated subprocesses can be added, replaced or removed from a State Machine by adding, replacing, or removing the respective submachine states, which contain the subprocess to be added, replaced or omitted. The rules for adding, replacing and omitting encapsulated subprocesses still have to be investigated in detail. Generally speaking, if an encapsulated subprocess shall be replaced by an encapsulated subprocess with a compatible interface, this corresponds to the case described in 5.1.1. Else, if the interface of the replacing subprocess is incompatible, the question rises how the transitions formerly connected to the replaced subprocess shall be connected to the replacing subprocess. If a submachine state is added, according to which rules can it be connected to the remaining process? If a submachine state is omitted, what happens to the adjunctive transitions and their related Activities? Shall the incoming and outgoing transitions be merged? According to which rules? Or shall they be omitted? But this may lead to a disjointed State Machine. The problems occurring during the omission of a subprocess could be avoided by replacing the submachine to be omitted by a “Null Submachine”.

5.1.3 Parameterization

The control flow of State Machines can be parameterized using guards. This can lead to the selection of an outgoing arc at static choice points represented by a junction pseudo state. Here, the routing decision doesn’t depend on the calculation results of a previously executed Activity. In dynamic choice points, on the other hand, the calculation results of the preceding Activity are evaluated before the routing decision is made. These two possibilities are depicted in Figure 25. Since incoming and outgoing transitions of fork and join nodes are not allowed to have guards, optional parallel calculations can only be activated and deactivated by enclosing an optional parallely executable subprocess by choice and merge pseudo states, whose transitions can be parameterized in order to activate or deactivate the optional subprocess. This is shown in Figure 26. Concerning the dataflow, it isn’t represented explicitly in State Machines.

The output of an Activity in a State Machine can be adapted by changing the input parameters of the Activity. Optional Activities carried out during a transition can be activated/deactivated by adding respective guards to their transitions. According to [Gom05] also entry-, exit- and do-Activities can be activated/deactivated by means of guards.

Figure 25: **Parameterization of decisions in State Machines**

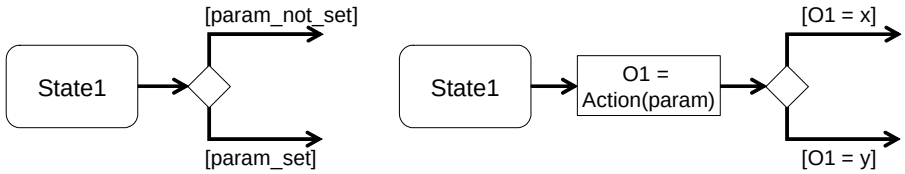
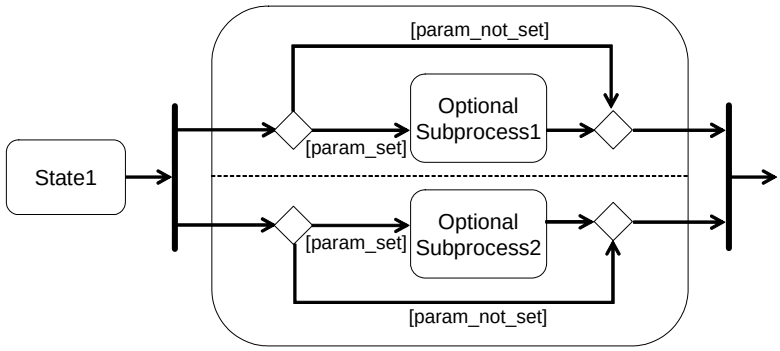


Figure 26: **Activitation/deactiviation of optional parallel processing through parameterization**



5.1.4 Variability in Data Types

In State Machines data types can only be represented as the type of input and output data of an Activity. Theoretically, also in guard-conditions data types can be evaluated.

5.2 Composite Variability Mechanisms

5.2.1 Inheritance

According to [BHK04] the UML specification provides an inheritance mechanism for the derivation of specialized State Machines. While deriving specialized State Machines, simple states can be replaced by decomposed states and orthogonal or decomposed states can be expanded by regions. New transitions and substates can be introduced into an orthogonal state. The submachine implemented by a submachine state can be changed. The new submachine has to have at least the same number of entry- and exit-points as the replaced submachine. Moreover, a transition may be replaced by another transition. The new transition disposes of the same initial state and the same triggers as the replaced transition. Final state, Actions and constraints optionally connected to the transition have to be defined anew.

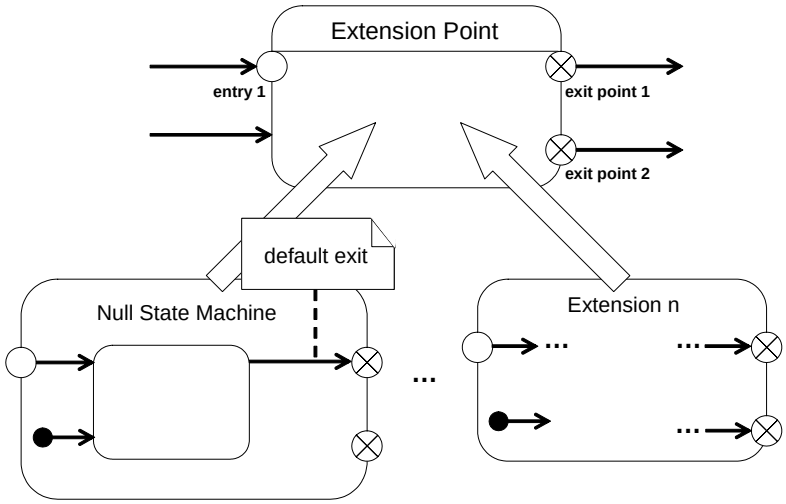
5.2.2 Design Patterns

Design Patterns using encapsulation and inheritance can also be represented in State Machines. The strategy pattern can be represented in State Machines for example using a submachine state referencing an empty submachine. From this empty submachine different variants can be derived by means of State Machine inheritance and inserted instead of the empty submachine. On Activities contained in the State Machine Diagram design patterns can be applied as described in 4.2.2.

5.2.3 Extensions/Extension Points

Similar to Extensions and Extension Points in Activity Diagrams also in State Machines Extensions can be realized by means of subprocess interfaces for which implementing subprocesses containing the extending subprocess can optionally be inserted. The Extensions/Extension Points variability mechanism as defined for Activity Diagrams can also be applied on the Activities/Actions occurring in the State Machine.

Figure 27: Representation of extensions/extension points in State Machines

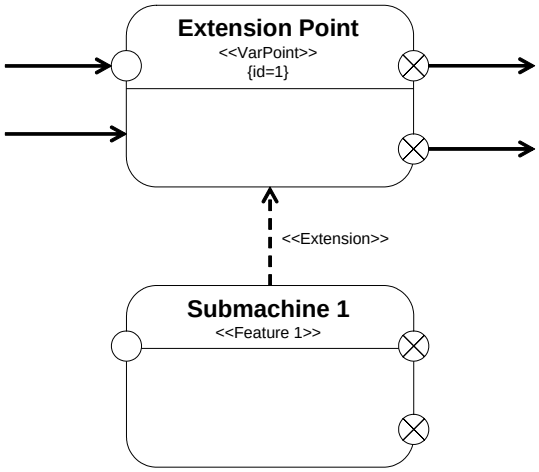


5.3 Notation for Variability Mechanisms in State Machines

For linking variations to their respective variation points in a State Machine, for describing by means of which variability mechanism the variants are integrated into the process and for assigning the variants to the product feature they realize, the same lightweight UML extensions can be used as for Activity Diagrams.

Just as in Activity Diagrams also in State Machines variation points are highlighted using the <<VarPoint>> stereotype having a tagged value “id” containing an unique variation point identifier. Variation points are interlinked with their variants using a Dependency relation, which indicates the variability mechanism to apply by means of an adjunctive stereotype showing the name of the variability mechanism. For identifying a variability mechanism the stereotypes from Figure 16 are used. The feature a variant implements is indicated by a stereotype holding the feature name. An example for the notation of variability in State Machines is shown in Figure 28.

Figure 28: Example for notation of variability in State Machines

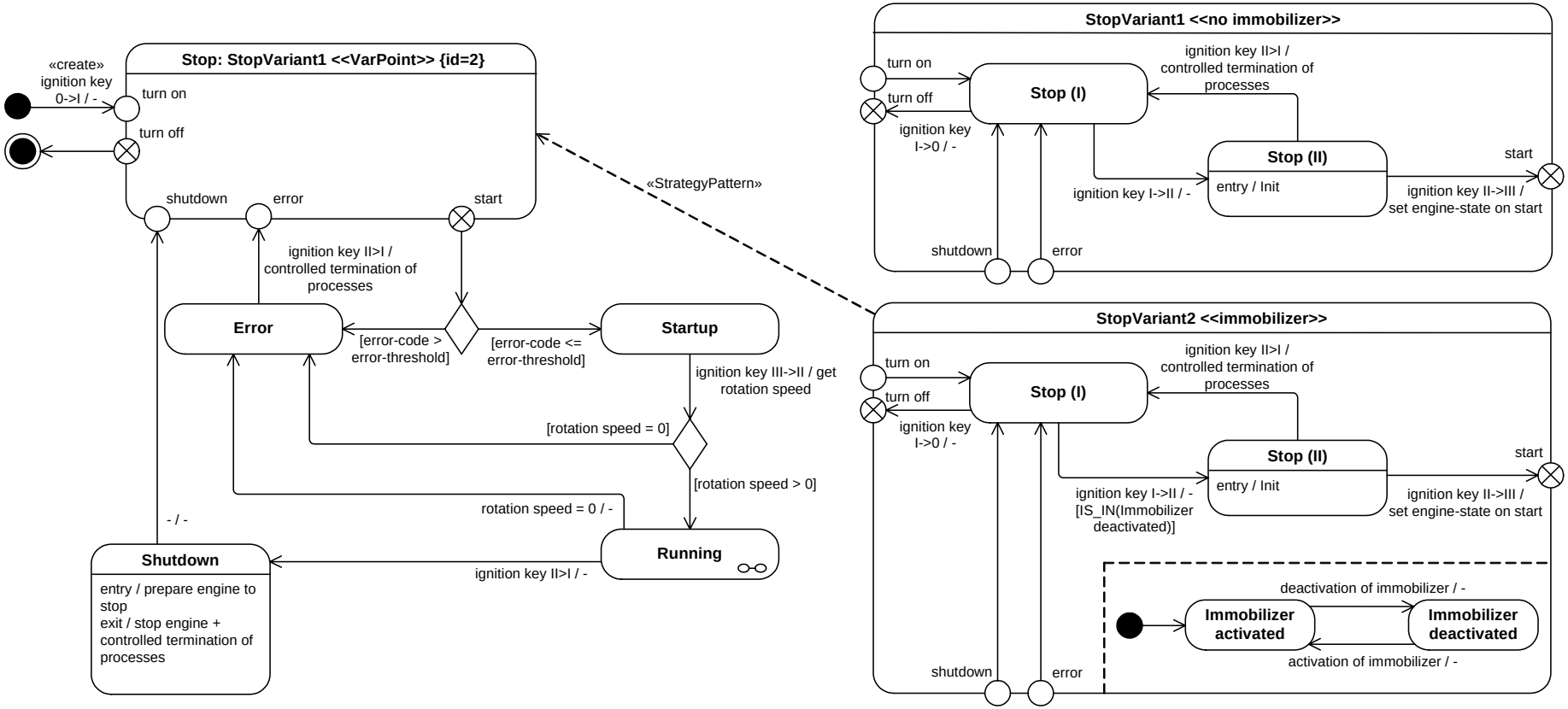


5.4 Example

In this section we give an example for the application of different variability mechanisms for UML State Machines on the basis of the motor control unit process described in [RSW04].

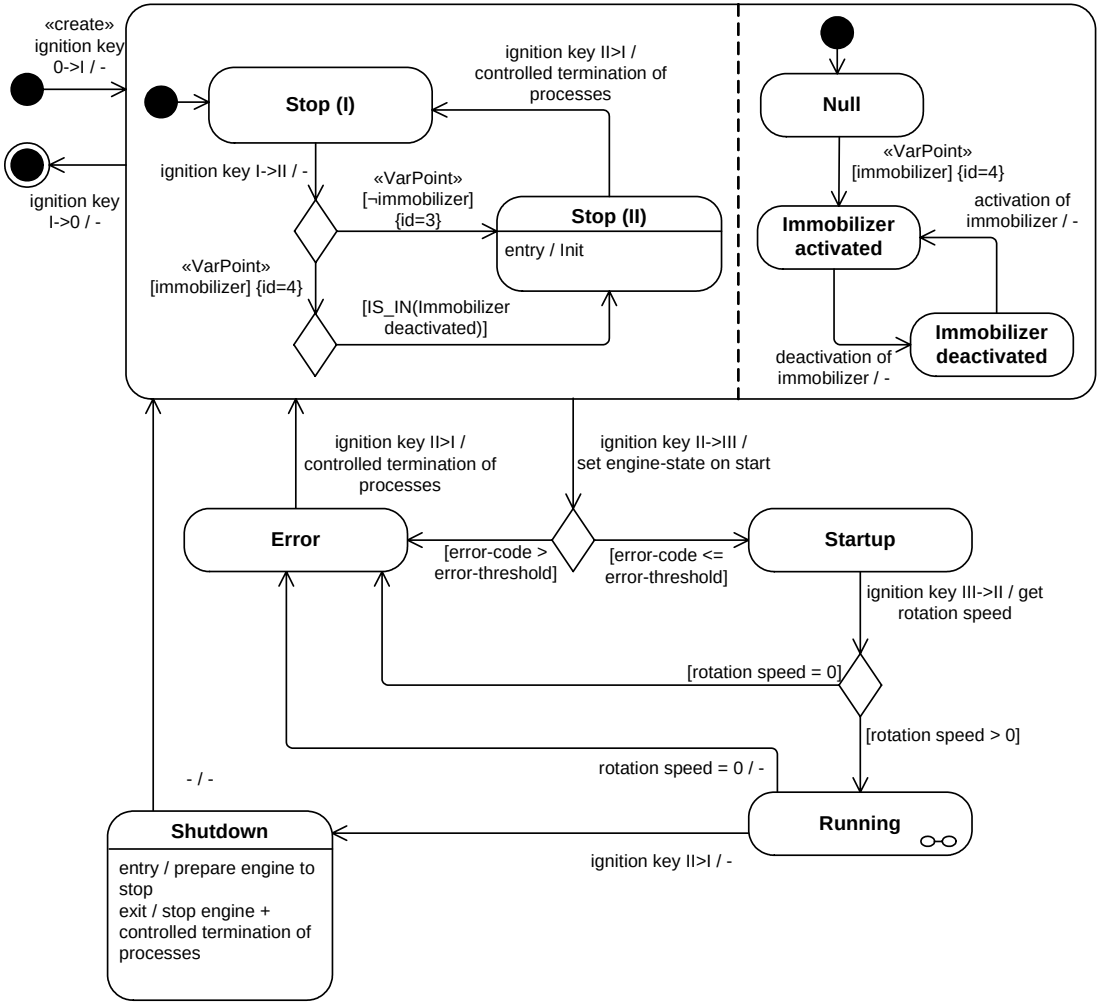
Figure 29 shows the basic version of the motor control unit process without immobilizer. This basic variant encapsulated in the “Stop” submachine state can be replaced by a subprocess variant “Stop-Immobilizer”, which regards the presence of an immobilizer using the Strategy Pattern.

Figure 29: **Basic motor control unit process and immobilizer variant derived using the Strategy Pattern**



The variability can also be represented using Parameterization. This is depicted in Figure 30.

Figure 30: **Representation of the optional immobilizer using parameterization**



6 Variability Mechanisms for BPMN

This section analyzes the representation of the variability mechanisms introduced in chapter 4 in the Business Process Modeling Notation (BPMN).

6.1 Preliminaries

The BPMN places processes inside Business Process Diagrams (BPD). A so-called variant rich business process diagram needs to contain three additions to standard business process diagrams. The first addition is a marking of the places where variability occurs. Second, the possible resolutions should be shown in the diagram. Third, the variability mechanism used to derive the resolution should be shown.

The first requirement, the identification of variation points, can be adapted by the use of annotations in BPMN. However, this approach has some drawbacks. An annotation marking a variation point cannot be distinguished from other annotations in the diagram. Furthermore, the representation of variability in the process models would differ from UML activity diagrams and State Machines, whereas the notations are otherwise more or less congruent. To overcome these limitations, we propose to adapt the concept of a stereotype from the UML2 specification to BPMN. Each activity, association, and artifact can have a stereotype attached. The name of the stereotype is written in italic letters, placed between two angle brackets at each side. It is recommended to place the stereotype above the name of the object if used within an activity or artifact, or beside the association. In sub-processes, the stereotype can be placed before the name of the sub-process. For the purposes of a variant rich business process diagram, the introduction of a stereotype called <<*VarPoint*>> is sufficient. This stereotype can also be expressed graphically as a puzzle-piece like marker at the bottom of an activity. However, if the graphical representation is used, the textual notation has to be omitted.

Furthermore, each variation point is only marked at the level of detail in the diagram where it actually occurs. Those, if a sub-process contains variation points, but is not itself a variation point, it is not marked. The variation points are then only contained in the expanded view.

For an easier understanding of a variant rich business process diagram and the variability mechanisms used, the stereotype variant can be refined with tagged values, as defined according to the UML2 specification. A tagged

value can be written below a stereotype in curly brackets by using the keyword type: {tag=value}. Each stereotype can have two predefined tagged values, feature and type. The feature tag represents the feature the variant belongs to, whereas the type tag further indicates which kind of variability is used. The values of the type tag can be optional, abstract, null, and default. Their semantics will be explained later on.

To save screen as well as paper space, the four types of the stereotype <<*VarPoint*>> , which are represented as tagged values, can also be represented as own stereotypes, thereby specializing <<*VarPoint*>>. The corresponding stereotypes are <<*Optional*>>, <<*Abstract*>>, <*Null*>>, and <<*Default*>>. Furthermore, the tagged values of a stereotype can be omitted in the graphical representation.

Possible resolutions to a variation point are either contained in the graphical representation of the variation point itself, thereby representing the default behavior, as well as by using associations from the variation point to activities or artifacts which are marked as variant.

6.2 Basic Variability Mechanisms

Encapsulation of sub-processes. A BPMN sub-process can hide alternative variant sub-processes behind an invariant interface. Thereby, an interface is defined as the set of input and output events of an activity. The interface activity is marked with the stereotype <<*Abstract*>>. Possible realizations of the interface are connected using associations marked with <<*Implementation*>>.

Figure 31: BPMN interfaces

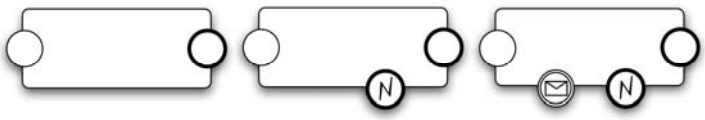


Figure 31 shows the representation of different BPMN interfaces. While it is possible to have more then one incoming or outgoing sequence flow from a sub-process, only one start and end event is required. Multiple incoming sequence flows are and-joined, whereas multiple outgoing flows are and-splited. It is possible to model exceptional as well as several kinds of intermediate events at the edges. However, they do not directly connect to the internal sequence flow and act only as a visual representation for the outer connection of different flows.

Figure 32: Alternative behavior by encapsulation

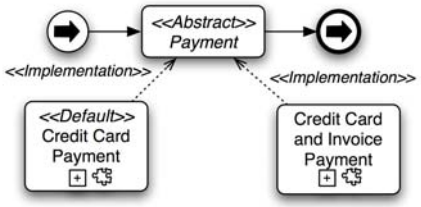


Figure 32 shows how the encapsulation of a sub-process can be used to model alternative behavior. The alternative behavior can occur at a task marked with the <<*Abstract*>> stereotype as well as the name of the variation point, which is “Payment” in the figure. Possible implementations are shown as separate sub-processes, either collapsed or expanded. If there exists a default implementation, it can be marked with the stereotype <<*Default*>>, like the sub-process “Credit Card Payment” in the figure. A directed association ranging from the implementation sub-process to the variation point marks the sub-process as a possible resolution to the variation point. The associations have to be marked with the stereotype <*Implementation*>>. Note the use of the graphical symbol to represent the stereotype <<VarPoint>> at the bottom of the sub-process “Credit Card and Invoice Payment”.

Parameterization. Each BPMN attribute can be parameterized to support optional, alternative, or range variation points. For a graphical representation, the attribute is written beside the element and surrounded with a grouping box. If the connection between the attribute and the element can be misinterpreted, an association should be used. Associations are also used to link variant data objects that contain the possible parameters to the grouping box that surrounds the attribute. The association is marked with the stereotype <<*Parameterization*>>.

Figure 33: Range/Value/Expression Parameterization

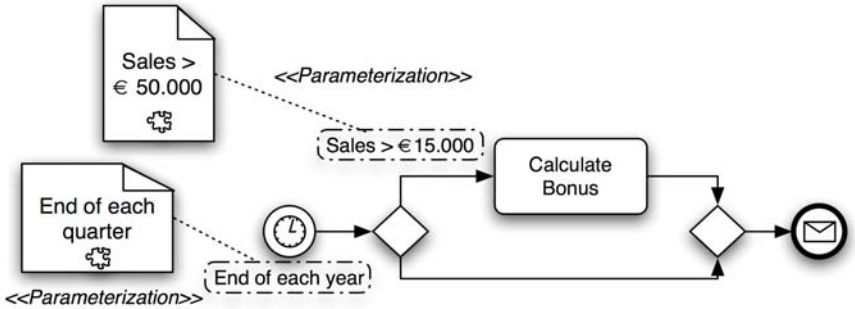


Figure 33 shows the parameterization of two different attributes. The upper one parameterizes the *ConditionExpression* attribute of a sequence flow. The default value is a guard that activates the sequence flow if the sales are

greater then €15.000. An alternative parameterization changes the attribute to activate the sequence flow if the sales are greater then €50.000. The lower one offers an alternative for the *TimeDate* attribute of the intermediate timer event. The default behavior triggers the event at the end of each year, whereas the alternative behavior triggers the event at the end of each quarter.

6.3 Composite Variability Mechanisms

Inheritance. Inheritance modifies an existing (default) sub-process by adding or removing elements regarding to specific rules. This allows for realizing alternative variation points. An association represents inheritance from the child activity to the parent activity when it is marked with the stereotype <<*Inheritance*>>.

Figure 34: Alternative behavior by inheritance (collapsed example)

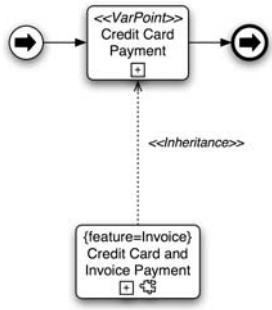


Figure 34 shows alternative behavior by the use of inheritance. The default sub-process is shown at the top of the figure, placed between the sequence flows. It is marked with the <<*VarPoint*>> stereotype. Optionally, the <<*Default*>> stereotype could be used, but the placement of the sub-process between sequence flows already marks the default status. The alternative is realized by inheritance, which is indicated by the <<*Inheritance*>> stereotype at the association between the two sub-processes. The specialized sub-process “Credit Card and Invoice Payment” belongs to the feature “Invoice” as annotated with the tagged value “feature”. The stereotype <<*VarPoint*>> is shown as a graphical marker (the puzzle piece like symbol at the bottom of the sub-process).

Figure 35: Alternative behavior by inheritance (expanded example)

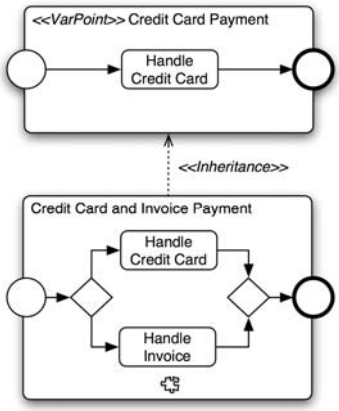


Figure 35 shows the expanded sub-processes of **Figure 34**. It can be seen that the task “Handle Credit Card” is re-used in the specialization “Credit Card and Invoice Payment”. However, there are currently no rules of how to derive a correct specialization regarding to formal criteria.

Extension Points. Extension points use a combination of encapsulation and “null sub-processes” to realize optional variation points. An extension point activity is marked with the stereotype <<*Null*>*.* Associations marked with <<*Extension*>> connect optional implementations. If there is only one optional variant, it can be shown instead of the null activity, marked with a <<*Optional*>> stereotype.

Figure 36: Optional behavior by Null activities

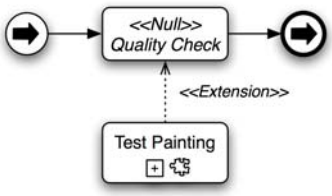


Figure 36 uses extension points to realize optional behavior. The optional extension point “Quality Check” is marked with the <<*Null*>> stereotype. Possible resolutions are attached with associations labeled with an <<*Extension*>> stereotype. **Figure 36** contains one optional resolution of the variation point, called “Test Painting”.

Figure 37: Simple optional behavior

If there is only one optional resolution of a variation point, it can be marked with the stereotype <<*Optional*>> and directly placed between the sequence flows, without the use of a <<*Null*>> task (**Figure 37**).

Design Patterns. The concepts of encapsulation and inheritance can be used to implement design patterns that describe variability. There are no additional graphical notations required; the patterns can be formed by the use of the above mentioned concepts.

Figure 38: Alternative behavior by encapsulation and inheritance

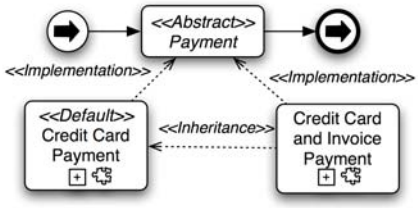


Figure 38 implements the strategy design pattern. It is derived from **Figure 32** with an additional inheritance relation between “Credit Card and Invoice Payment” and “Credit Card Payment”.

6.4 Example

This section gives an example for the application of different variability mechanisms for business process diagrams. The example extends the e-business shop introduced in the PESOA technical report 8, appendix A [Puh04] by directly expressing the variability resulting from the features of the e-business shop.

Figure 39: High-level processes of the e-business shop example

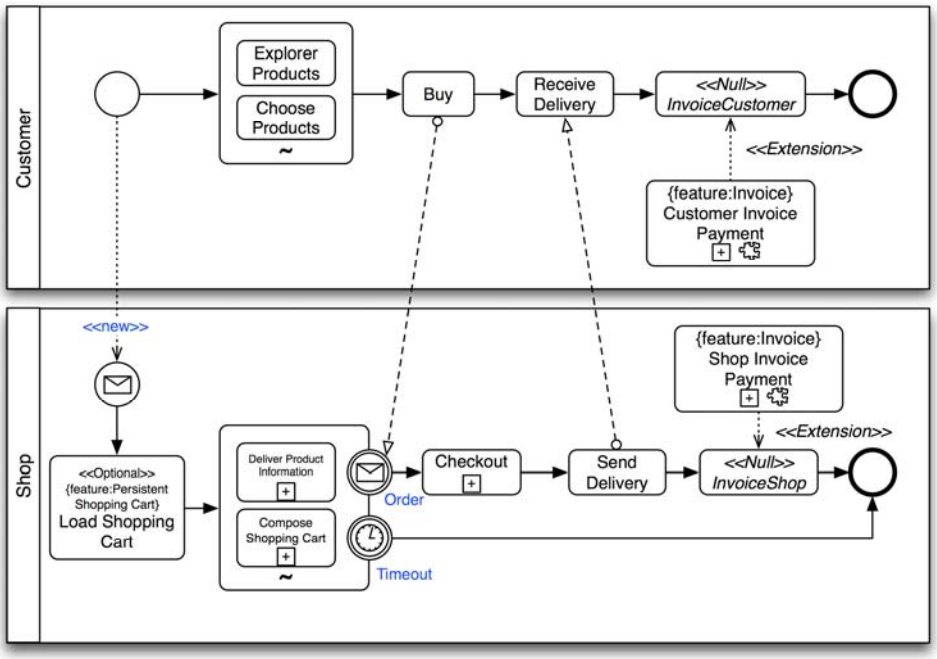


Figure 39 contains the variant rich high-level workflow of the e-business shop. The variant elements, which have been marked with color in [Puh04] have been realized by different variability mechanisms. The Customer's pool contains the optional behavior "InvoiceCustomer", which is represented by a null activity. If the feature invoice is selected, the sub-process "Customer Invoice Payment" is included at the extension point. Note that the original model of the variant covered a gateway as well; this has been placed inside the sub-process. The Shop's pool has the optional task "Load Shopping Cart" which is included if the feature "persistent shopping cart" is selected. The null activity "InvoiceShop" is filled with "Shop Invoice Payment" if the invoice feature is selected. The task "Shop Invoice Payment" corresponds to the "Customer Invoice Payment". As both variation points are enabled by the same feature, their realizations always appear together.

Figure 40: Expanded sub-process "Customer Invoice Payment"

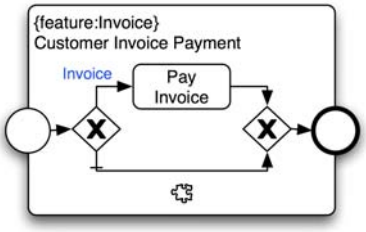


Figure 40 contains the expanded sub-process “Customer Invoice Payment”. The decision if the payment includes an invoice is evaluated inside the sub-process as it is also part of the variation point.

Figure 41: Expanded sub-process “Shop Invoice Payment”

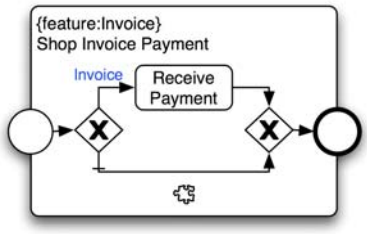


Figure 41 contains the expanded sub-process “Shop Invoice Payment”, which equals Figure 40 as a counterpart for the Customer.

Figure 42: Expanded sub-process “Deliver Product Information”

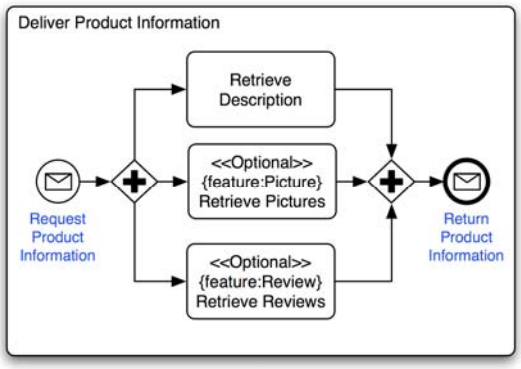


Figure 42 contains the expanded sub-process “Deliver Product Information”. It also uses the <<*Optional*>> stereotypes to mark “Retrieve Pictures” and “Retrieve Reviews” as variation points with one possible resolution.

Figure 43: Expanded sub-process “Compose Shopping Cart”

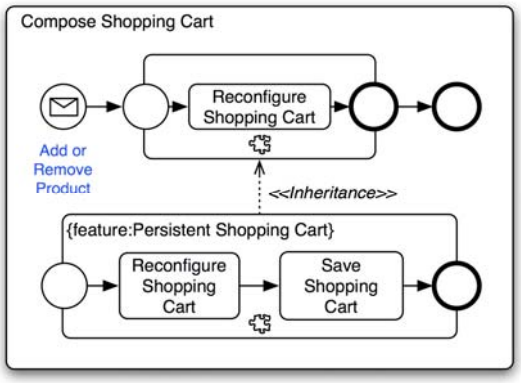


Figure 43 contains the expanded sub-process “Compose Shopping Cart”. It utilizes inheritance to modify the default behavior by inserting an additional task “Save Shopping Cart” after the reconfiguration of the shopping cart has taken place.

Figure 44: Expanded sub-process “Checkout”

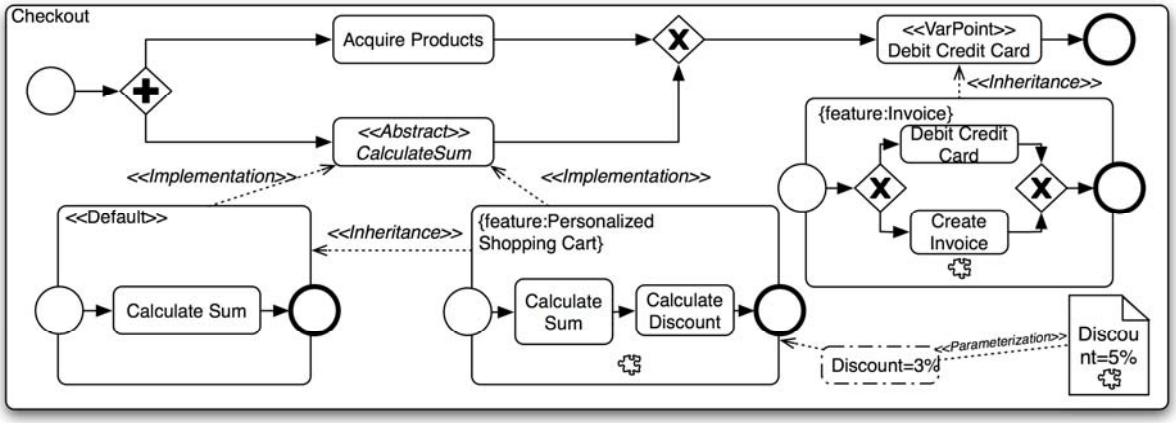


Figure 44 expands the sub-process “Checkout”. It uses the concept of design patterns to describe the possible resolutions to the alternative variation point “CalculateSum”. The first resolution implements the default behavior, it only calculates the sum. The second, alternative resolution specializes the default one by using inheritance to add the additional calculation of a discount. The percentage of the discount is parameterized with a default value of 3 The task “Debit Credit Card” has also an alternative implementation derived by the use of inheritance.

7 Modeling Variability in Matlab/Simulink

The automotive domain is characterized by a variety of product variants based on electronic systems. This is only enabled through an increasing usage of embedded software [3]. Significant here are concepts for variant configuration of the embedded software. This comprises

- concepts for modeling variability in architecture models or directly in the source code, and
- concepts for the management and the explicit presentation of variability (e.g., using feature models).

Only the integration of both concepts enables a (partial) automation of the configuration process, i.e., resolving variability in order to be able to instantiate valid product variants.

Due to the increasing influence of model-based development of automotive embedded software, concepts for modeling variability in software architecture models have become of particular interest. An important exponent for model-based software development is Matlab/Simulink [2]. Modeling a software architecture that contains the variability of all product variants leads to a generic architecture model in Matlab/Simulink. By means of the configuration knowledge, the model of a concrete product variant can be extracted from both the management of variability within the feature model and the generic architecture model. Afterwards, source code can be (auto-)generated out of the model of a concrete product variant using Matlab.

Our proceeding is geared towards the generative domain model [1]. This model separates application-oriented concepts within the feature model from implementation concepts – in our case the architecture models. Features of the domain of interest are managed and structured in a hierarchy within the feature model. The feature model itself contains common and variable features[1] and their dependencies. For our implementation, we have modeled the feature model outside Matlab/Simulink [4].

The following aspects are considered when mapping features out of the feature model on variability within the generic architecture model:

- Concepts for identifying variability within the generic architecture model of the software (variation points)

[1] Although not necessary in our contect, we regard common features as part of the feature model, too, in order to include all features of the product family.

- Concepts for modeling variants (assigning variants to variation points)

This chapter describes alternatives for implementing these concepts in generic Simulink architecture models. These concepts are the basis for a (partial) automation of variant configuration in the scope of model-based development of embedded software with Matlab/Simulink.

7.1 Concepts for Modeling Variants

Basically, Matlab/Simulink provides two concepts for modeling variants in architecture models:

- Substitution of whole Simulink blocks regarding the occurrence of (local) variability in the context of *configurable subsystem block*s – comparable to the variability mechanism Inheritance.

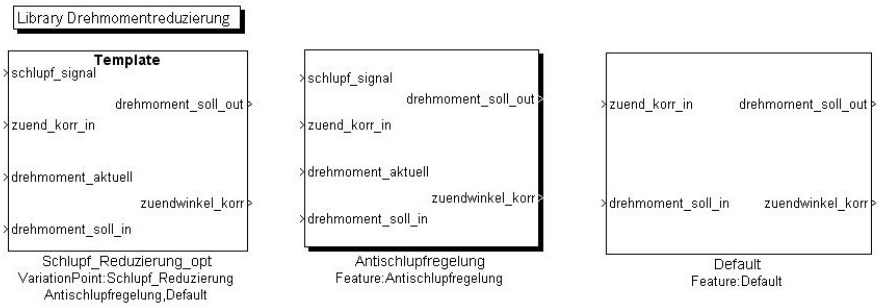
- Application of (block) parameters – comparable to the variability mechanism Parameterization

Both concepts are intrinsic to Matlab/Simulink. In the following, both concepts are explained by means of examples. In particular, we will dwell on the mapping of feature types out of the feature model [1] on concepts for modeling variants in Simulink.

7.1.1 Configurable Subsystem Blocks

A concept for describing variability includes the selection of alternative block variants. These block variants are added to a *library* and assigned as a *member* to a template block, the *configurable subsystem block*. Template blocks enable selection of a variant from different alternatives in the generic model. Now parts of the model can be designed to be exchangeable. Yet, it is not necessary for exchangeable blocks to have the same in- and out-signals or ports. Unused signals are terminated or grounded from Simulink automatically and are not considered during code generation should source code optimization become necessary. By using *stateflow diagrams* in combination with *configurable subsystem blocks,* specific issues regarding trigger signals have to be taken into account, i.e., handling them as normal in-port signals.

Fig. 45. Example of a Configurable Subsystem Block[2]



The alternative choice can also be employed to model optional variants. Admittedly, a default block has to be defined. This block is used, if the optional element is selected. Generally, the default block is modeled as an empty block (interrupting, terminating, and grounding signals) or as a block that pipes the signals (directly from in-ports to out-ports). If applicable, a combination of both may be used. In Figure 45 the template block Schlupf_Reduzierung is a variation point. It can be replaced by one of the member blocks Antischlupfregelung or Default, representing the optional feature Antischlupfregelung. In our figure Antischlupfregelung has been selected (depicted by the black angle around the block)

Combining *configurable subsystem blocks* also allows the implementation of a multi-choice (or assignment) – although only in a complex form. In this case, all signals from all *member* block combinations have to be available at the *template* block from the start. However, alternatives and options are more relevant for modeling variability in data and control flow, represented in Simulink.

7.1.2 Application of (Block) Parameters

Parameterization is useful for variable scaling in data transformations within function blocks. Typically, parameterization is used for linking variant-specific characteristic curves or scaling of functions. The Simulink block library itself contains a number of such blocks, which are already parameterized. These blocks can be customized for solving specific problems of a selected application domain. Examples are one- or multidimensional *lookup-table* blocks, parameterized by one- or multidimensional arrays. In particular, these blocks are commonly employed in combination with characteristic curve fields in the engine control unit. Basing on this pattern, developers can define any generic parameterizable blocks.

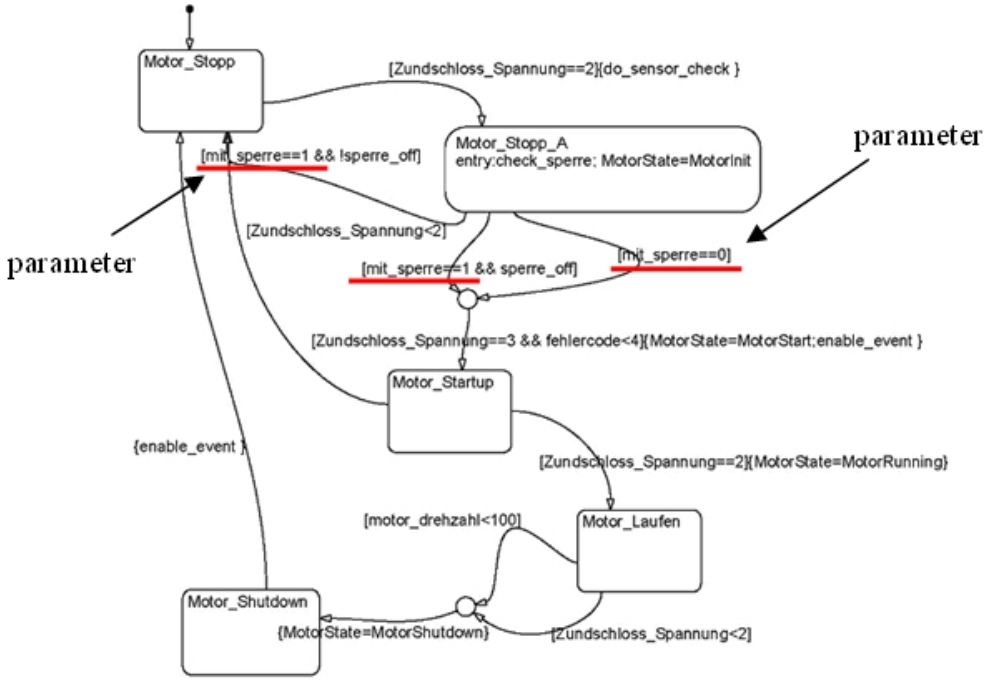
[2] The blocks Antischlupfregelung and Default are assigned to the template block as *members*. When using *configurable subsystem blocks,* one of the *member* blocks can be chosen in the model.

Parameters can also be used to specify branching in the control flow (and, thus, in *stateflow diagrams*) statically, i.e., to manage variant-specific branching. This is important, as it is not possible to exchange subelements such as *States*, *Subcharts*, or *Transitions* in stateflow diagrams on the Simulink level.

With variability in behavior, i.e., within *state charts*, two options only arise:

- Exchange the whole chart (embedded in a *configurable subsystem-block*)
- Design a chart that contains all possible variants. Afterwards, this chart can be constrained statically for a specific variant using parameters (figure 46).

Fig. 46. State Chart with Parameter-based Variability Mechanism



7.2 Concepts for Identifying Variability

Template blocks (*configurable subsystem blocks*) and parameterized blocks (parameterized *masked blocks*) are variation points and have their implicit or

explicit counterparts in the feature model[3]. In Simulink, these blocks have to be flagged in order to indicate this relation.

Our recommendation here is to add a tag to the block properties of *configurable subsystem blocks* with the notation *VariationPoint::Featurename*. The notation *Featurename* corresponds to the name of the variation point in the feature model. This enables a simple mapping of variability in the feature model with the generic Simulink model. Then, *model construction commands* [2] can be employed to pursue and visualize this mapping. It has proven useful to add additional information to the tags of variation points – such as the mode for realizing variability. This additional information is not mandatory, but it facilitates configuring the models.

For parameterized blocks, our recommendation is to use the tags *VariationPointIB:: Parametername* and *VariationPointIW::Parametername*. This differentiation depicts where parameters are set: directly at the parameterized blocks (i.e. *VariationPointIB*) or in the model's Matlab workspace (i.e. *VariationPointIW*).

When using *configurable subsystem blocks*, another convention for describing alternatives or options is to add a tag to the *member* blocks with the name of the feature that is represented by the *member* blocks. This could be done in the form of *Feature::Featurename*, where *Featurename* is the name of the feature.

[3] Not all features of feature diagrams are variation points. Features that are classified as mandatory and do not have variable subfeatures do not require a counterpart in the Simulink model. They are only modelled for the sake of completeness of the product family.

8 Conclusions and Outlook

In this report we have given an overview of variability mechanisms commonly applied in product family engineering, thereby identifying which of these variability mechanisms are architecturally relevant and thus should be represented in architecture models.

Next, we have analyzed the transferability of these architecturally relevant variability mechanisms on a basic process model and their representation in UML Activity Diagrams, UML State Machines, BPMN, and Matlab/Simulink – an important exponent for model-based software development in Automotive industry. Moreover, we have shown how to depict variation points, process variants and variability mechanisms in UML Activity and State Machine diagrams using only lightweight UML extension mechanisms; the same is true of variability mechanisms in Matlab/Simulink, too. As the BPMN does not contain official extension mechanisms, we adopted the lightweight stereotype approach from the UML. This representation allows for the integration of variant-rich processes into the product family engineering development process.

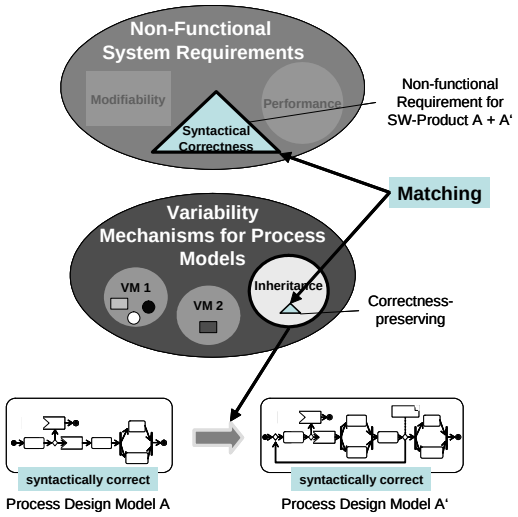
In the next step of our work we will analyze by means of which techniques the variability mechanisms for process models can be realized on the source code level. This will allow for their automatic implementation in later phases of the software development process, thus allowing for a more model driven software development. This is also important for the next PESOA phase.

Concerning the integration with earlier phases of the software development the long-term goal of our research is to categorize the variability mechanism for process models according to the relevant non-functional characteristics of their modification, like the maintenance of the syntactical correctness of a process or the modifiability of the process according to respective metrics [ReV04]. The idea is that using the right variability mechanisms the requirements of a system, which are realized by a corresponding system design [IEE98], can be maintained while deriving process design variants for similar software products by means of the suitable variability mechanisms.

Figure 45 visualizes the above mentioned ideas. The upper ellipse shows possible non-functional requirements for software products. Syntactical correctness is the non-functional requirement relevant for software product A and A', whose process design model shall be derived from the process design model of software product A. Therefore, syntactical correctness is highlighted. The lower ellipse contains various variability mechanisms with different non-functional properties they preserve if being used for the derivation of process variants. An inheritance mechanism shall, for example, have the

property to be correctness-preserving. Therefore, it is selected to derive a process design model for software product A' from software product A as shown in the lowest part of the figure. This is possible, since inheritance shall be correctness-preserving and because the process of A shall also be – according to the requirements of A – syntactically correct.

Figure 45: **Requirement driven derivation of process variants by means of appropriate variability mechanisms**



At first, we will focus on the syntactical correctness as a non-functional requirement which is essential for the design of a process.

References

[ABB01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. Component-Based Product Line Engineering with UML. The Component Software Series. Addison-Wesley Publishing Company, 2001.

[AnG00] M. Anastasopoulos and C. Gacek. Implementing product line variabilities. IESE Report No. 089.00/E, November 2000.

[BaB01] F. Bachmann and L. Bass. Managing Variability in Software Architectures. Proc. Symp. Software Reusability: Putting Software Reuse in Context, , ACM Press, New York, pp. 126-132, 2001.

[BBG05] J. Bayer, W. Buhl, C. Giese, T. Lehner, A. Ocampo, F. Puhlmann, E. Richter, A. Schnieders, J. Weiland. Process Family Engineering: Modeling variant-rich processes. PESOA Report No. 18/2005, DaimlerChrysler Research and Technology, Delta Software Technology, Fraunhofer IESE, Hasso-Plattner-Institute, 2005.

[BFK05] J. Bayer, T. Forster, S. Kiebusch, T. Lehner, A. Ocampo, J. Weiland. Merkmal- und Entscheidungsmodellierung. PESOA Report No. 21/2005, DaimlerChrysler Research and Technology, Fraunhofer IESE, University of Leipzig, 2005.

[BHK04] Born, M., Holz, E., Kath, O.: *Softwareentwicklung mit UML 2*. München: Addison-Wesley 2004

[BJM96] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl. Pattern-Oriented Software Architecture – A System of Patterns. John Wiley & Sons, 1996.

[Bos00] Jan Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Harlow, England et al., 2000.

[CE00] K. Czarnecki, U. Eisenecker, Generative Programming – Methods, Tools, and Applications, Addison-Wesley, Boston, MA, 2000

[Cla01] M. Clauß. Untersuchung der Modellierung von Variabilität in UML. Diplomarbeit, Technische Universität Dresden, Fakultät Informatik, Juli 2001.

[ClN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Upper Saddle River, NJ 07458, 2002.

[CoN98] S. Cohen, and L. Northrop. Object-Oriented Technology and Domain Analysis. In Proceedings of the Fifth International Conference on Software Reuse, 1998

[GBS01] J. van Gurp, J. Bosch, M. Svahnberg. On the Notion of Variability in Software Product Lines. Proceeedings of WICSA 2001, August 2001.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[Gom04] H. Gomaa, D. Webber. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In Proceedings of the 37th Annual Hawaii International Conference on System Sciences, HICSS`04, pp. 1-10, IEEE Computer Society Press, January 2004.

[Gom05] H. Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2005.

[IEE98] Software Engineering Standards Committee of the IEEE Computer Society. IEEE Recommended Practice for Software Design Descriptions, Std 1016-1998

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Longman, Harlow, England et al., 1997.

[KeM99] B. Keepence, M. Mannion. Using Patterns to Model Variability in Product Families. In IEEE Software, pp 102-108, July/August 1999.

[MW04] The MathWorks, Simulink – Simulation and Model-based Design, Natick, MA, 2004

[OMG03] OMG: Unified Modeling Language: superstructure. Version 2.0. 2003. Internet-URL: http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

[Pen03] T. Pender. UML Bible. Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[Puh04] F. Puhlmann. Modeling Workflows in the E-Business Domain. PESOA Report No. 08/2004, Hasso-Plattner-Institute, 2004.

[RBS00] M. Riebisch, K. Böllert, D. Streitferdt, B. Franczyk. Extending the UML to Model System Families. Integrated Design and Process Technology (IDPT-2000), 2000.

[ReV04] H.A. Reijers and I.T.P.Vanderfeesten. Cohesion and Coupling Metrics for Workflow Process Design. In J. Desel, B. Pernici and M. Weske, editors, Proceedings of the 2nd International Conference on Business Process Management (BPM 2004), Lecture Notes in Computer Science 3080, 290-305. Springer Verlag, Berlin, 2004.

[RSW04] E. Richter, A. Schnieders, J. Weiland. Prozessanalyse und –modellierung in der Domäne Automotive. PESOA Report No. 07/2004, DaimlerChrysler Research and Technology, Hasso-Plattner-Institute, 2004.

[Sch97] H. A. Schmid. Systematic Framework Design by Generalization. Communications of the ACM, 40(10):48 - 51, Oktober 1997.

[ScP05] A. Schnieders, F. Puhlmann. Activity Diagram Inheritance. In Proceedings of the 8th International Conference on Business Information Systems BIS, Poznan, Poland,
April 20-22 2005

[SGB03] M. Svahnberg, J. van Gurp, J. Bosch. A taxonomy of variable realization techniques (DRAFT). Submitted November 2003, accepted for publication in Software Practice & Experience.

[SvB03] Mikael Svahnberg and Jan Bosch. Issues Concerning Variability in Software Product Lines, volume June of 146. Lecture Notes in Computer Science, 2003.

[SZ03] Jörg Schäuffele, Thomas Zurawka, Automotive Software Engineering – Grundlagen, Prozesse, Methoden und Werkzeuge, Vieweg, Wiesbaden, Juli 2003

[WR05] J. Weiland, E. Richter, Konfigurationsmanagement variantenreicher Simulink Modelle, Tagungsband GI-Jahrestagung, Bonn, September 2005 (to appear)