HAL
archives-ouvertes.fr

Behaviors generation from product lines requirements

Tewfik Ziadi, Loïc Hélouët, Jean-Marc Jézéquel

▶ **To cite this version:**

Tewfik Ziadi, Loïc Hélouët, Jean-Marc Jézéquel. Behaviors generation from product lines requirements. Proc. UML2004 workshop on Software Architecture Description, Sep 2004, Lisbon, Portugal. 2004. <hal-00795037>

HAL Id: hal-00795037
https://hal.inria.fr/hal-00795037

Submitted on 27 Feb 2013

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.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Behaviors Generation From Product Lines Requirements *

Tewfik Ziadi, Loic Hélouët, Jean-Marc Jézéquel

IRISA, Campus de Beaulieu 35042 Rennes Cedex, France
{tziadi, lhelouet, jezequel} @irisa.fr

Abstract. Modeling variability in product lines (PL) has received a lot of attention in recent years, building on the idea that product could be automatically derived from a PL through model transformations, at least for its static architecture (e.g. class diagrams). This paper proposes to go beyond these static aspects by also addressing the behavioral aspect of software product lines. Inspired by the way UML2.0 sequence diagrams can be algebraically composed, we propose to specify PL behavioral requirements as algebraic expressions extended with constructs to specify variability. Then we propose a two stages approach to synthesize detailed behavior for each product member in the PL. The first stage uses abstract interpretation of the variability operators in scenarios to get behavior specialization of the PL according to a given decision criteria. The second stage uses statechart synthesis from product expressions. We describe the interest of our method on a well known case study, and briefly discusses its implementation in a prototype tool.

1 Introduction

The Software Product Line (PL) approach (also called Product Family), have received a great attention in last years. Several product line approaches concerning the entire software life cycle (requirements, design, development, testing, and evolution) have been proposed.

Capturing and specifying requirements in software development is a very important activity. Several notations and formalisms such as Use Cases and scenarios are now very popular for single products development. In the PL context, most works [7, 2, 21, 13] extend UML Use Cases with variability mechanisms to document PL requirements. They introduce variability into the textual description of Use Cases. In addition to textual templates, Use Cases can be illustrated by means of interactions between system objects using scenarios such as UML sequence diagrams.

While scenarios capture requirements in the early stage of the development process, statecharts [8] are often used for a more detailed design, as they are closer to the implementation. The idea of synthesizing statecharts out of a collection of scenarios has thus received a lot of attention in the context of single

* This work has been partially supported by the ITEA project ip02009, FAMILIES in the Eureka Σ! 2023 Programme

products development. However, no work proposes statecharts synthesis from PL requirements. In this paper we propose an algebraic approach that generates statecharts from PL scenarios, thus fostering a better traceability between PL requirements and the detailed design.

We specify PL requirements as algebraic expressions on basic UML2.0 sequence diagrams, where variability is introduced by means of three new algebraic constructs. Our synthesis approach is defined in two steps: we first define an algebraic way to derive product expressions from PL ones and then statecharts are generated by transforming product scenarios given as an expression into a composition of statecharts.

This paper is organized as follows: Section 2 shows, through the well known Banking Product Line (BPL) [1] example, how PL requirements are specified using UML2.0 sequence diagrams. Section 3 describes our synthesis approach and illustrates it on the BPL example. Section 4 discusses the interest of our approach. Section 5 presents related works.

2 PL Requirements as UML2.0 Sequence Diagrams

Capturing and specifying requirements is often a preliminary task during software development. Several notations such as Use Cases and Scenarios have been proposed to document and formalize systems requirements. To be useful in the PL context, these formalisms should allow for the expression of variability in requirements. Variabilities are characteristics that may vary from a product to another one. In this Section we use scenarios represented as UML2.0 sequence diagrams (SDs) to specify PL behavioral requirements. Variabilities are introduced by means of three mechanisms: optionality, variation and virtuality [24]. We take advantage from UML2.0 SDs and their composition operators to specify PL scenarios as algebraic expressions extended by algebraic constructs for variability. Before showing how PL requirements are specified using UML2.0 sequence diagrams, we first present an example that will be used throughout the paper.

2.1 Running example

Throughout this paper, we reuse the example of a Banking Product Line (BPL) as described in [1]. It is a set of products providing simple functionalities to clerks in the banking domain. It provides four main functionalities:

- Creation of accounts: 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.
- Money withdrawal from accounts.
- Currency exchange calculation(exhange from and to Euro).

Variability in the BPL example concerns the support of overdrawing to a set limit and the currency exchange calculation. Table 1 shows four different products members of the BPL. The `BS1` product for example supports limits on accounts and does not support exchanges calculation.

Table 1. The Banking PL Members

Product	Limit support	Exchange calculation
BS1	YES	NO
BS2	NO	NO
BS3	NO	YES
BS4	YES	YES

2.2 UML2.0 sequence diagrams

Sequences diagrams (SDs) have been extended in UML2.0 [6] by means of composition operators. This allows the specification of more elaborated behaviors than in UML 1.4, which contain alternatives, loops, and so on. In fact, UML 2.0 sequence diagrams can be considered as the algebraic composition of simple interactions, that will be called basic Sequence Diagrams hereafter.
Figure 1 shows basic SDs defining possible scenarios for the Banking PL. To simplify the presentation, we only show here a portion of the BPL excluding SDs related to exchange calculation. The sequence diagram `Deposit` for example describes the interaction of Clerk actor and two objects Bank and Account to deposit money on an account.

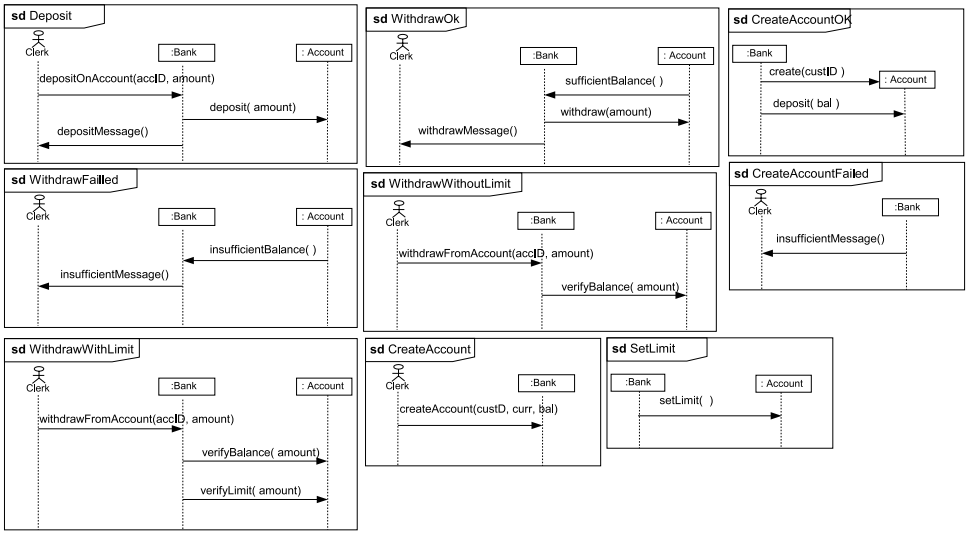

Fig. 1. UML2.0 Sequence Diagrams for the Banking PL

UML2.0 basic SDs can be composed in composite SDs called *combined interaction* using a set of operators called *interaction operators* [6]. We will only use three fundamental operators: `seq`, `alt`, and `loop`. The `seq` operator specifies a

weak sequence 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 an interaction. For all these operators, each operand is either a basic or a combined SD.

The combined SD `BPLPortion` in Figure 2 shows how basic SDs for the BPL are related. It refers to basic interactions using the `ref` operator. `BPLPortion` specifies that there are three 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`. Following UML2.0 notations [6], combined SDs are defined by rectangles which left corner is labelled 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 `BankSystem` basic SD `CreateAccountOk` is referenced before SD `SetLimit`. This is equivalent to the expression `CreateAccountOk seq SetLimit`.

2.3 Variability

As shown in [24], variability can be specified in UML2.0 sequence diagrams using simple stereotypes and tagged values. We briefly describe here three of these mechanisms, interested readers can consult [24] for more detail:

- **Optional interaction.** A sequence diagram can be defined as optional. This means that the interaction specified by this SD is only supported by some products.
- **Variation interaction.** A variation SD is a SD that encloses a set of SDs variants. For any given product, only one SD variant will be present.
- **Virtual interaction.** A virtual SD in a PL means that the interaction specified by this SD can be redefined and refined for a specific product by another SD.

Combined SD in Figure 2 shows two variability mechanisms: optionality and variation.

- As some products of the BPL do not support overdraft, a stereotype <<`optionalInteraction`>> is added to the basic SD `SetLimit`. Notice that the same basic SD can be referred several times as optional in the combined SD. To distinguish different occurrences of the optional SD, the tagged value `optionalPart` is associated to the <<`optionalInteraction`>> stereotype (see Figure 2, tagged values are represented in UML2.0 as notes)
- There are two interaction 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 variants, which is indicated by the <<`variant`>> stereotype (See the `WithdrawFromAccount` in Figure 2).

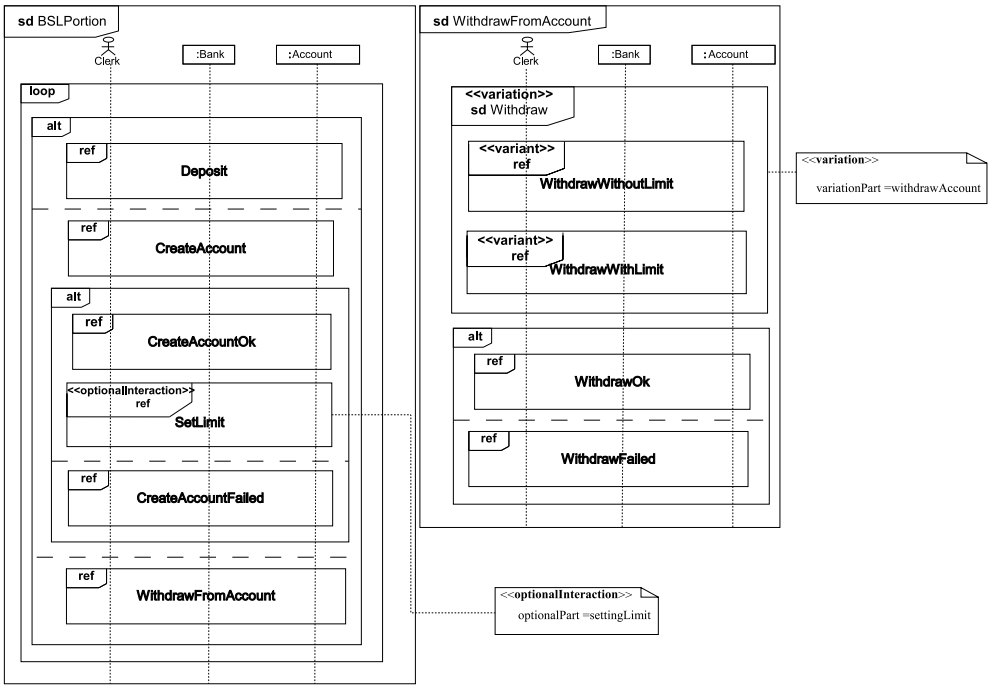

Fig. 2. The UML2.0 Combined Sequence Diagrams for the BPL

2.4 Algebraic Specification

From UML2.0 Combined SDs, an algebraic representation can easily be obtained. Combined SDs can be considered as expressions on basic SDs composed by interaction operators [6]. We call these expressions References Expressions for SD.

Definition 1. *A* reference expression *for sequence diagrams (noted RESD hereafter) is an expression of the form*[1]*:*

```
<RESD>::=<PRIMARY> ( "alt" <RESD> |"seq" <RESD>)*
<PRIMARY>::=E∅ | <IDENTIFIER> | "("<RESD>")" |
          "loop" "(" <RESD> ")"
<IDENTIFIER>::= (["6","a"-"z","A"-"Z"]|["0"-"9"])*
```

`seq`, `alt` *and* `loop` *are the SD operators mentioned above.* $E_\emptyset$ *is the empty expression that defines a Sequence Diagram without interaction.*

So far, this algebraic framework does not contain means to specify variability. We introduce three algebraic constructs that correspond to the three variability

[1] We use a notation close to EBNF (Extended Backus-Naur Form) to define reference expressions.

mechanisms presented above. This allows defining optional, variation and virtual expressions.

Definition 2. *The optional expression (OpE) is specified in the following form:*
`OpE ::= "optional" <IDENTIFIER> "[" <RESD> "]"`
where `<IDENTIFIER>` *refers to the name of the optional part and the* `<RESD>` *refers to its corresponding expression.*

The tagged value optionalPart in UML2.0 SD specifies the name of the optional part in the optional expression. For the BSF example, optionality of the interaction `SetLimit` is specified by the expression: **optional** `settingLimit` **[SetLimit]**

Definition 3. *A Variation expression (VaE) is defined as follows:*
`VaE::= "variation" <IDENTIFIER> "[" <RESD> "," ( <RESD>)* "]"`

For example, the variation interaction `Withdraw` in Figure 2 encloses two interaction variants. It is specified algebraically as follows:
variation `Withdraw [ WithdrawWithLimit, WithdrawWithoutLimit ]`

Definition 4. *Virtual expressions (ViE) are specified as:*
`ViE ::= "virtual" <IDENTIFIER> "[" <RESD> "]"`

The SD `BPLPortion` of Figure 2 can be algebraically represented by the following expression:

$E_{BPLPortion}$ = **loop**(`Deposit` **alt** `CreateAccount` **seq** `(CreateAccountOk` **seq** **(optional** `settingLimit [ SetLimit ])` **alt** `CreateAccountFailed)` **alt** **variation** `withdrawAccount [ WithdrawWithLimit, WithdrawWithoutLimit ]` **seq** `( WithdrawOk` **alt** `WithdrawFailed))`

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

```
<RESD-PL>::=<PRIMARY-PL> ( "alt" <RESD-PL> | "seq" <RESD-PL>)*
<PRIMARY-PL>::=E∅ | <IDENTIFIER> | "("<RESD-PL>")" |
                 "loop" "(" <RESD-PL> ")" | VaE | OpE |ViE
```

3 Synthesizing Products Behaviors

In the previous Section, we have specified PL behavioral requirements using scenarios represented as UML2.0 SDs enriched with variability mechanisms. Scenarios are not the only way to describe software behaviors, statecharts [8], for example, are another formalism that is often used to depict the behavioral aspect of systems. However, if scenarios capture requirements in the early stage of the development process, statecharts models are more dedicated to detailed design phases as they are closer to an implementation (some tools such as Rhapsody [11] generate code from them). Furthermore scenarios and statecharts differ

on their nature: scenarios capture interactions between a *set of objects*, and statecharts represent the internal behavior of a *single object*. Statecharts synthesis out of a collection of scenarios has received a lot of attention in the context of single products development [14, 15, 17, 22]. 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. Variability is resolved by deriving the PL-RESD 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.

3.1 Product Expressions derivation

The first step towards product behaviors synthesis is to derive the corresponding product expressions from PL-RESD. As shown previously, PL-RESDs include a set of variation points. Derivation needs some decisions (or choices) associated to these variation points to produce a specific product RESD. A *decision model* is used to capture and record decision resolution associated to each product member in the PL.

Definition 5. *A decision model (noted hereafter DM) for a product P is a set of pairs* `(name`$_i$`, Res)`*, where* `name`$_i$ *designates a name of an optional, variation or virtual part in the PL-RESD and* `Res` *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 E_1, E_2, E_3.. as expression variants, the resolution is* `i` *if E_i is the selected expression.*
- *The resolution of a virtual part is a refinement expression* `E`*.*

For the derivation of products expressions from the `BPLPortion` of the BPL example, one needs decision resolutions for the optional expression `settingLimit` and for the variation expression `Withdraw`. The `BS1` product supports limit on accounts. This requires the presence of the `SetLimit` SD and the choice of the `WithdrawWithLimit` SD variant which is the first variant expression. So, the `BS1` product decision model is:
`DM1 ={(settingLimit, TRUE), (Withdraw, 1)}`. The decision model for the `BS2` product is: `DM2 ={(settingLimit, FALSE), (Withdraw, 2)}`

The derivation can be seen as a model specialization through abstract interpretation of a generic PL expression `PLE` in DMi context, where DMi is the decisions 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 not in the product expression. This is defined as:

   $$[\![\textbf{ optional } name\ [\ \texttt{E}\ ]\ ]\!]_{DMi} = \begin{cases} \texttt{E} & \text{if } (name, TRUE) \in DMi \\ \texttt{E}_\emptyset & \text{if } (name, FALSE) \in DMi \end{cases}$$

   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 seq $E_\emptyset$ = E ; $E_\emptyset$ seq E = E
- E alt $E_\emptyset$ = E ; $E_\emptyset$ alt E = E
- loop ($E_\emptyset$) =$E_\emptyset$.

This allows us to replace a complete part of a PL-RESD by $E_\emptyset$ when this part should be removed.

2. Interpreting a variation expression means choosing one expression variant among its possible variants. This is defined as:
   $[\![$ **variation** *name* [E_1, E_2, ..] $]\!]_{DMi} = E_j$ *if* $(name, j) \in DMi$
3. Interpreting virtual expressions means replacing the virtual expression by another expression:
   $[\![$ **virtual** *name* [E] $]\!]_{DMi}$ = E' *if* $(name, E') \in DMi$, E *otherwise*

The BS1 product expression E_{BS1} is obtained by the interpretation of the $E_{BPLPortion}$ in the DM1 context: $E_{BS1} = [\![E_{BPLPortion}]\!]_{DM1}$
The derivation of the BS1 product with a decision model given by context DM_1 produces the following expression :

E_{BS1} = **loop**(Deposit **alt** CreateAccount **seq** SetLimit **seq**
(CreateAccountOk **alt** CreateAccountFailed **alt**
WithdrawWithLimit **seq** (WithdrawOk **alt** WithdrawFailed))

The BS2 product does not provide overdrawing on accounts, which will be characterized by the absence of the SetLimit SD in the derived expression, and by the choice of SD WithdrawWithoutLimit at variation point Withdraw. The product expression obtained for product BS2 is:

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

3.2 Statecharts Generation

The derived product expression are expressions without variability, i.e expressions that only compose basic SDs by interaction operators: alt, seq, and loop. The second step of our synthesis approach aims at generating statecharts for objects in each derived product at the detailed design level. Product scenarios are translated into statecharts using the method proposed in [25].

We generate flat statecharts, i.e. statecharts without hierarchy. Figure 3 shows examples of flat statecharts, in which states represented by double circled states are called junction states. Junction states will have an additional role during statechart composition. Transitions are labelled 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 statechart $ST_\emptyset$ in Figure 3).

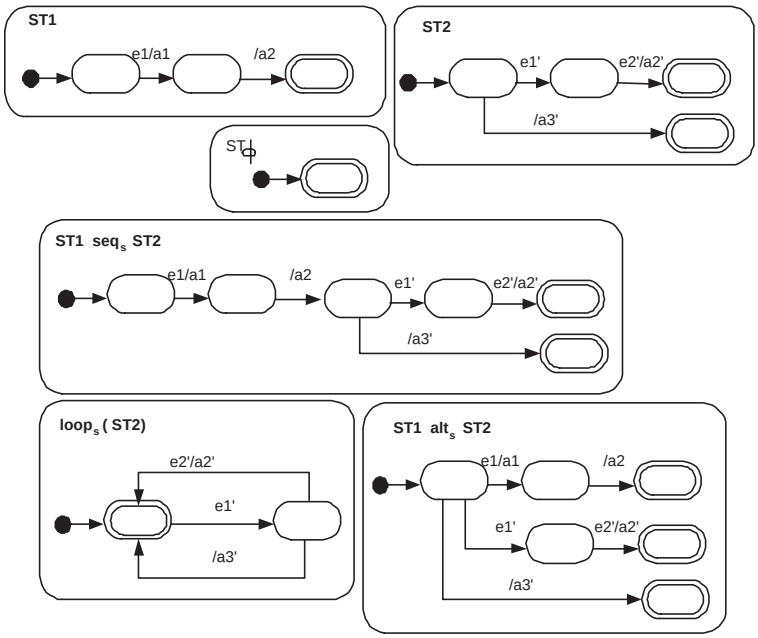

Fig. 3. Flat statecharts

Statecharts operators. Our algebraic framework for statecharts composition is inspired from the algebraic composition of UML2.0 sequence diagrams. We have formalized three statechart operators: $\texttt{seq}_s$, $\texttt{alt}_s$ and $\texttt{loop}_s$ respectively for the sequential composition, the alternative and the iteration of statecharts. We briefly describe these operators in the rest of this section, the complete formalization can be found in [25]:

Sequence ($\texttt{seq}_s$). 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 3 shows the sequential composition of the `ST1` and `ST2`.

Alternative ($\texttt{alt}_s$). The statechart resulting from the alternative composition describes a *choice* between the behaviors of its operands. See for example `ST1` alt_s `ST2` in Figure 3.

Loop ($\texttt{loop}_s$). This operator defines *iteration* of a statechart. Figure 3 shows the iteration of the `ST2`.

As for sequence diagrams, we algebraically describe statecharts composition with reference expressions.

Definition 6. *A* Reference expression *for statecharts (noted REST hereafter) is an expression of the form:*
<REST>::=<PRIMARY-REST> ("$\texttt{alt}_s$" <REST> | "$\texttt{seq}_s$" <REST>)*

```
<PRIMARY-REST>::=ST_∅ | <IDENTIFIER> | "("<REST>")"
        | "loop_s" "(" <REST> ")"
```

Generation process. Using our algebraic framework for statecharts, translating product UML sequence diagrams to statecharts can easily be defined in two steps. First flat statecharts are generated from basic sequence diagrams and then product RESD is mapped to RESTs:

Basic sequence diagrams. The first step of our synthesis algorithm is to generate a statechart $P(S, O)$ depicting the behavior of O in S for each object O and each SD S in the system. We do not detail here the algorithm computing $P(SD, O)$, which can be found in [25]. To summarize, this algorithm is a projection of SDs on object lifelines. Receptions in the SD become events in the statechart and emissions become actions. For a transition associated to a reception, the action part will be void, and for transitions associated to actions, the event part will be empty. The generated statechart contains a single junction state that 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 4 illustrates the synthesis of the statechart associated to the `Bank` from the `Deposit` basic SD.

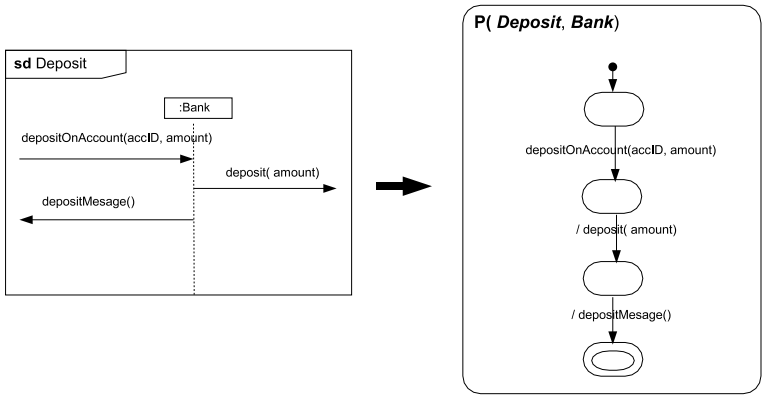

Fig. 4. Statechart synthesis from basic SD

Combined sequence diagram. Once we have obtained a collection of statecharts through projections of basic SDs, we can combine them with the same algebraic operators used for SD reference expressions. For each object O, a REST is constructed by replacing in the RESD `seq`, `alt`, and `loop` respectively by statecharts operators $\texttt{seq}_s$, $\texttt{alt}_s$, and $\texttt{loop}_s$, and each reference to a 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 BS1 product. The Bank's REST, called $\texttt{REST}_{BS1}$ is described below. Figure 5 shows the statechart obtained from this REST.

$\texttt{REST}_{BS1}$ = $\textbf{loop}_s$(P(Deposit, Bank) $\textbf{alt}_s$ P(CreateAccount, Bank) $\textbf{seq}_s$ P(SetLimit, Bank) $\textbf{seq}_s$ (P(CreateAccountOk, Bank) $\textbf{alt}_s$ P(CreateAccountFailed, Bank) $\textbf{alt}_s$ P(WithdrawWithLimit, Bank) $\textbf{seq}_s$ (P(WithdrawOk, Bank) $\textbf{alt}_s$ P(WithdrawFailed, Bank)))

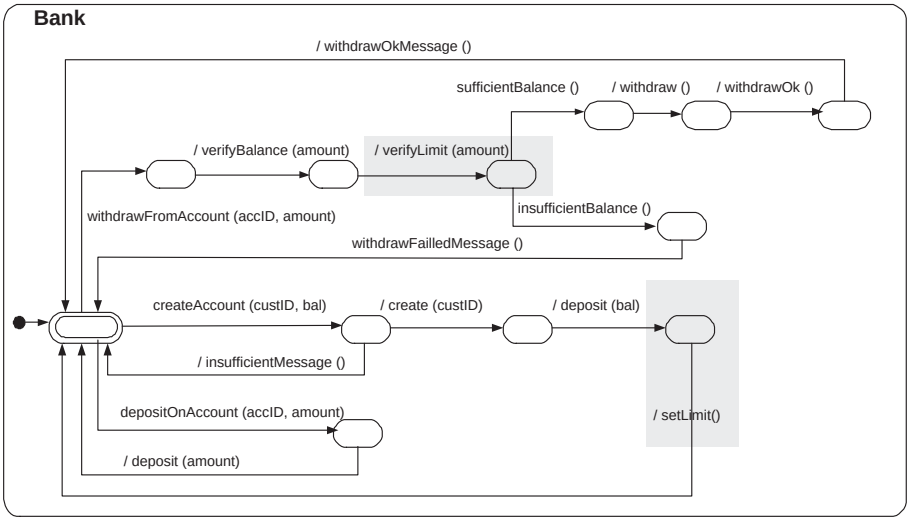

Fig. 5. The Bank Statechart in the BS1 product

The same method can be applied for the BS2 product. An expression $\texttt{E}_{BS2}$ is produced from the generic expression, and then transformed into the statechart composition expression $\texttt{REST}_{BS2}$ defined below. Figure 6 shows the Bank statechart obtained from $\texttt{REST}_{BS2}$. Note that as BS1 and BS2 only differ on the presence or not of an overdrawing limit, the synthesized statecharts will be very similar, and differ only on some transitions. The differences between the statecharts obtained for product BS1 and BS2 are illustrated in Figure 5 by grey zones.

$\texttt{REST}_{BS2}$ = $\textbf{loop}_s$(P(Deposit, Bank) $\textbf{alt}_s$ P(CreateAccount, Bank) $\textbf{seq}_s$ (P(CreateAccountOk, Bank) $\textbf{alt}_s$ P(CreateAccountFailed, Bank) $\textbf{alt}_s$ P(WithdrawWithoutLimit, Bank) $\textbf{seq}_s$ (P(WithdrawOk, Bank) $\textbf{alt}_s$ P(WithdrawFailed, Bank)))

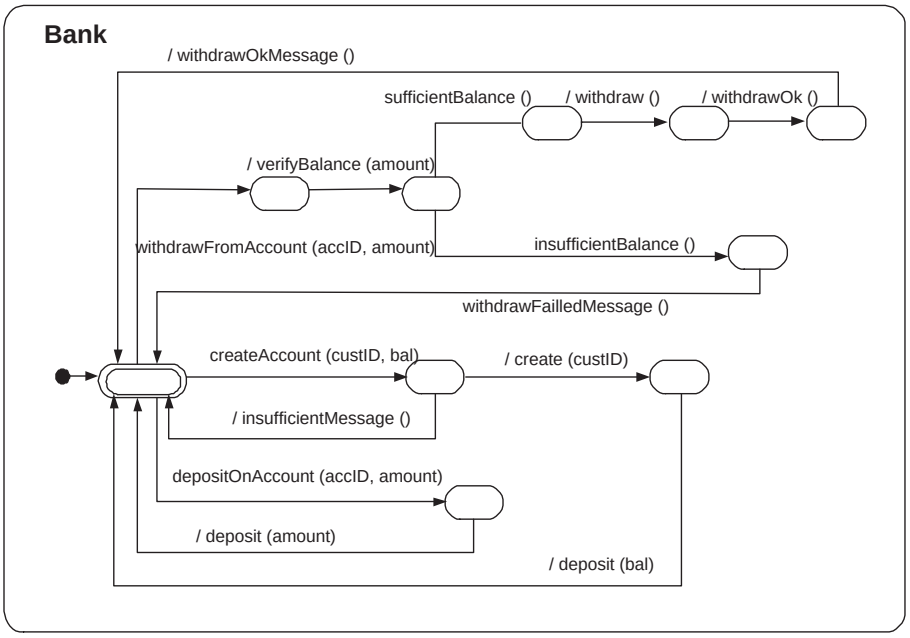

Fig. 6. The `Bank` Statechart in the `BS2` product

4 Implementation and discussion

A prototype tool of the proposed approach has been implemented in Java. PL-RESDs and decision models are specified in textual formats. The prototype implements a product expressions derivation from PL-RESD expression according to a given decision models. A Statechart for a specific object is generated from the derived expression and basic interactions which are also specified in textual format [12]. A more complete description of this prototype can be found in [23]. We have used our approach for a complete BPL case study with fourteen basic SDs. Table 2 shows statistics (number of states and transitions) on the generated statecharts for the Bank object in each BPL member.

Table 2. States and transitions for the generated `Bank` statechart

	# States	# Transitions
`BS1`	12	16
`BS2`	10	14
`BS3`	13	19
`BS4`	15	21

A Flexible Approach. Defining statecharts synthesis from UML2.0 SDs as a mapping from RESD to RESTs gives a certain flexibility to the synthesis process: any modification (adding or removing a SD for example) of the RESD only lightly influences the synthesis process. It is only sufficient to modify (adding or removing the corresponding statechart) the RESTs, thus fostering a better traceability between the requirements and the detailed design. To illustrate this, let us consider again the BS1 product with a new functionality for currency exchange calculations. This is described by the three new basic SDs shown in Figure 7. The new BS1 RESD is obtained from the older one by adding references to `SetCurrency`, `ConvertToEuro` and `ConvertFromEuro`:

$\texttt{REST}_{BS1}$ = **loop**(Deposit **alt** CreateAccount **seq** (CreateAccountOk **seq** SetLimit *seq SetCurrency* **alt** CreateAccountFailed) **alt** WithdrawWithLimit **seq** (WithdrawOk **alt** WithdrawFailed) *alt ConvertToEuro alt ConvertFromEuto)*

The new Bank's REST is obtained from the older one by adding the synthesized statecharts from the three basic SDs. We keep the same composition information added in the new BS1 RESD:

$\texttt{REST}_{newBS1}$ = $\textbf{loop}_s$(P(Deposit, Bank) $\textbf{alt}_s$ P(CreateAccount, Bank) $\textbf{seq}_s$ P(SetLimit, Bank) seq_s *P(SetCurrency, Bank)* $\textbf{seq}_s$ (P(CreateAccountOk, Bank) $\textbf{seq}_s$ $\textbf{alt}_s$ P(CreateAccountFailled, Bank)) $\textbf{alt}_s$ P(WithdrawWithLimit, Bank) $\textbf{seq}_s$ (P(WithdrawOk, Bank) $\textbf{alt}_s$ P(WithdrawFailled, Bank)) alt_s *P(ConvertFromEuro, Bank)* alt_s *P(ConvertToEuro, Bank))*

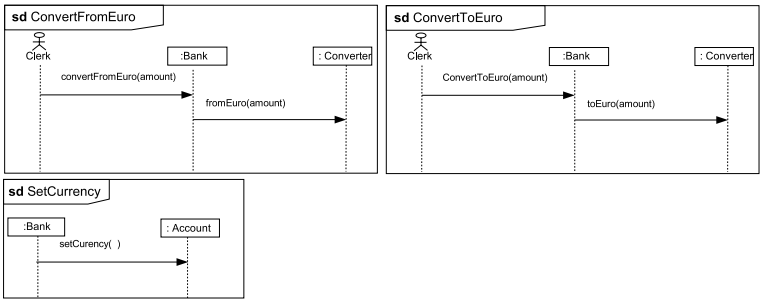

Fig. 7. Basic SDs for currency exchange calculations

PL Engineering process. The proposed approach can easily be integrated into the general PL process [18]. It fulfills two important objectives in PL: Domain

engineering and Application Engineering [4]. The integration of variability into scenarios with PL-RESD allows for the definition of generic requirements, which brings a new contribution to domain engineering. Derivation of a specific product and then of specific statecharts is a step towards detailed design phases. Standard approaches such as [9] can be used to generate applications from the synthesized statecharts. As a part of synthesis can be reused during statechart generation, our approach clearly deserves reuse in application engineering.

5 Related work

This section briefly compares our work with other approaches related to variability integration in requirements, and to statechart synthesis from scenarios.

Requirements Modeling in Product Lines. Few approaches model variability in requirements using scenarios. Gomaa [5] introduces variability in UML collaboration diagrams with three stereotypes `<<kernel>>`, `<<optional>>` and `<<variant>>`. These stereotypes are also defined for use cases and class diagrams. While we explicitly formalize the derivation process, Gommaa et al do not describe how the introduced stereotypes are used to derive products architectures. *Atkinson et al.* [1] introduces the stereotype `<<variant>>` which can be applied to messages in sequence diagrams and to statecharts. In our approach, variability is only introduced in scenarios which are more close to users understanding than statecharts. Most approaches on PL requirements rely on Use Cases rather than on scenarios to formalize PL requirements including variability. *Halmans et al.* [7] presents a detailed study on requirements engineering for product lines, and extends Use Cases with stereotypes to specify variability. Use Cases are described using templates. Bertolino [2] introduces tags to describe variability in a textual description of uses cases. Massen [21] extends the UML Use Case meta-model to allow variability. John [13] tailors Use Case diagrams and textual use cases to support PL requirements specification. Even if the textual description through templates, used by the previous work, is a good way to document PL requirements, sequence diagrams are more operational and as shown with our approach detailed design can be generated from them. *Haugen et al.* [10] also use UML2.0 sequence diagrams to specify requirements. They introduce a new operator called `xatl` to distinguish between mandatory and potential behaviors. A potential behavior represent a variant of a mandatory behavior. This is close to our `variation` construct where interaction variants correspond to the potential behaviors.

Statecharts Synthesis from scenarios for single products. Several approaches for Statechart synthesis from scenarios have been proposed this last decade. This section gives a brief overview of some of them. Note however that all of these approaches are dedicated to synthesis for a single product, and do not consider synthesis for several products. Due to the poor expressive power of UML1.x sequence diagrams, the proposed solutions for statecharts synthesis [14, 15, 17, 22] often use additional information or ad hoc assumptions for

managing several scenarios. For example, *Whittle et al.* [22] enriches messages in sequence diagrams with pre and postconditions given in OCL (Object Constraints Language) 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 statecharts and transitions thanks to information provided by lifeline orderings and SD operators. *Koskimies et al.* [15] uses Biermann-Krishnaswamy algorithm [3] which infers programs from traces. This work establishes a correspondence between traces and scenarios and between programs and statecharts. In [17, 14] it is also proposed to use interactive algorithms to generate statecharts from UML1.x sequences diagrams. Several other approaches [19, 20, 16] study state machines synthesis from Message Sequence charts (MSC) [12], 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 current SD in UML 2.0 and our approach can be used to generate statecharts from MSCs.

6 Conclusion

In this paper we have proposed an approach to derive product behaviors from PL requirements. Firstly algebraic construct are introduced to specify variability in UML2.0 sequence diagrams. Then, we use interpretations of the algebraic expressions to resolve variability and derive product expressions. The derived expressions are then transformed into a set of statecharts. The introduction of variability can be used to factorize common behaviors in different products, and should then facilitate domain engineering phases. As discussed in [25], statecharts synthesis should be more considered as a step towards implementation rather than as a definitive bridge from user requirements to code. However, some parts of the synthesis can be reused from a product to another, hence facilitating reuse during application engineering.

References

1. 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*. Component Software Series. AW, 2001.
2. A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use Case Description of Requirements for Product Lines. In *Requirement Engineering for Product Lines (REPL02)*, pages 12–18, September 2002.
3. A.W Biermann and Krishnaswamy.R. Constructing programs from example computations. *IEEE Transaction Software Engineering*, 2(3):141–153, September 1976.
4. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. AW, 2000.
5. H. Gomaa. Modeling Software Product Lines with UML. In P. Knauber and G. Succi, editors, *SPLW2*, pages 27–31, Toronto, Canada, May 2001. IESE. IESE-Report. No. 051.01/E.
6. Object Management Group. Uinified modeling language specification version 2.0: Superstructure. Technical Report pct/03-08-02, OMG, 2003.

7. G. Halmans and K. Pohl. Communicating the variability of a software-product family. *Software System Model*, 3:15–36, 2003.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
9. D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceeding of International Conference on Software Engineering (ICSE 1996)*, 1996.
10. O. Haugen and K. Stolen. STAIRS-Steps to Analyze Interactions with Refinement Semantics. In *<<UML2003>>-The Unified Modeling Language Conference*, pages 388–402, October 2003.
11. I-Logix. Rhapsody. http://www.ilogix.com/products/rhapsody/index.cfm.
12. ITU-T. Z.120 : Message sequence charts (MSC), november 1999.
13. I. John and D. Muthig. Tailoring Use Cases for Product Line Modeling. In *Requirement Engineering for Product Lines (REPL02)*, pages 26–32, September 2002.
14. I. Khriss, M. Elkoutbi, and R. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *Proceedings of UML'98: Beyond the Notation*, pages 115–126bis, 1998.
15. K. Koskimies, T. Systä, J Tuomi, and Männistö.T. Automated support for modeling oo software. *IEEE Software*, 15:87–94, Janu 1998.
16. I. Krger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.
17. E. Mäkinen and T. Systä. Mas-an interactive synthesizer to support behavioral modeling. In *Proceeding of International Conference on Software Engineering (ICSE 2001)*, 2001.
18. L.M. Northrop. A framework for software product line practice -version 3.0. Web http://www.sei.cmu.edu/plp/framework.html, Software Engineering Institute, 2002.
19. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *Proceeding of International Conference on Software Engineering (ICSE 2001)*, 2001.
20. S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transaction on Software Engineering*, 29(2):99–115, February 2003.
21. T. van der Maßen and H. Lichter. Modeling Variability by UML Use Case Diagrams. In *Requirement Engineering for Product Lines (REPL02)*, pages 19–25, September 2002.
22. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceeding of International Conference on Software Engineering (ICSE 2000)*, 2000.
23. T. Ziadi. Technical and additional material. http://www.irisa.fr/triskell/results/UML04/.
24. T. Ziadi, L. Hélouët, and J.M. Jézéquel. Toward a uml profile for software product lines. In *Proceedings of the Fifth Internationl Workshop on Product Familly Engineering (PFE-5)*, volume 3014 of *LNCS*. Springer Verlag, 2003.
25. T. Ziadi, L. Hélouët, and J.M. Jézéquel. Revisiting statecharts synthesis with an algebraic approach. In *International Conference on Software Engineering, ICSE'26, Edinburgh, Scotland, United Kingdom*, May 2004.