Product Line Engineering with the UML: Deriving Products
Tewfik Ziadi, Jean-Marc Jézéquel

To cite this version:
inria-00512550

HAL Id: inria-00512550
https://inria.hal.science/inria-00512550
Submitted on 30 Aug 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.
Software Product Line Engineering with the UML: Deriving Products

T. Ziadi and J.-M. Jézéquel

Abstract
Software product line engineering introduces two new dimensions into the traditional engineering of software-based systems: the variability modeling and the product derivation. The variability gathers characteristics that differ from one product to another, while the product derivation is defined as a complete process of building products from the product line. Software Product Line Engineering with the UML has received a lot of attention in recent years. However, most of these works only concern variability modeling in UML static models and few works concern behavioral models. In addition, there is very little research on product derivation. This chapter investigates the product derivation in the context of the product line engineering with the UML. First, a set of extensions are proposed to model product line variability in two types of UML models: class diagrams (the static aspect) and sequence diagrams (the behavioral aspect). Then we formalize product derivation using a UML model transformation. An algorithm is given to derive a static model for a product and an algebraic approach is proposed to derive product-specific statecharts from the sequence diagrams of the product line. Two simple case studies are presented, based on a Mercure product line and the banking product line, to illustrate the overall process, from the modeling of the product line to the product derivation.

15.1 Introduction

Rather than describing a single software system, the model of a software product line (PL) describes the set of products in the same domain. This is done by distinguishing elements shared by all the products of the line, and elements that may vary from one product to another. Concepts of commonality and variability are, respectively, used to designate common and variable elements in a PL [39]. Variability can concern two main aspects: optionality or variation [7,18]. An optional element only concerns some products and it can be omitted in others. Variation elements define alternatives (variants) to choose from. Beyond variability modeling, the product derivation process is defined as a complete process of constructing products from the software PL [12].

Unified modeling language (UML) [33] is an object-oriented notation for software system modeling. It proposes a set of models to specify several aspects of systems. Class diagrams are UML models that can be used to specify static aspects of systems, while
sequence diagrams (SD) and statechart diagrams are examples of models describing behavioral aspects. Software PL Engineering with the UML has received a lot of attention in recent years [3,5,9,10,13,14,18,26,27,37,38]. Section 15.4 presents a study on these works and shows that the most of existing works only concern UML static models and few works concern behavioral models [3,14,17]. In addition, there is very little research on product derivation [3,13]. The product derivation support is a significant criterion for determining the utility for users of any PL approach. The approaches that only model variability in UML models without product derivation support have only a descriptive utility. This means that these approaches are only useful for PL architecture description.

In this work we defend the idea that any approach of PL engineering should go beyond the descriptive utility and propose supports for resolving the variability and obtaining product models. For this, we investigate the product derivation process in the context of PL engineering with the UML. We give an overview of PL design by first presenting structural variability involved in class diagrams, then how behavioral aspects may be designed using UML sequence diagrams. We then formalize product derivation as UML model transformations. First, a transformation algorithm is given to automatically derive the static product model from the PL model. Second, an algebraic approach is proposed to derive product-specific statecharts from PL sequence diagrams.

To present these design techniques, Sect. 15.2 focuses on static aspects of the PL design, its constraints, and its derivation process into specific products; this part also stresses the need to check derived products with respect to variability constraints. Next, Sect. 15.3 proposes an algebraic approach to derive product-specific statecharts from the SD of the PL. Here PL behaviors are specified as algebraic expressions on basic UML2.0 sequence diagrams, where variability is introduced by means of three new algebraic constructs. Our derivation approach is defined in two steps: We first define an algebraic way to derive product expressions from the PL expression and then statecharts are generated by transforming product SD given as an expression into a composition of statecharts. Section 15.4 discusses related work, and finally Sect. 15.5 draws some conclusions and perspectives.

15.2 Deriving Static Aspects

15.2.1 The Mercure Product Line

As a case study for describing static aspect derivation, we consider the Mercure PL, which is a line of Switched Multi-Megabit Data Service (SMDS) servers whose design and implementation have been described in [23,24]. It can abstractly be described as a communication software delivering, forwarding, and relaying messages from and to a set of network interfaces connected into heterogeneous distributed system. The Mercure PL must handle variants for five variation points: any number of specialized processors (Engines), network interface boards (NetDriver), levels of functionality (Manager), user interface
(GUI) and support for languages (Language). Figure 15.1 shows a feature diagram of the Mercure PL (we follow FODA notations [28]). The Mercure consists of Engine, Net Driver, Manager, GUI, and Language. The Mercure product may support one or more of Engine 1, …, Engine N, the selection being represented by FODA alternative features. In the same way, we define all NetDriver, Manager, GUI, and Language dimensions.

The FODA [28] notations allow us to specify dependency relationships, called composition rules, between domain features. FODA supports two types of composition rules: the “require” rule that expresses the presence implication of two or more features, and the “mutually exclusive” rule that captures the mutual exclusion constraint on feature combinations. A “require” rule is identified in the context of the Mercure PL: it specifies that the choice of the NetDriver1 implies the choice of the Engine1 (see Fig. 15.1).

**Fig. 15.1.** The FODA diagram for the Mercure PL

### 15.2.2 PL Static Architecture as UML Class Diagrams

To describe the PL static architecture, we use UML class diagrams. In [42], we have proposed a UML profile for PL. This profile includes mechanisms to specify variability within two types of UML 2.0 diagrams: class diagrams and sequence diagrams. For class diagrams, we proposed to specify variability using two mechanisms:

- **Optionality.** Optionality in PL means that some features are optional for the PL members, i.e., they can be omitted in some products. To specify optionality in class diagrams, we introduced the `<<optional>>` stereotype. This stereotype can be applied to classes, packages, attributes, or operations [42].
Variation. Inheritance in UML allows defining variability in class diagrams [2]. The idea is to define a variation point as an abstract class and variants as concrete subclasses. Each subclass defines the implementation of the abstract class in a specific way. However, this variability is only resolved at run time and it is not explicit in the model. To explicitly specify the variation in UML class diagram, we introduced two stereotypes <<variation>> and <<variant>> [42]. The <<variation>> stereotype is associated with the abstract class while <<variant>> is associated with subclasses. Each product can choose one or more subclasses [42]. Figure 15.2 shows an example of a variation point specified using the <<variation>> and <<variant>> stereotypes. Notice that the subclass A in Fig. 15.2 is not stereotyped <<variant>>; this means that this subclass is mandatory for all products.

![Fig. 15.2. Example of a variation point](image)

Let us now apply these extensions to the Mercure PL. As previously specified in the FODA diagram of the Mercure PL, the Mercure product may support a set of Engines among Engine1, Engine2, EngineN. Using the variation mechanism presented earlier, we define an abstract class called Engine and stereotyped <<variation>> and the several dimensions as subclasses stereotyped <<variant>>. In the same way we specify other variation points: NetDriver, Manager, GUI, and Language. Figure 15.3 shows the UML class diagram of the Mercure PL. It basically says that a Mercure system is an instance of the Mercure class, aggregating an Engine (that encapsulates the work that Mercure has to do on a particular processor of the target distributed system), a collection of NetDrivers, a collection of Managers (that represent the range of functionalities available), and the GUI that encapsulates the user preference variability factor. A GUI has itself a collection of supported languages (see Fig. 15.3).
15.2.3 Product Line Constraints

In addition to variability, the PL architecture is defined as a standard architecture with a set of constraints [4]. In this context, we have identified in [45] two types of PL constraints that guide the product derivation process. We proposed to define them as Object
Both the generic constraints that apply to all PLS, and specific constraints that concern a specific PL (a detailed description of these constraints can be found in [45]).

**Generic Constraints**

The introduction of variability using the <<variant>>, <<variation>>, and <<optional>> stereotypes improves genericity, but can generate some inconsistencies. For example, if a mandatory element depends on an optional or on a variant one, the derivation can produce an incomplete product model. So the derivation process should preserve the consistency of the derived products. In [45], we proposed the formalization of consistency constraints using OCL and we called them *Generic Constraints*. An example of such constraint is the dependency constraint that forces mandatory elements to depend on mandatory ones only. It is specified using OCL as the following invariant for the *Dependency*\(^1\) metaclass:

```oclm
context Dependency inv:
    self.supplier->exists (S|
        S.isStereotyped('optional') or
        S.isStereotyped('variant')) implies
    self.client->forall (C|
        C.isStereotyped('optional') or
        C.isStereotyped('variant'))
```


\(^1\)A dependency in the UML specifies a require relationship between two or more elements. It is represented in the UML metamodel [33] by the metaclass *Dependency*; it represents the relationship between a set of suppliers and clients. An example of the UML Dependency is the “Usage,” which appears when a package uses another one.

**Specific Constraints**

A fundamental characteristic of the PL is that all elements are not compatible. That is, the selection of one element may disable (or enable) the selection of others. For example in the class diagrams for the Mercure PL in Fig. 15.3, the choice of the class variant *Net-Driver1* in the specific product needs the presence of the *Engine1* variant. Another challenge for the product derivation is to ensure these dependencies in the derived products. In our work, these dependencies are called *Specific Constraints* and are also formal-
ized as OCL metalevel constraints [45]. The presence constraint in the Mercure PL is formalized as an invariant for the Model metaclass as follows:

```plaintext
context Model inv:
    self.presenceClass('NetDriver1') implies self.presenceClass('Engine1')
```

presenceClass(C) is an auxiliary operation indicating if a specific class called C is present in the model. It is formalized using OCL as follows:

```plaintext
context Model::presenceClass(C : Class) : Boolean;
    presenceClass =
        self.ownedMember->exists(el : NamedElement|
            (el.oclIsKindOf(Class) and el.name = C.name) or
            (el.isKindOf(Namespace) and el.presenceClass(C)))
```

### 15.2.4 From Product Line Models to Product Models

Deriving static aspects in PL consists in generating the UML class diagram of each product from the PL class diagram. As shown previously, the PL class diagram is defined by a set of variation points and to derive a product-specific class diagram, some decisions (or choices) associated with these variation points are needed. For example, each Mercure product could choose among the presence or absence of all variant classes. A mechanism is needed to capture the decisions that are made for a specific product. As in [3], we call this mechanism a decision model. In this section, we propose to use the Abstract Factory design pattern as a decision model associated with the PL class diagram. Then we propose an algorithm, based on models transformation, to derive product class diagrams. To illustrate this algorithm, we use three products in the Mercure PL: FullMercure, CustomMercure, and MiniMercure:

- **FullMercure** is the product that includes all NetDrivers, all Engines, all Managers, all GUIs, all Languages. Thus, all combinations can be dynamically bound.
- **CustomMercure** is a restricted product. It only supports two different network drivers: NetDriver1 and NetDriver2, one manager: Manager1, two GUIs: GUI1 and GUI2, two languages: Language1 and Language2.
- **MiniMercure** is the lightest product that only supports NetDriver1, Engine1, GUI1, Manager1, and Language1.

#### The Decision Model

The Abstract Factory is a creational design pattern [15]. It allows defining an interface for creating a line of related objects. In [25], one of the authors proposed the use of this pattern to refine product derivation at compilation time. Our aim in this section is to reuse again this pattern as a design of the PL decision model. Figure 15.4 shows the structure of our decision model applied to the Mercure PL. We use an abstract factory, called Mercure_Factory, to define an interface for creating variants of Mercure’s five variation
points. The abstract class Mercure_Factory defines five factory methods, one for each variation point. new_gui() for example is the factory method, which concerns the GUI variation point. These factory methods are abstractly defined in the class Mercure_Factory and given concrete implementation in its subclasses called concrete factories. We create one concrete factory for each product in the PL FullMercure, CustomMercure, and MiniMercure in Fig. 15.4 are concrete factories for the Mercure PL. We propose to specify decisions related to each product using stereotypes applied to method factories. We use stereotypes to restrict the return type of factory methods to the possible one. For example, the CustomMercure product model includes only GUI1 and GUI2. The Factory Method that corresponds to the GUI variation point is new_gui(), so we add two stereotypes <<GUI1>> and <<GUI2>> to this factory method (see Fig. 15.4).

![Diagram](image)

**Fig. 15.4.** The Abstract Factory as a decision model for the Mercure PL

**Derivation**

Now we have to tackle the automation of the derivation process exploiting the variation points and the decision model. The derivation algorithm we use to derive product models is described in Fig. 15.5. It takes as input the PL class diagram, and the concrete factory from the decision model and it generates as output the product class diagram. It is decomposed into three steps: selection of variant classes, model specialization, and model optimization. They are:
– **Step 1: Variant classes selection.** The first step consists of selecting variant classes using the concrete factory. For each factory method, we retrieve its stereotypes. These stereotypes define the names of the selected subclasses of the abstract class returned by the factory method. When the factory method does not define stereotypes (such as in the **FullMercure** concrete factory methods), all the subclasses of its return type are selected.

– **Step 2: Model specialization.** In this step, we remove all variants classes from the model that have not been selected in the first step. However, to preserve coherence, variant ancestors of selected variant elements are not removed.

– **Step 3: Model optimization.** Here we delete unused factories and optimize the inheritance. Inheritance optimization is applied when there is only one concrete class inheriting from an abstract one. In this case the abstract class is omitted and replaced by the concrete one.

---

**Algorithm: DeriveProductModels()**

**Input:** PL_classDiagram: Model, aConcreteFactory: Class

**Output:** Product_classDiagram: Model

– **Step 1: Variant classes selection**

```java
selectedVariantsList := [];
for each factory method in aConcreteFactory do
    define definedVariantsList to significant stereotypes of the factory;
    if definedVariantsList is empty then
        selectedVariantsList.add(all sub-classes of the returned type of the factory);
    else
        selectedVariantsList.add(definedVariantsList);
    end if
end for

– **Step 2: Model specialization**

for each variant class C in PL_classDiagram do
    if (the class name of C not in selectedVariantsList) and (names of all sub classes of C not in selectedVariantsList) then
        delete the class C from the PL_classDiagram;
    end if
end for

– **Step 3: Model optimization**

delete unused concrete factories;
optimize inheritance;
Product_classDiagram := PL_classDiagram;
return Product_classDiagram;
```

---

**Fig. 15.5.** Static aspect derivation: the derivation algorithm
To achieve the implementation of the derivation algorithm, we have used the INRIA Model Transformation Language (MTL). Information about implementation and technical materials can be found at http://modelware.inria.fr/mtl. We have applied the derivation for the three Mercure products: FullMercure, CustomMercure, and MiniMercure. Figure 15.6 shows the CustomMercure model obtained by derivation from the Mercure model in Fig. 15.3.

Fig. 15.6. The CustomMercure model, automatically derived from the Mercure PL model
Derivation vs. Constraints

The PL model should satisfy generic constraints before the derivation and the product model derived should satisfy specific constraints. The generic constraints represent the preconditions of the derivation algorithm while specific constraints represent the post-conditions:

\[
\text{DeriveProductModels(PL\_classDiagram:Model, } \\
\text{aConcreteFactory:Class)} \\
\text{\hspace{1cm} pre: check Generic Constraints on PL classDiagram} \\
\text{\hspace{1cm} post: check Specific Constraints on the Product classDiagram} \\
\text{result.}
\]

15.3 Deriving Behavioral Aspects

In addition to static aspect description, behavior modeling plays an important role in the traditional engineering of software-based systems; it is the basis for systematic approaches to requirements capture, specification, design and simulation, code generation, testing, and verification. Scenario languages such as UML2.0 SD are an example of formalisms for modeling behavior. They focus on the global interactions between actors and system components. To be useful in the PL context, SD should also allow for expression of variability. We show in this section how variability can be expressed in UML2.0 SD using UML stereotypes and tagged values. We take advantage of UML2.0 SD and their composition operators to specify PL SD as algebraic expressions extended by algebraic constructs for variability. Then we present an algebraic approach to derive the product behaviors from the PL SD. Before illustrating behavioral aspect derivation, we briefly present the banking product line (BPL) as an example, which is used throughout this section.

15.3.1 The Banking Product Line

In this section, we reuse the example of a BPL as described in [3]. It is a set of products providing simple functionalities to clerks in the banking domain. It provides four main functionalities:

- **Creation of accounts (F1).** Customers are able to open simple accounts but must do so with a minimum balance. Account can have an associated limit specifying to what extent a customer can overdraw money.
- **Money deposit on accounts (F2).** Customers can deposit an amount of money on their accounts.
- **Money withdrawal from accounts (F3).** Customers can withdraw money from their account. If the account has a limit, a customer can only withdraw money up to this limit. If not, he (or she) cannot withdraw beyond the current balance of the account.
- **Currency exchange calculation (F4).** The bank system can offer a functionality for exchange calculation. This particularly concerns currency exchange: euros, dollars, etc.
Variability in the BPL example concerns the support of overdrawing to a set limit, which is optional because some products do not allow the addition of limits on accounts. Currency exchange calculation is also an optional functionality and it is only supported by some products. Table 15.1 shows four different product members of the BPL. The BS1 product for example supports limits on accounts and does not support exchange calculation while BS4 is a complete product with limits on accounts and exchange calculation support.

<table>
<thead>
<tr>
<th>product</th>
<th>limit support</th>
<th>exchange calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS1</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>BS2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>BS3</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>BS4</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

15.3.2 Product Line Behaviors as UML2.0 Sequence Diagrams

UML2.0 Sequence Diagrams

UML2.0 SD [33] enhances the previous versions of scenarios proposed in UML1.x by introducing composition operators. A basic SD describes a finite number of interactions between a set of objects. The semantics of a basic SD is now based on partially ordered events (instead of ordered collections of messages as in UML1.x), which makes it easy to introduce concurrency and asynchronism, and allows the definition of more complex behaviors.

Figure 15.7 shows the basic SD related to the Banking PL. A UML2.0 SD is represented by a rectangular frame labeled by the keyword sd followed by the name of the SD. The SD Deposit for example shows interactions between Clerk, Bank, and Account to deposit an amount on a specific account. The vertical lines represent life lines for the given objects. Interactions between objects are shown as horizontal arrows called messages (like deposit). Each message is defined by two events: message emission and message reception, which induce an ordering between emission and reception. Events located on the same lifeline are ordered from top to down.

UML2.0 basic SD can be composed into composite SDs called combined interactions using a set of operators called interaction operators [33]. We only use three fundamental operators: seq, alt, and loop. The seq operator specifies a weak sequence \(^2\) between the behaviors of two operand SDs. The alt operator defines a choice between a set of interaction operands. The loop operator specifies an iteration of the SD. For all these operators, each operand is either a basic or a combined SD. The combined SD BankPL in Fig. 15.8 shows how basic SDs for the BPL are related. It refers to the basic interactions.

\(^2\)UML2.0 [33] defines two operators, seq and strict, to define weak and strict sequence, respectively. A weak sequence means that only events on the same lifeline in the first SD are executed before events on the same lifeline in the second SD. A strict sequencing means that all events in the first SD are executed before events in the second diagram.
using the ref operator. BankPL specifies that there are five main alternative behaviors for requirements of BPL members (1) Account creation. (2) Deposit on account. (3) Withdraw from account (this last functionality is described using the combined SD WithdrawFromAccount). (4) Exchange calculation from euro and (5) Exchange calculation to euro. Following UML2.0 notations [33], combined SDs are defined by rectangles whose left corner is labeled by an operator (alt, seq, loop). Operands for sequence and alternative are separated by dashed horizontal lines. Sequential composition can also be implicitly given by the relative order of two frames in a diagram. For example, in the SD BankPL basic SD CreateAccountOk is referenced before SD SetLimit. This is equivalent to the expression CreateAccountOk seq SetLimit.
Variability in Sequence Diagrams

As shown in [42,43], variability can be specified in UML2.0 SD using simple stereotypes and tagged values. We briefly describe here these mechanisms; interested readers can refer to [42,43] for more details:

- **Optionality.** A combined SD can refer to an optional SD: interactions specified by this optional SD are only supported by some products and can be omitted in others. To specify optionality of an SD, we introduced the `<<optionalInteraction>>` stereotype and the `optionalPart` tagged value. The tagged value specifies the occurrence name of the optional SD (to differentiate among various occurrences of the optional SD, since an optional SD might be referred to more than once in the same combined SD). Figure 15.8a shows an example of a combined SD called CDS1, which refers to an optional SD called SD1. The tagged value `optionalPart` takes `SD1-occ1` as value.3

- **Variation.** This variability mechanism makes it possible to define a set of variants of behaviors from which a particular product would have to select exactly one variant. Using UML2.0 SDs, the variation of the behavior is modeled as a combined SD stereotyped `<<variation>>`, which refers to a set of subinteractions stereotyped `<<variant>>`. Each subinteraction specifies a variant behavior. As for the optional SD, a variation SD `<<variation>>` can be referred to several times in the same combined SD. To differentiate among multiple occurrences, we introduce the tagged value `variationPart` to specify the name of the occurrence. Figure 15.8b shows an example of a variation SD called CSD2, which refer to two SD variants SD-v1 and SD-v1. Note that this variation mechanism is different from the `alt` interaction operator. The variation mechanism proposes a choice that must be made at product derivation time so that the derived product contains only one of the alternative behaviors, while the `alt` operator defines a choice made after the product derivation, i.e., at run time.

- **Virtuality.** The virtuality of an SD means that its behavior can be redefined by another SD or refinement associated to a specific product. This type of variability is inspired by an existing construction in MSC [22]. The behavior of the virtual SD will be replaced at product derivation time by the behavior of the refinement SD associated with the product. Virtuality is introduced by the stereotype `<<virtual>>` and the tagged value `virtualPart` indicating the occurrence of the virtual interaction. Figure 15.8c shows an example of a combined SD called CSD3, which refers to a virtual SD called SD3.

3 We follow new notations of tagged values in UML2.0: a tagged value is now represented in UML2.0 as a note [33].
The combined SD in Fig. 15.9 BankPL illustrates two variability mechanisms: *optionality* and *variation*.

1. Since some products of the BPL do not support overdrawing, a stereotype `<<optionalInteraction>>` is added to the basic SD `SetLimit` and the tagged value `optionalPart` takes the value `settingLimit` (see the combined SD `AccountCreation` in Fig. 15.9). In addition, since exchange calculation is an optional functionality in the BPL, basic SD `SetCurrency`, `ConvertToEuro`, and `ConvertFromEuro` are defined as optional too (see the combined SD `AccountCreation` in Fig. 15.9).

2. There are two SD variants when withdrawing from an account: withdraw with balance and limit checking, and withdraw with balance checking only. The SD `Withdraw` is defined with the `<<variation>>` stereotype. The two SDs `WithdrawWithLimit` and `WithdrawWithoutLimit` are stereotyped `<<variant>>`. The tagged value `variationPart` takes `withdraw Account` as value (see the `WithdrawFromAccount` combined SD in Fig. 15.9).
Algebraic Specification

Taking advantage of UML2.0 composition operators for SD, we introduce in this section an algebraic specification of UML2.0 SDs in the form of reference expressions. We then extend it for PLs by including variability constructions defined above.

**Definition 1.** A reference expression for SD (noted RESD hereafter) is an expression of the form:

\[
\text{RESD} :: = \text{PRIMARY} \ ( \ "\text{alt}\" \ \text{RESD} \ | \ "\text{seq}\" \ \text{RESD}\)*
\]

\[
\text{PRIMARY} :: = \text{EØ} \ | \ \text{IDENTIFIER} \ | \ "(\text{RESD})" \ | \ "\text{loop}\" \ (\ "\text{RESD}\")\)*
\]

\[
\text{IDENTIFIER} :: = (\ [\text{a} - \text{z}, \text{A} - \text{Z}] \ | \ [\text{0} - \text{9}]\) *
\]

seq, alt and loop are the SD operators mentioned above. EØ is the empty expression that defines a sequence diagram without interaction.

So far, this algebraic framework does not contain any means to specify variability. We introduce three algebraic constructs that correspond to the three variability mechanisms presented earlier. This allows the definition of optional, variation, and virtual expressions.

**Definition 2.** The optional expression (OpE) is specified in the following form:

\[
\text{OpE} ::= \text{"optional\"} \ \text{IDENTIFIER} \ [\ "\text{RESD}\"]
\]

where \text{IDENTIFIER} refers to the name of the optional part and the \text{RESD} refers to its corresponding expression.

An optional SD (i.e., an SD stereotyped <<optionalInteraction>>) can be specified by an optional expression. The tagged value \text{optionalPart} in the diagram specifies the name of the expression. For the BPL example, optionality of the interaction SetLimit is specified by the expression:

\text{optional settingLimit [ SetLimit ]}

**Definition 3.** A Variation expression (VaE) is defined as follows:

\[
\text{VaE} ::= \text{"variation\"} \ \text{IDENTIFIER} \ [\ "\text{RESD}\", \ (\ \text{RESD}\)* \ ]
\]

For example, the variation interaction Withdraw in Fig. 15.9 encloses two interaction variants. It is specified algebraically as follows:

\text{variation withdrawAccount [ WithdrawWithLimit, WithdrawWithoutLimit ]}
Fig. 15.9. The UML2.0 combined sequence diagram for the Banking PL
Definition 4. Virtual expressions (ViE) are specified as:

\[ \text{ViE} ::= "virtual" \text{<IDENTIFIER>} "[" <\text{RESD}> "]" \]

Hence, algebraic expressions including variability will be defined by expressions of the form:

\[
\text{<RESD-PL>} ::= \text{<PRIMARY-PL>} (\"alt\" \text{<RESD-PL>} | \"seq\" \text{<RESD-PL}>)*
\]

\[
\text{<PRIMARY-PL>} ::= \text{EØ} | \text{<IDENTIFIER>} | \"(" <\text{RESD-PL}> ")\" | \"loop\" "(" <\text{RESD-PL}> ")" | \text{VaE} | \text{OpE} | \text{ViE}
\]

The SD BankPL of Fig. 15.9 can be algebraically represented by the following expression:

\[
\text{EBPL} = \text{loop} (\text{Deposit alt} \text{ (CreateAccount seq (CreateAccountOk seq (optional settingLimit[SetLimit]) seq (optional settingCurrency [SetCurrency ]))) alt CreateAccountFailed})
\]

\[
\text{alt} (\text{( variation withdrawAccount [ WithdrawWithLimit, WithdrawWithoutLimit] seq (WithdrawOk alt WithdrawFailed))})
\]

\[
\text{alt} (\text{(optional fromEuro [ ConvertFromEuro ]})
\]

\[
\text{alt} (\text{(optional toEuro [ ConvertToEuro ]})
\]

15.3.3 Deriving Product Behaviors

In section “Algebraic specification,” we have specified PL behaviors using scenarios represented as UML2.0 SD enriched with variability mechanisms. Scenarios are not the only way to describe software behaviors; statecharts [19] are another formalism that is often used to depict the behavioral aspects of systems. However, if scenarios capture requirements in the early stage of the development process, statechart models are more dedicated to detailed design phases as they are closer to the implementation (some tools such as Rhapsody [21] generate code from them). To formalize product behavior derivation, we have studied the problem of statechart synthesis from scenarios. Furthermore, scenarios and statecharts differ in their nature (scenarios capture interactions amongst a set of objects, and statecharts represent the internal behavior of a single object). Statechart synthesis out of a collection of scenarios has received a lot of attention in the context of single product development [29,30,32,40]. So far, the proposed solutions do not consider the PL aspects. In this section, we propose an algebraic approach to synthesize product statecharts from PL scenarios. Firstly, variability is resolved by deriving the RESD-PL into a set of RESDs, one for each product. Then statecharts are generated by transforming product scenarios given as an RESD into a composition of statecharts.
**Step 1: Product Expressions Derivation**

The first step toward product behavior derivation is to derive the corresponding product expressions from the RESD-PL. Decision resolutions for a specific product are defined in what we call an *Instance of decision model (IDM)*, which is defined as follows:

**Definition 5.** An *Instance of Decision Model* (noted hereafter IDM) for a product P is a set of pairs (nameᵢ, Resᵢ), nameᵢ designates a name of an optional, variation or virtual part in the RESD-PL and Resᵢ is its decision resolution related to the product P. Decision resolutions are defined as follows:

- The resolution of an optional part is either **TRUE** or **FALSE**.
- For a variation part with E₁, E₂, E₃.. as expression variants, the resolution is i if Eᵢ is the selected expression.
- The resolution of a virtual part is a refinement expression Eᵢ.

Table 15.2 shows four Instances of Decision Model associated with the four products in the BPL. For example, IDM₁ is the Instance of Decision Model associated with the product BS₁, which supports limits on accounts and does not offer the currency exchange calculation functionality.

The derivation can be seen as a model specialization through abstract interpretation of a generic PL expression in the IDMᵢ context, where IDMᵢ is the Instance of Decision Model related to a specific product. For each variability mechanism, the interpretation in a specific context is quite straightforward:

1. Interpreting an optional expression means deciding on its presence or absence in the product expression. This is defined as:

   \[
   \text{optional name [ E] } |_{\text{IDMᵢ}} = \begin{cases} 
   E & \text{if } (\text{name}, \text{TRUE}) \in \text{IDMᵢ} \\
   E₀ & \text{if } (\text{name}, \text{FALSE}) \in \text{IDMᵢ}
   \end{cases}
   
   \]

   Note that the empty expression is a neutral element for the sequential and the alternative composition. It is also idempotent for the loop, i.e:

   - E \text{ seq } E₀ = E ; E₀ \text{ seq } E = E
   - E \text{ alt } E₀ = E ; E₀ \text{ alt } E = E
   - \text{loop } (E₀) = E₀

   This allows us to replace a complete part of a RESD-PL by E₀ when this part should be removed.
2. Interpreting a variation expression means choosing one expression variant among its possible variants. This is defined as:

\[
[\text{variation name } [E_1, E_2, \ldots]]_{\text{IDM}_i} = E_j \text{ if } (\text{name}, j) \in \text{IDM}_i
\]

3. Interpreting virtual expressions means replacing the virtual expression by another expression:

\[
[\text{virtual name } [E]]_{\text{IDM}_i} = E' \text{ if } (\text{name}, E') \in \text{IDM}_i
\]

### Table 15.2. Instances of the decision model for the banking product line

<table>
<thead>
<tr>
<th>Product</th>
<th>Instance of decision model (IDM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS1</td>
<td>IDM1 = {(settingLimit, TRUE), (settingCurrency, FALSE), (withdrawAccount, 1), (fromEuro, FALSE), (toEuro, FALSE)}</td>
</tr>
<tr>
<td>BS2</td>
<td>IDM2 = {(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, FALSE), (toEuro, FALSE)}</td>
</tr>
<tr>
<td>BS3</td>
<td>IDM3 = {(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, TRUE), (toEuro, TRUE)}</td>
</tr>
<tr>
<td>BS4</td>
<td>IDM4 = {(settingLimit, TRUE), (settingCurrency, TRUE), (withdrawAccount, 1), (fromEuro, TRUE), (toEuro, TRUE)}</td>
</tr>
</tbody>
</table>

The BS2 product expression $E_{BS2}$ is obtained by the interpretation of the $E_{BPL}$ in the IDM2 context:

$E_{BS2} = [E_{BPL}]_{IDM2}$.

The derivation of the four optional expressions and the variation expression in $E_{BPL}$ is realized as follows:

\[
[\text{optional } \text{settingLimit } [\text{SetLimit}]]_{IDM2} = E_0
\]

\[
[\text{optional } \text{settingCurrency } [\text{SetCurrency}]]_{IDM2} = E_0
\]

\[
[\text{optional } \text{toEuro } [\text{ConvertToEuro}]]_{IDM2} = E_0
\]

\[
[\text{optional } \text{fromEuro } [\text{ConvertFromEuro}]]_{IDM2} = E_0
\]

\[
[\text{variation withdrawAccount } [\text{WithdrawWithLimit, WithdrawWithoutLimit}]]_{IDM2} = \text{WithdrawWithoutLimit}
\]
The reference expression obtained for the BS2 is the expression $E_{BS2}$ below. Since $E_\emptyset$ is a neutral element for seq and alt, $E_\emptyset$ is removed from the product expression:

$$E_{BS2} = \text{loop}(\text{Deposit alt (CreateAccount seq (CreateAccountOk) alt CreateAccountFailed) alt (WithdrawWithoutLimit seq (WithdrawOk alt WithdrawFailed}))}$$

The BS4 product, which provides overdriving on accounts and exchange operations, will be characterized by the presence of SetLimit, SetCurrency, ConvertToEuro, and ConvertFromEuro SDs; and by the choice of WithdrawWithLimit SD. The product expression obtained for product BS4 is:

$$E_{BS4} = \text{loop}(\text{Deposit alt (CreateAccount seq (CreateAccountOk seq (SetLimit seq SetCurrency )) alt CreateAccountFailed) alt (WithdrawWithLimit seq (WithdrawOk alt WithdrawFailed)) alt (ConvertFromEuro) alt (ConvertToEuro)})$$

**Step 2: Statechart Synthesis**

The derived product expressions are expressions without variability, i.e., expressions that only compose basic SDs by interaction operators: alt, seq, and loop. The second step of our derivation approach aims at generating statecharts for objects in each derived product. Product SD are translated into statecharts using the method proposed in [44]. We generate flat statecharts, i.e., statecharts without hierarchy. Figure 15.10 shows examples of flat statecharts, in which states represented by double circled states are called junction states. Junction states are introduced to formalize statechart composition [44]. Transitions are labeled e/a, where e is a triggering event and a is an action. $ST_\emptyset$ refers to an empty statechart, containing a single state, which is at the same time an initial and a junction state (see the $ST_\emptyset$ statechart in Fig. 15.10).

**Statechart Operators**

Our method for statechart synthesis is based on an algebraic framework for statechart composition. This framework is inspired by the algebraic composition of UML2.0 SD [44]. We have formalized three statechart operators: seq, alt, and loop, for the
sequencing, alternation, and the iteration of statecharts, respectively. In the rest of this
section, we briefly describe these operators; the complete formalization can be found in
[44]:

- **Sequence** ($seq$). The sequential composition of two statecharts is a statechart
  that describes the behavior of the first operand followed by the behavior of the
  second one. Figure 15.11 shows the sequential composition of the $ST_1$ and $ST_2$.
- **Alternative** ($alt$). The statechart resulting from the alternative composition
describes a choice between the behaviors of its operands. See for example $ST_1 alt ST_2$ in Fig. 15.11.
- **Loop** ($loop$). This operator defines iteration of a statechart. Figure 15.11 shows
  the iteration of the $ST_2$.

As for sequence diagrams, we algebraically describe statechart composition with refer-
ence expressions.

**Definition. 6.** A reference expression for statecharts (noted $REST$ hereafter) is an expres-
sion of the form:

$$<REST> ::= <PRIMARY-REST> | "alt" <REST> | "seq" <REST> | "loop" <REST> | "(" <REST> ")"$$

**Synthesis Process**

Using our algebraic framework for statecharts, translating product UML SD to statecharts
is defined in two steps: synthesis from basic sequence diagrams and synthesis from com-
bined SD. The next paragraphs describe these two steps.

**Synthesis from basic sequence diagrams.** In the first step of our synthesis method we gen-
erate statecharts from all basic SD in the PL. This step is based on an algorithm generating
a statechart $P(SD, O)$ depicting the behavior of each object $O$ in each basic SD SD. We
do not detail here the algorithm computing \( P(SD, O) \), which can be found in [44]. To summarize, this algorithm uses projections of SDs on object lifelines to generate the statecharts. Receptions in the SD become events in the statechart and emissions become actions. For a transition associated with a reception, the action part will be void, and for transitions associated with actions, the event part will be empty. The generated statechart contains a single junction state, which corresponds to the state reached when all events situated on an object lifeline have been executed. When an object does not participate in a basic SD, the algorithm generates an empty statechart. Figure 15.12 illustrates the synthesis of the statechart associated with the Bank object from the Deposit basic SD.
Figure 15.13 shows the flat statecharts generated from the twelve basic SDs from Fig. 15.9 for the Bank object.

**Synthesis from Combined Sequence Diagrams.** Once we have obtained a collection of statecharts through projections of basic SDs, we now deal with combined SDs. Our method is based on the correspondence between interaction operators and statecharts operators and it allows constructing RESTs from RESDs [44]. For each object O, a REST is constructed by replacing in the RESD seq, alt and loop by statecharts operators seqs, alts and loops, respectively, and each reference to an SD S by the statechart P(S,O). From the REST obtained, a statechart can be built using statechart composition operators.

Let us apply this construction method to the combined SD for the BS2 product. The Bank’s REST, called REST_{BS2} is described below. Figure 15.14 shows the statechart obtained from this REST.

```
REST_{BS2} = loop_{s} (P(Deposit, Bank) alt_{s} (P(CreateAccount, Bank)
seq_{s} (P(CreateAccountOk, Bank) alt_{s} P(CreateAccountFailed, Bank)))
alt_{s} (P(WithdrawWithoutLimit, Bank) seq_{s} (P(WithdrawOk,Bank)
alt_{s} P(WithdrawFailed,Bank))))
```

The same method can be applied for the BS4 product. Its reference expression E_{BS4} is transformed into the statechart composition expression REST_{BS4} defined below. Figure 15.15 shows the Bank statechart obtained from REST_{BS4}. Note that as BS2 and BS4 differ in the presence or the absence of an overdrawing limit and exchange operations, the synthesized statecharts differ in the transitions that concern these two functionalities. The differences between the statecharts obtained for product BS2 and BS4 are illustrated in Fig. 15.15 by gray zones.

```
E_{BS4} = loop_{s} (P(Deposit, Bank) alt_{s} (P(CreateAccount, Bank)
seq_{s} ((P(CreateAccountOk, Bank) seq_{s} P (SetLimit, Bank)
seq_{s} P(SetCurrency, Bank)) alt_{s} P (CreateAccountFailed, Bank)))
alt_{s} (P(WithdrawWithoutLimit,Bank) seq_{s} ((P (WithdrawOk, Bank)
alt_{s} P(WithdrawFailed, Bank)))
alt_{s} (P(ConvertFromEuro, Bank))
alt_{s} (P(ConvertToEuro, Bank)))
```
Fig. 15.13. Bank basic statecharts

Fig. 15.14. The Bank statechart in the BS2 product
15.3.4 Implementation and Validation

In the context of the ITEA Families [1] project, a prototype tool of the proposed approach has been implemented in Java and is integrated into the Eclipse platform. It is freely available from http://modelware.inria.fr/plibs. UML2.0 SD with variability are specified in Eclipse, thanks to the Omondo case tool (see Fig. 15.16a) Then RESD-PL are automatically extracted from these diagrams. The prototype implements product expression derivations from RESD-PL according to a given IDM. Then a statechart for a specific object is generated from the derived expression. The generated statecharts can be visualized using the Omondo case tool again (see Fig. 15.16b). A complete description of the prototype can be found at http://modelware.inria.fr/plibs.
We have used our approach for a complete BPL case study with 14 basic SDs. Table 15.3 shows statistics (number of states and transitions) on the generated statecharts for the Bank object in each BPL member (these statistics show that the generated statechart for the Bank object differs from one product to another). We have also validated our approach on two case studies: The camera PL [42] and the auction PL [41]. As we noticed in Sect. 15.3, some tools allow generating code from statecharts. We are currently studying code generation from the generated statecharts in our method using existing tools.

<table>
<thead>
<tr>
<th>product</th>
<th># states</th>
<th># transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS1</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>BS2</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>BS3</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>BS4</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

15.4 Related Work

Software PL Engineering with the UML has received a lot of attention in recent years. Table 15.4 summarizes existing work on PL engineering with the UML. Most of these works address variability modeling whereas only two works refer to the product derivation process.
For variability modeling, many works [5,17,18,26,37] are related to functional models (use cases). Halmans et al. [18] extend use cases with stereotypes to specify variability. Use cases are described using templates. Bertolino et al. [5] introduce tags to describe variability in a textual description of uses cases. In Chap. 11, readers can find a detailed description of Bertolino et al.’s work. Maßen et al. [37] extend the UML use case metamodel to support variability. John et al. [26] tailor use case diagrams and textual use cases to support PL requirements specification. In our work, we do not consider uses cases. Even if the textual description through templates, used by the previous works, is a good way to document PL requirements, SD are more operational and as shown with our approach detailed design can be generated from them.

There are many works [3,10,14,16,27,34,38] that propose extensions to specify variability in UML static models. However, few works model variability in behavioral models: Gomaa et al. [17] introduce variability in UML collaboration diagrams with three stereotypes <<kernel>>, <<optional>>, and <<variant>>. KobrA [3] introduces the stereotype <<variant>>, which can be applied to messages in SD and to statecharts. The KobrA’s solution to specify variability in SD is difficult to use in practice. Indeed, if all messages in the same SD are optional, the user should specify all these messages with the stereotype <<variant>>. This can compromise the readability of the SD. On contrary, our <<optionalInteraction>> is applied to the complete SD. Flege [13,14] also introduces variability in UML statecharts. Note that all these works only concern UML1.x models.

Concerning product model derivation, only KobrA [3] and Flege [13] refer to this. While we formalized product derivation as UML model transformations, KobrA and Flege do not propose a means to implement derivation. Cerón et al. [8] propose two practices implementing the product architecture derivation. The main assumption in this proposition is: the PL is defined by an engineering assets repository and each product should choose components from this repository to obtain a product-specific architecture. Haugen et al. [20] also use UML2.0 SD to specify behaviors of systems. They introduce a new operator called xatl to distinguish between mandatory and potential behaviors. A potential behavior represents a variant of a mandatory behavior. This is close to our variation construct where interaction variants correspond to the potential behaviors.

In addition to these works, readers can find in Chap. 6 a complete study about Model Driven Engineering for Software PLs. The chapter also proposes a framework for modeling variability in PLs.

In Sect. 15.3, we have used statechart synthesis from scenarios to derive product-specific behaviors. There are many works on statechart synthesis; however these works only concern single product development (i.e., without consideration for variability). To our knowledge, there are no other works proposing statechart synthesis from software PL scenarios. The next paragraph describes existing works on statechart synthesis in the context of a single product development. There are works that synthesis statecharts from UML1.x, from Message Sequence Charts MSC [22] and from Live Sequence Charts [11].

Due to the poor expressive power of UML1.x SD, the proposed solutions for statechart synthesis [29,30,32,40] often use additional information or ad hoc assumptions for managing several scenarios. For example, Whittle et al. [40] enrich messages in SD with pre- and postconditions given in (Object Constraint Language) OCL, which refer to global
state variables. State variables identify identical states throughout different scenarios and guide the synthesis process. Our approach does not use variables, and structures the state-charts and transitions based on information provided by lifeline orderings and SD operators. Koskimies et al. [30] use the Biermann–Krishnaswamy algorithm [6], which infers programs from traces. This work establishes a correspondence between traces and scenarios and between programs and statecharts. In [29,32] it is also proposed to use interactive algorithms to generate statecharts from UML1.x sequences diagrams.

Several other approaches [31,35,36] study statechart synthesis from MSC [22], a scenario formalism similar to sequence diagrams. MSCs allow composition of basic scenarios (bMSCs) with High-Level Message Sequence Charts (HMSC). This composition mechanism is very close to that of current SDs in UML2.0 and our approach can be used to generate statecharts from MSCs.

Finally, Chap 13 also uses SD but it uses them to derive product-specific .test cases from PL requirements and not for statechart synthesis.

### Table 15.4. Existing works on PL engineering with the UML

<table>
<thead>
<tr>
<th></th>
<th>variability modeling</th>
<th>Product Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>functional aspects</td>
<td>static aspects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>static aspects</td>
</tr>
<tr>
<td>Bertolino et al. [5]</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Halmans and Pohl [18]</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>John and Muthig [26]</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Mallen and Lichter [37]</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Robak et al. [34]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Clauß [9,10]</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Gomaa [16, 17]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Flöge [13, 14]</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SPLIT-Daisy [27]</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Webber [38]</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

### 15.5 Conclusions and Future Research

In this chapter we have described PL design and derivation techniques building on advanced model transformation technology. Working at the level of UML design models, derivation of both static and behavior aspects was considered. For static aspect derivation, we started from a class diagram modeling the full PL along with a decision model given in the form of a set of concrete factories to build specialized UML models corresponding to the selected products. The challenge of such model manipulation is to be able to transform the model accessing its metalevel and ensuring the integrity of the derived model according to the PL-specific constraints.
For behavioral aspects derivation, we started from UML2.0 Sequence Diagrams extended with algebraic constructs to specify variability. We use interpretations of the algebraic expressions to resolve the variability and derive product expressions, which are ultimately transformed into a set of product-specific statecharts. The introduction of variability in behavioral models can be used to factorize common behavioral models in different products, and should then facilitate domain-engineering phases. However, some parts of the synthesis can be reused from one product to another, hence facilitating reuse during application engineering. As discussed in [44], statechart synthesis should be considered more as a step toward implementation rather than as a definitive bridge from user requirements to code.

In the context of the ITEA Families [1] project, prototype tools of the proposed approaches have been implemented. We used Model Transformation Language MTL and its related framework UMLAUT-NG for implementing the static aspect derivation. For behavioral aspects, a prototype tool has been implemented in Java and integrated into the Eclipse platform. We used our approach in several case studies; however we hope in the future to use it in an industrial context.

Acknowledgments

This work has been partially supported by the ITEA project ip02009, FAMILIES in the Eureka σ! 2023 Program. We wish to thank Loïc Hélouët for many inspiring discussions. We also gratefully acknowledge the reviews of Stan Bühne, Juan Carlos Dueñas, Timo Kälkölä, Kim Laueneroth, Jim Steel, and Patrick Tessier, which significantly improved the quality of this chapter.

References

15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Pattern Elements of Reusable Object-Oriented Software (Addison-Wesley, Reading, MA 1995)
37. Webber, D.L.: The variation point model for software product lines. Ph.D. thesis (George Mason University, George Mason University, Fairfax, VA 2001)