Modeling and Implementing Variability in State Machine Based Process Family Architectures for Automotive Systems

Arnd Schnieders
Hasso-Plattner-Institute for IT Systems Engineering at the University of Potsdam,
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam, Germany
ardn.schnieders@hpi.uni-potsdam.de

ABSTRACT
In this paper we present an approach for modeling UML State Machine based process family architectures, which is part of our work to transfer Product Family Engineering techniques to process oriented software like automotive systems. For modeling UML State Machine based architectures of automotive system families we introduce a set of variability mechanisms. These allow for taking the different properties of variability realization techniques into account during the development of a process family architecture. We also touch some variability implementation issues in C/C++ and provide a lightweight UML extension for the introduction of variability mechanisms into State Machines.

Categories and Subject Descriptors: D.2.11 [Software]: Software Architectures

General Terms: Design

Keywords: Process Family Engineering, Automotive, UML 2.0 State Machines, Product Family Architecture, Variability Mechanisms

1. INTRODUCTION
Nowadays, in the automotive market 90 percent of the real innovation in cars relates to the electronics of a car and the share in the creation of value constitutes already 30 percent with an increasing tendency [27]. Thereby, 50 percent of the development costs for a control unit account for the software, whose size doubles every 2-3 years. Now, automotive software frequently has to be developed in many different variants, which allows for a better market penetration regarding diverging customer needs and financial power of the customers. There are for example engine control units optionally implementing speed control, electronic stability programs, etc. On one hand due to the increasing number of product features the software systems and their variants become more and more complex and manifold, but on the other hand the competitiveness of a company highly depends on their ability to bring high-quality product variants rapidly to market. In the long run this challenge can only be faced by advanced software development techniques like product family engineering. However, up to now product family engineering research has concentrated on software systems, where static diagrams like class diagrams or component diagrams represent the main blueprint for the development of the software system while process oriented software, i.e. software where processes represent the central design artifact, has not been regarded adequately. As a result, existing product family engineering approaches are not suited well for process oriented software like embedded automotive systems. The intention of our work is to contribute in closing this gap by supporting the investigation of product family engineering of process oriented software, in short process family engineering. Thereby, we concentrate on the UML State Machine [16, 22] based representation of process family architectures following a new variability mechanism centric approach. According to a case study [20] performed in cooperation with DaimlerChrysler a combination of UML 2.0 State Machines and Activity Diagrams are most suitable for modeling software for embedded automotive control units. With our variability mechanism centric approach we want to show a way how to make variability implementation related decisions during architecture development and thus to support a more model-driven variability implementation in product family engineering. Variability mechanisms represent techniques for realizing the variability within a system and therefore play a prominent role in software product family engineering. Moreover, with our approach we want to provide a technique for modeling effectively and intuitively variability in process family architectures and thus to support the reuse of process family architecture parts within the process family. Finally, we aim at a variability modeling approach, which can be integrated into existing UML tools without major effort.

This paper is structured as follows: In section 2 we give a brief introduction to process family engineering and process family architectures. In section 3 we define a set of variability mechanisms for State Machine based process family architectures and examples for their implementation in C/C++. In section 4 we give an example for modeling a State Machine based process family architecture. In section 5 we give an overview of related work. Section 6 summarizes
the contents of this paper and gives an outlook to future research.

2. PRELIMINARIES

In this section we give a brief introduction to Process Family Engineering and Process Family Architectures.

2.1 Process Family Engineering

Product family engineering is a paradigm to develop software applications using a set of software subsystems and interfaces that form a common structure based on which derivative products tailored to individual customer needs can be efficiently developed according to [15]. Another important aspect is that within a software product family reuse isn’t restricted to the reuse of implementation artifacts but is expanded to any development artifact (like e.g. requirement or design models).

In contrast to a single-system development process product family engineering is characterized by a so called dual lifecycle [28] as indicated in figure [10]. In order to emphasize that our work focuses on the development of process-oriented software, we use the term process family engineering instead of product family engineering and process family infrastructure instead of product family infrastructure. However, the basic development process is the same for product family engineering as for process family engineering. In the first section of the process family development process (called process family engineering) generic development artifacts (called the process family infrastructure) are developed based on which process family members are derived efficiently in the corresponding phase within the second section (called application engineering) of the process family engineering process.

Figure 1: Process Family Engineering Process

2.2 Process Family Architectures

During the design of a process family a process family architecture (PFA) is developed based on the process family requirements. The PFA acts as reference architecture for the members of the process family and describes the basic structure for the applications of the process family. It defines the functional as well as the non-functional requirements on the process family. Moreover, the PFA describes which techniques shall be applied for realizing the variability (i.e. the variability mechanisms) and on which variation points they shall be applied. The selection of appropriate variability mechanisms is crucial for the design of the process family since they can have a substantial impact on the functional and non-functional properties of the system. Additionally, the proper selection of a variability mechanism guarantees for an easy generation of process family members based on the process family infrastructure. The following two examples shall illustrate the impact of the variability mechanism selection on the properties of an automotive system. For a engine control unit optionally providing the feature speed control for example, the variability mechanism parameterization may be a good choice since it allows for the easy configuration of the car by parameterizing the embedded software after the assembly of the car to implement speed control or not. This allows for a more flexible reaction to changing customer preferences. However, a performance decrease can be the result, since unused code remains in the embedded software. On the other hand, for other variabilities, which can be encapsulated well and which shall not be configured after software installation, the variability mechanism encapsulation may make more sense, since it provides a better runtime performance and allows for an easy addition of new variants. Consider for example a windshield wiper, whose speed is adjusted by an encapsulated speed control subsystem. The speed of the wiper can either be static or adjusted by a manual regulator. Now, if a rain sensor shall be supported instead, the configuration of the wiper control boils down to the insertion of a new subsystem implementation. However, using encapsulation the system cannot be reconfigured after installation of the software. So, the impact of process family engineering on the properties of the developed embedded automotive systems, like e.g. RAM requirements or performance, highly depends on the techniques applied for variability realization, i.e. the variability mechanisms. The impact of different variability mechanisms for embedded automotive systems on the system properties has been discussed in [9].

So, for supporting process family engineering, concepts and a notation for process family architecture variability mechanisms (in the following called PFA variability mechanisms) are required, which allow for modeling architecturally relevant decisions concerning the realization of the system’s variability. Figure 2 describes the dependencies between the process family requirements, the process family architecture, PFA variability mechanisms and implementing variability mechanisms. The model is structured according to the three phases of process family engineering into three packages: Analysis, Design and Implementation. The requirements on the process family members are realized by a corresponding PFA. The variability in the process family is modeled by means of variation points to which variants can be bound by applying PFA variability mechanisms. The variability mechanisms represented in the PFA are realized in the program code by so called implementing variability mechanisms. During the implementation different implementing variability mechanisms can come into question, which can show different binding times. Which variability mechanisms are available highly depends on the application domain. For implementing embedded systems in automotive for example, typically C and C++ are applied. The
Variation Point

derived variability mechanisms we can further divide this cat-
detail in section 3.1. Concerning the second category of de-
stricted form. Basic variability mechanisms are described in
dition/omission/replacement of single elements
mechanism
data type variability
data type variability
and
encapsulation of varying sub-processes
nisms:
We have identified four types of basic variability mecha-
ability mechanisms, which don't require any other variability mechanisms.
ability mechanisms. As the name in-
ability mechanisms and variability mechanisms, which are
introduce variability into process models.

Figure 2: Role of Variability Mechanisms in Process
Family Engineering

3. VARIABILITY MECHANISMS

This section identifies a set of architecturally relevant vari-
ability mechanisms. With architecturally relevant variability
mechanisms we denote those variability mechanisms hav-
ing a considerable impact on the architecture of a system.
We thereby refer to the variability mechanisms referenced
most frequently in the context of product family engineering.
Collections of variability mechanisms can be found for ex-
ample in [2, 3, 11, 12, 24, 26]. By defining variability
mechanisms for State Machines we provide a means to in-
troduce variability into process models.

Variability mechanisms can be categorized into basic vari-
ability mechanisms and variability mechanisms, which are
derived from basic variability mechanisms. As the name in-
dicates, basic variability mechanisms are stand-alone mecha-
nisms, which don’t require any other variability mechanisms.
We have identified four types of basic variability mecha-
nisms: encapsulation of varying sub-processes, parameter-
ization, addition/omission/replacement of single elements,
and data type variability. Since data flow is not modeled
in UML State Machines, we will leave out the variability
mechanism data type variability. Moreover, we support ad-
dition/omission/replacement of single elements only in a re-
stricted form. Basic variability mechanisms are described in
detail in section 3.1. Concerning the second category of de-
rived variability mechanisms we can further divide this cat-
egory into variability mechanisms derived by restriction (see
section 3.2) and by combination of other variability mecha-
nisms (see section 3.3). With State Machine inheritance and
extension we introduce two examples for variability mecha-
nisms derived by restriction and with the strategy pattern
an example for a variability mechanism derived by combi-
nation. For every PFA variability mechanism we describe
its functionality and its representation in UML State
Machines. Moreover, we give an example on how it can be
implemented.

In section 3.4 we show, how State Machines can be ex-
tended by a notation for variability. In section 3.5 we de-
scribe how application specific process architectures can be
derived by configuring the process family architecture model.
Due to a lack of space the variability mechanisms cannot be
illustrated all by respective State Machines but only exem-
plarily in section 4.

3.1 Basic Variability Mechanisms

3.1.1 Addition, Omission, and Replacement of Single
Elements

Functionality. Variability in software systems can be
reached by the addition, replacement, and omission of arbi-
trary parts of the software like for example program frag-
ments or entire components.

PFA Variability Mechanism. For State Machines we re-
strict the elements to be added, omitted, and replaced to
states and transitions. Here, for the sake of clarity, we pro-
vide a formal description for the addition, omission, and
replacement of states, while leaving it out for the other vari-
ability mechanisms due to a lack of space.

 Addition of states. A state ADD can be added between sev-
eral predecessor vertices PRE1...PREn and one successor
vertex SUC, i.e. all transitions interrupted by ADD origi-
nally have to lead from a predecessor vertex PRE1...PREn
to the same successor vertex SUC. Additionally, in order to
support automation, we have to assume, that SUC doesn’t
have any entry points. Moreover, for the first case we as-
sume, that ADD doesn’t have any entry or exit points. Now,
ADD can be added by setting SUC as the target vertex of
the transitions (T(PRE1,SUC)...T(PREN,SUC) origi-
nally leading from the predecessor vertexes to the successor
vertex. Moreover, a single transition T(ADD,SUC) from
ADD to SUC is added. In the case that ADD does have
entry points, additional information has to be provided, to
which entry point every interrupted transition shall lead.
Concerning exit points, for every exit point an additional
transition from the exit point to the successor vertex has to
be created.

 Addition of transitions. A new transition can be added be-
tween arbitrary vertexes.

 Omission of states. A state OMM being located between se-
veral predecessor vertexes PRE1...PREn and one succes-
sor vertex SUC can be omitted automatically, if all outgoing
arcs of OMM lead to the same target vertex SUC. During
omission the successor vertex SUC is set as the new desti-
nation vertex for all transitions T(PRE1,OMM)...T(PREn,OMM)
originally leading from the predecessor vertexes to the state
to be omitted. All transitions T(OMM1,SUC)...T(OMMn,SUC)
from the state to be omitted to the successor vertex are
deleted.
However, this works only if the events assigned to the deleted transition don’t represent a precondition for entering the successor vertex.

**Omission of transitions.** Transitions linking two vertexes can be omitted from a State Machine. However, the omission of a transition, which is the only one leaving a state, can lead to a deadlock situation, while the omission of a transition, which is the only one entering a state, can lead to a dead node. If pseudo states are concerned the syntax of the resulting State Machine may even be violated. Moreover, the omission of transitions having an exit point or entry point as source or target, leads to follow-up variabilities. A transition can be replaced by another transition with the same source and target vertex and different "event (argument-list) guard condition / action"-expression.

**Implementing Variability Mechanism.** In C and C++ encapsulated subprocess and event-handling variants can be added, omitted, and replaced by structured or scattered preprocessor directives as well as by conditional compilation.

### 3.1.2 Encapsulation of Varying Sub-Processes

**Functionality.** Application-specific subsystem implementations are inserted into an invariant subsystem interface.

**PFA Variability Mechanism.** In State Machines subsystem interfaces are represented by submachine states and, more precisely by their entry and exit point connection point references. Cases are conceivable where also the do-Activities of a submachine state shall be part of the interface definition. However, this is a matter of Activity Diagram variability, which is discussed in [23]. Varying subsystem implementations, represented by submachines corresponding to the interface defined by the submachine state, can be invoked alternatively.

**Implementing Variability Mechanism.** Encapsulation of subprocesses can be realized in C as well as in C++ by providing subprocess interface descriptions in header files the subprocess variants implement.

### 3.1.3 Parameterization

**Functionality.** Using parameterization variants of encapsulated subsystems are generated by configuring a generic encapsulated subsystem with a set of parameter values. The prerequisite for this is that all possible variants are provided in the application subsystem’s code.

**PFA Variability Mechanism.** For parameterizing a State Machine first a State Machine containing all possible variants is modeled. Afterwards, guard conditions are assigned to variant specific transitions, which are evaluated depending on the value of a global variable. Thus, parts of the State Machine can be activated/deactivated based on the value of the global variable. The global variable and its variant specific value are provided by the class, which acts as the classifier of the parameterized State Machine.

**Implementing Variability Mechanism.** In C, as well as in C++ parameterization can be realized by conditions in the program flow being affected by parameter values, which are read for example from a configuration file (selection on configuration data [9]).

### 3.2 Variability Mechanisms Derived by Restriction

#### 3.2.1 Inheritance

**Functionality.** Following the reuse-oriented inheritance definition by [25], inheritance allows for the arbitrary addition and redefinition of class properties.

**PFA Variability Mechanism.** State Machine inheritance restricts the set of all possible manipulations of single elements to a set of addition and redefinition transformations, which are compatible to the common understanding of inheritance. A State Machine inheritance definition is provided by the UML State Machine specification and is summarized for example in [1]. We follow the State Machine inheritance transformations suggested by the UML Specification.

**Implementing Variability Mechanism.** State Machine inheritance can be implemented by means of preprocessor directives and conditional compilation.

#### 3.2.2 Extension/Extension Points

**Functionality.** Extensions and Extension Points are used to extend a subsystem at predefined points, the extension points, by additional optional behavior selected from a set of possible variants.

**PFA Variability Mechanism.** Extensions/extension points are a restricted form of the variability mechanism encapsulation of varying subprocesses. In the case of extensions one of the possible alternative implementing submachines has to be a submachine without content, which is invoked in the case that the State Machine shall not be extended at the extension point.

**Implementing Variability Mechanism.** Extensions/Extension Points can be realized by function calls invoking either a noop function or one with the extending functionality. The respective function may either be integrated into the same file/class or sourced out into a separate file/class.

### 3.3 Variability Mechanisms Derived by Combination

#### 3.3.1 Design Patterns

Here we will concentrate on the 'strategy pattern' as one of the design patterns referenced most frequently in the context of product family engineering. However, the variability mechanisms introduced so far could be combined to realize further design patterns.

**Functionality.** The idea of the strategy pattern is to make different algorithm variants, which are hidden behind a common interface, interchangeable. The algorithm variants are derived from a common algorithm version using inheritance.

**PFA Variability Mechanism.** The strategy pattern corresponds to a combination of the variability mechanisms State Machine inheritance and encapsulation of varying subprocesses.
Using the strategy pattern variant specific submachines are derived by State Machine inheritance from an abstract, i.e. incomplete submachine assigned to a submachine state and inserted as the new implementation of the submachine state. **Implementing Variability Mechanism.** Since the strategy pattern is a combination of encapsulation of subsystems and inheritance, it can be implemented using a combination of the respective implementing variability mechanisms. These mechanisms can also be used to realize strategy with subclasses [9].

### 3.4 Notation

In the last section we have described a set of variability mechanisms for State Machines, which allow for describing how to realize variability in process families. However, in order to be able to create process family implementation artifacts based on the process family architecture and to support the derivation of application specific implementation artifacts, some additional information are required. At least the varying parts, i.e. the variation points, of the process family have to be identified. Moreover, the variants which can be bound to the variation points during application engineering have to be described and assigned to their variation points. And, of course, we have to represent the variability mechanism to apply for binding a variant to its variation point. Additionally, the representation of the binding time for the variability as well as the representation of system requirements a variant realizes have to be supported. The concepts, which are depicted in the basic metamodel in figure [2] as well as their interrelations determine the information to be represented in variability mechanism centric process family architectures. We provide a corresponding notation in the following paragraphs.

In order to highlight variation points in UML State Machines we assign the stereotype «VarPoint» to the affected State Machine element. Variants, which are identified by the stereotype «Variant» are assigned to their variation point using UML Dependencies. The variability mechanism is modeled by assigning a stereotype with a name of the variability mechanism to the Dependency relation. A list of stereotypes for identifying variability mechanisms is provided in figure [4]. Moreover, we suggest the introduction of the stereotype «Variable» to denote variability below the level of detail currently shown. Concerning parameterization, it can be used to highlight the parameter dependent process parts. The binding time can be displayed by means of a tagged value (tagged value key bt) of the variability mechanism stereotype and, if necessary, the implementing variability mechanism may be uniquely identified by adding an identifier (tagged value key id). The functional system requirements a variation point implements is represented by means of tagged value (tagged value key feature) of the variation point stereotype, which can hold a list of functional system requirements.

All in all, for the representation of variability in UML State Machines only lightweight UML extension mechanisms are required, which supports an easy integration of variability modeling into existing UML tools.

### 3.5 Configuration of the Process Family Architecture

During configuration of the process family architecture in the application development phase the variants which are not relevant for the application shall be deleted from the process family architecture in order to retrieve an application specific process architecture. Thereby, the programmers receive a blueprint for the implementation activity, which clearly communicates how process family implementation artifacts have to be configured in order to derive application specific implementation artifacts. The specification of the variability mechanism guarantees that the application specific implementation artifacts can be derived easily and according to the system requirements balanced thoroughly during the process family architecture design.

### 4. EXAMPLE

This section gives an example for the variability mechanism centric representation of a variant rich State Machine for a strongly simplified engine control unit process family. The feature model in figure [4] following the notation suggested by [5] illustrates that the engine control unit can have the optional features immobilizer and speed control.

**Figure 3: Stereotypes for identification of variability mechanisms in UML State Machines**

<table>
<thead>
<tr>
<th>Variability Mechanism</th>
<th>Stereotype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation of Varying Subprocesses</td>
<td>«Implementation»</td>
</tr>
<tr>
<td>Addition, Omission of Components</td>
<td>«Option»</td>
</tr>
<tr>
<td>Replacement of Components</td>
<td>«Replacement»</td>
</tr>
<tr>
<td>Parameterization</td>
<td>«Parameterization»</td>
</tr>
<tr>
<td>Data Type Variability</td>
<td>«DataTypeVariation»</td>
</tr>
<tr>
<td>Inheritance</td>
<td>«Inheritance»</td>
</tr>
<tr>
<td>Strategy Pattern</td>
<td>«StrategyPattern»</td>
</tr>
<tr>
<td>Extension/Extension Points</td>
<td>«Extension»</td>
</tr>
</tbody>
</table>

**Figure 4: Feature for Engine Control Unit Family**

Figure [5] shows the top-level process of the engine control unit family. Once the driver turns the ignition key from position 0 via I to II the engine startup is launched, which comprises the activation of relevant processes and a sensor and actor check. The submachine state „Startup“ is marked as variation point by the stereotype «VarPoint» since depending on the configuration two different submachines can be bound to the variation point using the variability mechanism encapsulation of varying sub processes. In contrast to the submachine variant „Startup No-Im“, which doesn’t consider an immobilizer, the variant „Startup Im“ is only left via the exit point „ok“ if the immobilizer has been deactivated. Else, the State Machine changes into the state „Error“ and the driver has to restart the engine. The activation/deactivation of the immobilizer can be modeled with an additional region, which is added to the engine control.
unit top level process. A respective process variant can be derived by State Machine inheritance as shown in the lower part of figure 5. If the state „Startup“ is left via the exit point „ok“ and the driver turns the ignition key from position II to position III, the composite state „Engine is Running“ is entered. The stereotype < Variable > indicates that the state „Engine is Running“ contains a variant-rich subprocess, which is shown in figure 6.

The composite state „Engine is Running“ comprises several parallel regions as sketched in figure 6. The region „Speed Control“ implements the optional feature „Speed Control“.

Here, the variability mechanism parameterization is applied for variability realization. A guard condition, which evaluates the parameter „SpeedControl“ assures that the State Machine leaves only the initial state and launches the speed control in the case that the State Machine has been parameterized accordingly. The variation point is the „SpeedControl“ attribute of the classifier the State Machine is assigned to.

![Engine is Running](image)

Figure 6: Engine Control Unit - Subprocess Engine Running

5. RELATED WORK

For modeling product family architectures a number of alternative techniques have been proposed. There are for example approaches following the separation of concerns principle. Examples are the Hyper/UML approach [17] and two investigations on the composition of Statechart diagrams [15] [19]. Another group of publications suggests the explicit expression of variability within reference architectures by means of UML annotations [8] [29]. Non-graphical approaches comprise MOF-based product family architectures, which can be represented as XMI files [13] and the parameterization of BPEL [7] processes [14]. Another publication aims at representing BPEL based reference models [21]. Finally, a group of techniques could be referred to as template approaches [4][11]. The work presented in this paper can be classified as a template approach. In section 6 we address some characteristics that distinguish our approach from existing ones.

6. CONCLUSIONS

Product Family Engineering of process oriented software hasn’t been considered adequately in research so far. Therefore, in this paper we have introduced an approach for process family architecture modeling and implementation, which contributes to a rapid and cost-effective development and deployment of customer tailored automotive software systems. We have shown how a process family architecture for a family of automotive systems can be modeled with UML State Machines. For representing variability in UML State Machine based process family architectures we have introduced a set of variability mechanisms for UML State Machines and outlined variability implementation issues.

Apart from our concentration on the behavioral aspects of process-oriented systems, in contrast to existing template approaches our variability mechanism centric design approach for process family architectures allows for taking into consideration requirements concerning the variability implementation. Moreover, due to their correspondence to implementing variability mechanisms they allow for an intuitively understandable description of the system’s variability at the design level. The availability of a set of variability mechanisms, which is expandable by restriction or combination, supports the comfortable modeling of the system’s variability and the efficient reuse of common architecture parts. Since a variability mechanism centric process family architecture exactly describes at which points in the system variability occurs as well as the configurations to be made for specializing the generic system parts, an easy derivation of application specific implementation artifacts based on the configured process family architecture is possible. Finally, due to the utilization of only lightweight UML extension mechanisms existing UML State Machine tools can be enriched by variability modeling without major effort. This was also a prerequisite for an ongoing cooperative project between the Hasso-Plattner-Institute and DaimlerChrysler, which aims at the enhancement of the IBM Rational Software Modeler by variability mechanisms.

While in the context of PESOA two case studies have been performed for modeling variability mechanism centric process family architectures for an engine and a wiper control family following the approach described in this paper, open issues subject to future research comprise an extensive study of PFA-variability mechanisms and implementing variability mechanisms leading to a complete and domain independent set of PFA- and implementing variability mechanism categories. The exact relationship between the PFA- and implementing variability mechanism categories shall be analyzed subsequently.

7. REFERENCES

Figure 5: Engine Control Unit - Top Level Process