

# PLiBS: an Eclipse-based tool for Software Product Line Behavior Engineering

Tewfik ZIADI  
LIP6 & University of Paris 6

[Tewfik.Ziadi@lip6.fr](mailto:Tewfik.Ziadi@lip6.fr)

Jean-Marc JEZEQUEL  
IRISA (INRIA & University of Rennes1)

[Jezequel@irisa.fr](mailto:Jezequel@irisa.fr)

## ABSTRACT

This paper presents the PLiBS (Product Line Behavior Synthesis), an Eclipse-based tool for modeling and deriving behavior aspects in Software Product Lines (SPL). PLiBS allows specifying SPL behaviors using UML2 sequence diagrams extended by variability mechanisms. PLiBS implements a two-step approach to derive product behaviors. The first one uses model transformation to specialize sequence diagrams. While the second one uses UML state machines synthesis from sequence diagrams. This paper presents a guided tour around PLiBS. It presents its architecture and its main components. It also illustrates its uses on the Banking Product Line example.

## Keywords

Software product line, variability, sequence diagrams, state machines, synthesis.

## 1. INTRODUCTION

Software Products Lines (SPL) aim at decreasing development costs and times by developing a family of systems instead than one system at a time. Rather than describing a single software system, the model of a software product line 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 product line [19]. Beyond variability modeling, the *product derivation* process is defined as a complete process of constructing products from the software product line [12].

While many works have investigated the modeling and derivation of functional [4,10,13,16] and static [6,14, 18] aspects of SPL, there is few research on supporting behavior modeling of SPL [3,8, 9,16] and none for related product.

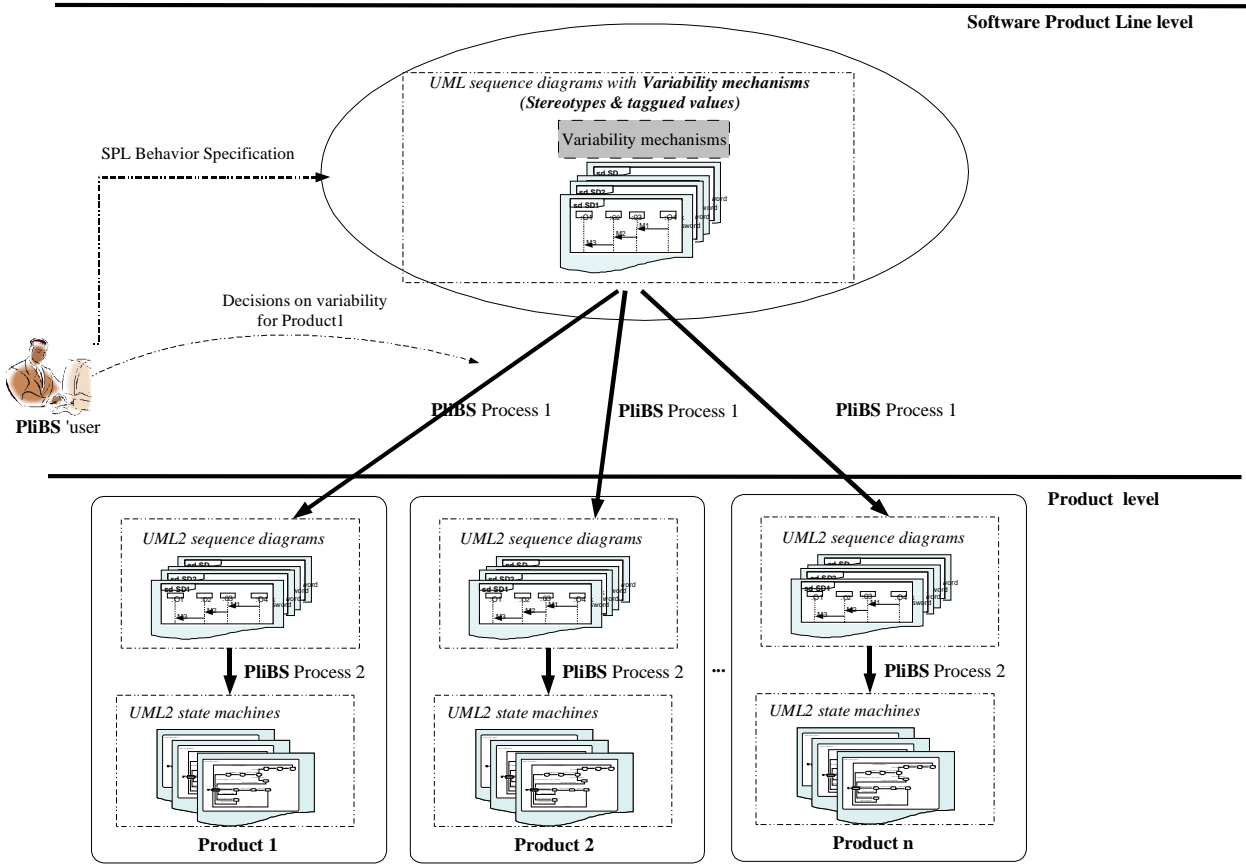
Behavior modeling plays an important role in the traditional engineering of software-based systems; it is the basis for systematic approaches to requirement captures, design, simulation, code generation, testing and verification. Two complementary formalisms for modeling behavior have proven

useful in practice: interaction-based (focusing on the global interactions between actors and components, e.g.; UML2 sequence diagrams) and state-based modeling (concentrating on internal states of individual components, e.g.; UML2 state machines). In the context of the Families project [1], we have implemented an Eclipse-based tool, called PLiBS (Product Line Behavior Synthesis) for SPL behavior engineering. The approach implemented in PLiBS is based on these two formalisms. UML2 sequence diagrams are firstly used to model SPL behaviors. While UML state-machines represent the results of the PLiBS derivation process.

The rest of the paper is organized as follows: Section 2 summarizes the approach that PLiBS implements. Section 3 presents the PLiBS architecture and its main components. Section 4 illustrates the use of PLiBS on the Banking Product Line example. Finally Section 6 concludes this work and dresses some perspectives.

## 2. THE APPROACH

It's admitted in the literature that software product line engineering defines two mains levels: 'Product Line' and 'Single Product'. At the first one, the architecture of the SPL is defined and the variability is explicitly specified. At the second one, the specified variability is resolved and the SPL architecture is specialized to obtain product-specific architectures. The approach implemented in the PLiBS tool (cf. Figure 1) uses this level duality to model the behavior aspect of SPL architecture. Behavior modeling plays an important role in the traditional engineering of single software-based systems; it is the basis for systematic approaches to requirement captures, design, simulation, code generation, testing and verification. Two complementary formalisms for modeling behaviors have proven useful in practice: interaction-based (focusing on the global interactions between actors and components, e.g.; UML2 sequence diagrams) and state-based modeling (concentrating on internal states of individual components, e.g.; UML2 state machines). Our approach is based on these two formalisms. UML2 sequence diagrams are used at the "Product Line" level to model SPL behaviors. While UML state-machines represent the results of our



**Figure 1.** Our approach for modeling and deriving behaviors of software product lines.

process derivation; they specify the product-specific behaviors. Our initial objective in this paper is to present the tool that implements the approach and not the approach itself. The approach is previously published in [22]. So in this section we only summarize main steps of our approach. We firstly describe how UML2 sequence diagrams are used to model SPL behaviors. Then we show how product-specific UML state-machines are derived.

## 2.1 Modeling Software Product Line Behavior

Our approach uses UML2 sequence diagrams. UML2 mainly improves UML1.x sequence diagrams in by the introduction of interaction operators to compose sequence diagrams. **seq** (for sequence composition), **alt** (for alternative composition), and **loop** (for the iteration of a SD) are examples of the UML2 interaction operators. The SPL behavior is specified in our approach as a collection of basic SD describing basic behaviors that concern all the product line and one combined SD. A combined SD is the name used within UML2 to call composed SD. It refers to a collection of basic SD and composes them using interaction operators.

To be useful in the SPL context, sequence diagrams should allow the expression of variability. As shown in [20], variability can be introduced in UML2 combined SD using UML extension mechanisms such as stereotypes and tagged values. This

includes three mechanisms: optionality, variation and virtuality. **Optionality** of a SD means that this SD can be omitted in some products. **Variation** means that the SD defines a set of SD variants and each product should choose one and only one SD variant. **Virtuality** is inspired by an existing mechanism in MSC [12]. The virtual SD will be replaced by another SD of refinement. Each product defines its SD refinement. The complete formalization of these mechanisms on the UML2 meta-model can be found in [20].

Our manipulation of SPL sequence diagrams is based on an algebraic framework for UML2 sequence diagrams [22]. Each UML combined sequence diagram is algebraically formalized as a RESD (Reference Expression for Sequence Diagrams). RESD is an expression where terms are references for basic SD and operators are SD operators (**seq**, **alt**, and **loop**). PL combined sequence diagram is formalized as a PL-RESD (Product Line-RESD). PL-RESD extends RESD by the introduced variability construction (**optional**, **variation**, and **virtual**). In what follows, we only present definitions of RESD and PL-RESD. Section 5 illustrates them on a concrete example.

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

$$\langle \text{RESD} \rangle ::= \langle \text{PRIMARY} \rangle ( \text{"alt"} \langle \text{RESD} \rangle \mid$$

```

                "seq" <RESD>)*
<PRIMARY> ::= E∅ | <IDENTIFIER> |
" (" <RESD> )"
                "loop" " (" <RESD> )"
<IDENTIFIER> ::= ( [ "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.

**Definition 2.** A reference expression for PL-sequence diagrams (noted PL-RESD hereafter) is an expression of the form:

```

<PL-RESD> ::= <PRIMARY-PL> ( "alt" <PL-RESD> |
                "seq" <PL-RESD> ) *
<PL-PRIMARY> ::= E∅ | <IDENTIFIER> |
                " (" <PL-RESD> )" |
                "loop" " (" <RESD-PL> )" |
"optional" <IDENTIFIER> " [" <RESD> "]" |
"variation" <IDENTIFIER>
                " [" <RESD> ", "( <RESD> ) * "]" |
"virtual" <IDENTIFIER>
                " [" <RESD> "]"

```

**optional**, **variation** and **virtual** are constructions that correspond to the three variability mechanisms introduced above.

## 2.2 Deriving Product-Specific Behavior

Deriving product-specific behaviors is realized within PLiBS in two steps. First, variability is resolved by deriving the PL-RESD into a set of RESD, one for each product. Then state machines are generated by transforming product sequence diagrams given as an RESD into a composition of UML2 state machines.

### 1. PLiBS process 1:

The first PLiBS process (cf. Figure 1) consists in generating the combined sequence diagram, in the form of an RESD, of each product from the PL-RESD. As mentioned previously, the PL-RESD is defined by a set of variability constructions. To derive a product-specific RESD, some decisions (or choices) associated with these variation points are needed. For example, each product should choose among the presence or not of all optional SD. A mechanism is needed to capture decisions that are made for a specific product. In our approach, we call this mechanism *Instance Decision Model (IDM)*. Each IDM captures decision resolutions for one product.

**Definition 2.** An Instance of Decision Model (noted hereafter *IDM*) for a product  $\mathcal{P}$  is a set of pairs  $(name_i, Res)$ ,  $name_i$  designates a name of an optional, variation or virtual part in the PL-RESD and  $Res$  is its decision resolution related to the product  $\mathcal{P}$ . 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  $E$ .

The first step is formalized through abstract interpretation of a generic SPL expression in the  $IDM_i$  context, where  $IDM_i$  is the Instance of Decision Model related to a specific product:

$$E_{\text{Product}} = \llbracket E_{\text{SPL}} \rrbracket_{IDM}$$

Interpretation of the  $E_{\text{SPL}}$  is based on the interpretation of each variability mechanism. Interpreting an optional expression for example, means deciding on its presence or absence in the product-specific expression  $E_{\text{Product}}$ . This can be formalized as follows:

$$\llbracket \text{optional name} [ E ] \rrbracket_{IDM_i} = \begin{cases} E \text{ if } (\text{name}, \text{TRUE}) \in IDM_i \\ E_{\emptyset} \text{ if } (\text{name}, \text{FALSE}) \in IDM_i \end{cases}$$

### 2. PLiBS process 2 :

The obtained product expressions from PLiBS process 1 are expressions without variability, i.e. expressions that only compose basic SDs by interaction operators: **alt**, **seq**, and **loop**. The second PLiBS process (cf. Figure 1) of our derivation approach aims at generating UML2 state machines for objects in each derived product. State machines synthesis out of a collection of sequence diagrams has received a lot of attention in the context of single product development. This allows fostering a better traceability between the requirements (depicted by sequence diagrams) and detailed design (depicted by state machines). In our approach, product sequence diagrams are translated into state machines using the method that we proposed in [21]. Our synthesis method generates state machines from all basic sequence diagrams in the product line and then composes them to obtain a full state machine for each object. To realize this composition, our method formalized three state machines operators (for sequence, alternative, and iteration composition). The composition is then defined using a correspondence between interaction operators and state machine operators. For example, if two basic SD are sequentially composed in the RESD, so the two generated state machines from these two basic SD are sequentially composed to obtain the full state machine. Interested readers can refer to [21] for a full description of our synthesis method.

## 3. THE PLiBS TOOL

PLiBS (Product Line Behavior Synthesis) is an Eclipse-based tool that implements the approach presented in Section 2. PLiBS is entirely implemented in Java and it uses the EclipseUML tool [23]: a UML tool integrated in the Eclipse platform. The EclipseUML tool is firstly used within PLiBS to specify PL UML2 sequence diagrams and secondly to display the produced UML2 state machines. This Section presents our extensions to

the EclipseUML to specify PL sequence diagrams, describes the PLiBS architecture and its main components. It finally discusses the implementation choices we made.

### 3.1 PL Sequence diagrams specification

Using PLiBS, PL sequence diagrams are specified using the EclipseUML tool [23]. Unfortunately, this tool only allows specifying basic sequence diagrams (see Figure 2) and no means to specify combined SD (i.e.; SD with interaction operators). To go beyond this limit, we propose a set of conventions to specify in the EclipseUML tool the main concepts of combined SD. This includes conventions to specify references to basic SD, interaction operators (sequence, alternative, and loop), and variability in PL SD.

#### Reference (ref)

Refence A combined SD in the PLiBS is specified as an EclipseUML SD including references. Reference to a basic SD is made by a “self message” event coupled with a note containing the name of the referenced basic SD. Figure 5 shows an example of a combined SD with a set of references. The self message and the note containing the name “Deposit” for example specifies that this combined SD refers to a SD called “Deposit”.

#### Sequence (seq)

The sequence between basic SD is implicitly specified by the order of appearance of these basic SD references. In the combined SD of Figure 5, the basic SD Deposit is referenced before the SD CreateAccount. This is equivalent to the expression `Deposit seq CreateAccount`.

#### Alternative (alt)

The EclipseUML tool introduces a “switch” mechanism that can be specified on the lifeline of objects in sequence diagrams. We propose to reuse this mechanism to specify the **alt** operator, each “case” sub-statement represent a particular operand of the **alt** operator. In Figure 3. three operands are given for the alternative node. In each operand, a reference to a basic SD can be added (see Figure 5.)

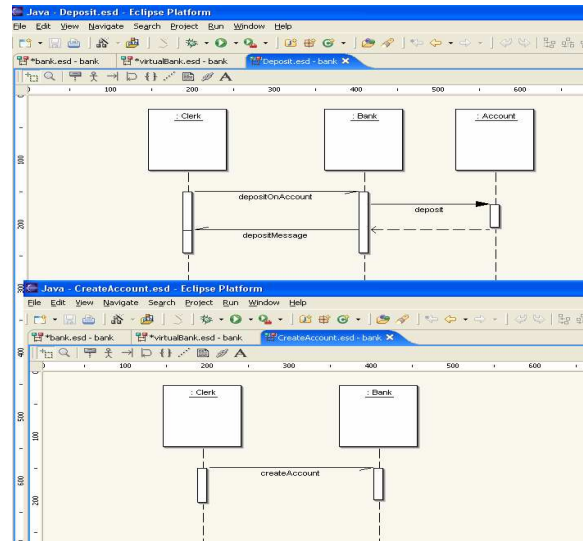


Figure 2. Basic sequence diagrams specification in PLiBS.

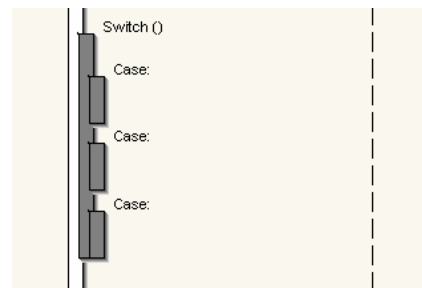


Figure 3. Alternative specification in PLiBS.

- **Loop**

The loop operator is introduced by a “do...while” or a “while” statement as shown below (see Figure 4)

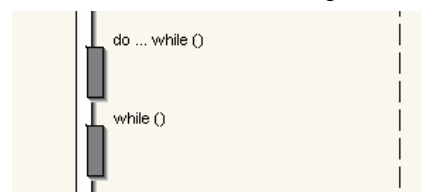


Figure 4. Loop definition in PLiBS.

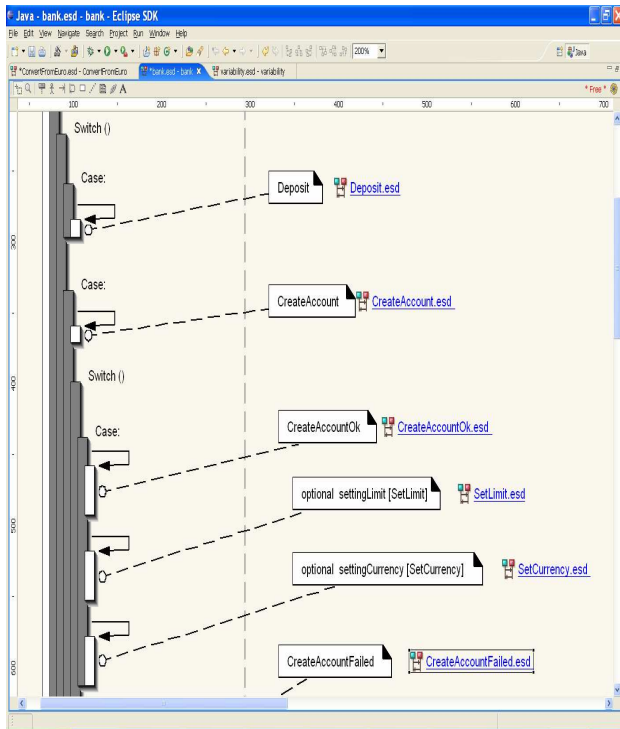


Figure 5. A portion of the Banking Product Line Combined SD including variability.

### Variability

As presented in Section 2, our extensions of UML2 sequence diagrams for variability, includes three mechanisms: optionality of interactions, variation and virtuality. Within the PLiBS tool, we propose three new keywords, one for each variability mechanism, which are displayed in UML notes associated with references in a PL combined SD. A reference to an optional sequence diagram is introduced with the **optional** keyword and the name of the SD follows between square brackets (see Figure 5). Variant SD references are introduced with the **variation** keyword. All variants appear between square brackets, each SD name separated by a comma. A virtual basic SD is specified in the same way as for optional SD (we only replace **optional** keyword by the **virtual** one). The combined SD of Figure 5 illustrates the specification of optionality. The SD SetLimit is optional; this is specified in the note associated with the SD SetLimit by the expression: optional settingLimit [SetLimit].

## 3.2 PLiBS architecture

Figure 6 shows the overall architecture of PLiBS. This includes six components:

**PL-RESD parser.** Is a parser for PL-RESD. The PL-RESD syntax is presented in Section 2.

**Basic SD parser.** Is a parser for basic sequence diagram syntax (the syntax used within PLiBS to describe basic SD is close to the MSC specification [12]).

**EclipseUML-PLiBS interface.** It extracts from the PL sequence diagrams, specified in the EclipseUML tool, the PL algebraic expression and the description of all basic SD. These outputs respectively represent inputs of the AE derivation component and the Basic SD parser.

**Algebraic Expression Derivation component (AE Derivation).** It implements the PLiBS process 1 (cf. section 2). Subsection 3.3 describes this component in more detail.

**RESD parser.** Is a parser for RESD. The RESD syntax is presented in Section 2.

**UML2 state machine synthesis component.** It implements the PLiBS process 2 (cf. section 2). This component is described in subsection 3.3

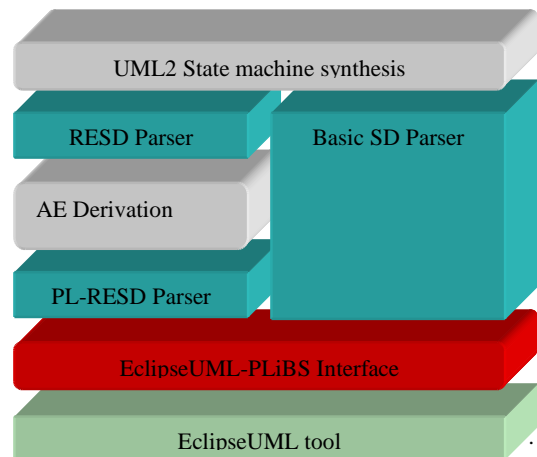


Figure 6. The PLiBS architecture

## 3.3 Main Components

There are two main components implementing the core functionalities in the PLiBS tool. The AE Derivation and the UML2 state machine synthesis components. Next paragraphs present these two components.

### 3.3.1 Algebraic Expression Derivation component

**Description:** This component uses a kind of a rewrite mechanism to implement the derivation of PL Algebraic Expression (AE). Based on instance decision models, the PL-RESD is specialized to obtain product-specific expressions. The component displays the PL-RESD obtained from the EclipseUML-PLiBS Interface component (see Figure 7). For decisions on variability, the AE derivation component displays to the user a UI that describes all variation points and their possible decisions. The user uses this UI to define here decision model instance. Figure 8 illustrates the instantiation of the optional parts. The TRUE panel for example contains all optional parts in the PL-RESD which are selected by the user.

**INPUT:** a PL-RESD (Product Line Reference Expression for Sequence Diagrams), a IDM (Instance of Decision Model)

**OUTPUT:** RESD (Reference Expression for Sequence Diagrams)

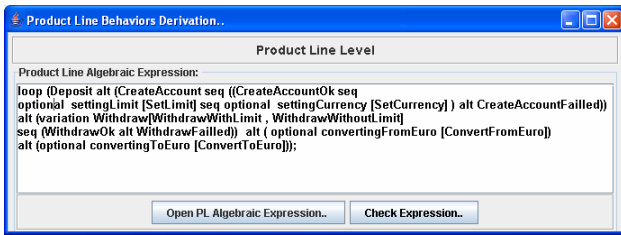


Figure 7. PL-RESD as displayed by the AE derivation component

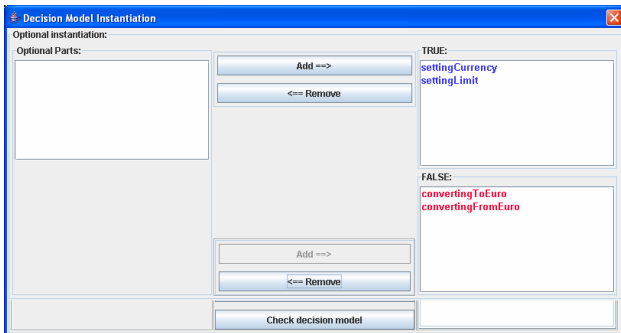


Figure 8. The instantiation of the optional parts.

### 3.3.2 UML2 state machine synthesis

**Description:** This component implements the PLiBS process 2 presented in Section 2. It implements the algorithm that generates UML2 state machines from basic sequence diagrams as formalized in [21]. It also implements state machines operator in order to obtain the full state machines [21]. Figure 9 shows the UI associated with this component. In the first panel, it displays the product-specific expression obtained from the AE Derivation component. The second panel contains the specification of all basic SD. The component generates the full state machine specifying the complete behavior of the selected object in the context of the product.

**INPUT:** the product-specific RESD, a specification of all basic SD.

**OUTPUT:** a UML2 state machine for each object participating in one or more basic SD.

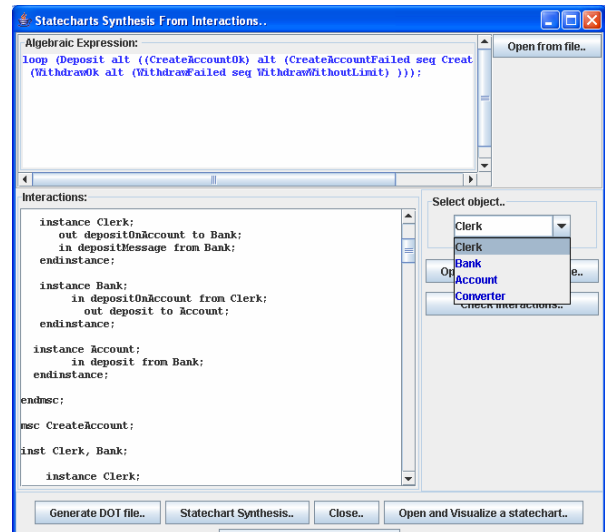


Figure 9. UML state machine synthesis

## 3.4 Discussion

The architectural choices that we made to implement PLiBS give two main advantages. First, the two components implementing the PLiBS core functionalities are independent from the EclipseUML tool. This facilitates integrating new UML tools. It is only sufficient to develop the component interface associated with the new UML tool. Second, the separation between PL expression derivation and UML2 state machines processes allows using PLiBS in the context of the single product development. In this context, the user only uses the UML2 state machine synthesis component.

Beyond implementing the main aspects of our approach, PLiBS lacks of PL constraints checking. Indeed, in addition to variability, the PL architecture is defined with a set of constraints. PL constraints mainly specify dependencies between variation points (requires or mutual exclusion are examples of these constraints). In [22], we formalized PL constraints as OCL meta-level constraints. In the future it will be interesting that PLiBS allow specifying and checking this kind of constraints. This can be realized by integrating in the PLiBS tool a checker of OCL meta-level constraints.

## 4. ILLUSTRATION: THE BANKING SPL

To illustrate the PLiBS use, we present the example of a Banking Product Line (BPL). 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 functionality concerns currency exchange: euros, dollars, etc.

Variability in the BPL example concerns the support of overdrawn 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. 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.

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

**Table 1. The Banking PL Members.**

Figure 5, shows the portion of the combined SD for the BPL as specified in PLiBS. The  $E_{BPL}$  expression, presented above, represents the PL-RESD extracted for the BPL SD thanks to the EclipseUML-PLiBS interface. It refers to the eleven basic SD. The basic SD `Deposit` for example is SD to deposit an amount on a specific account. Figure 2 shows the specification of this basic SD.

```

EBPL = loop ( Deposit
  alt (
    CreateAccount
    seq (
      CreateAccountOk
      seq optional settingLimit [ SetLimit ]
      seq optional settingCurrency [
        SetCurrency ] ) )
    alt CreateAccountFailed)
  alt (variation withdraw [ WithdrawWithLimit, WithdrawWithoutLimit ]
    seq (WithdrawOk alt WithdrawFailed))
  alt (optional fromEuro [ ConvertFromEuro ] )
  alt (optional toEuro [ ConvertToEuro ] ) )

```

The  $E_{BPL}$  illustrates two variability mechanisms: *optionality* and *variation*.

- Since some products of the BPL do not support overdrawn, optionality is added to the basic SD `SetLimit` (`SetLimit` is the basic SD that specifies the interactions between object for adding limits on account). In addition, since exchange calculation is an optional functionality in the BPL, basic SD `SetCurrency` (is the basic SD for adding currencies on account), `ConvertToEuro` and `ConvertFromEuro` are defined as optional (these two last basic SDs specify exchange calculations).
- 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 mechanism. The two SDs `WithdrawWithLimit` and `WithdrawWithoutLimit` are basic SD variants.

From the  $E_{BPL}$  and using the PE derivation component, four product expressions are derived, one for each banking product. The two expressions  $E_{BS1}$  and  $E_{BS2}$  below are respectively expressions obtained for the product BS1 and BS2 respectively. Using the UML2 State machine synthesis component, state machines can be synthesized for all objects participating in interactions. Figure 10, shows the state machine generated for the Bank object.

```

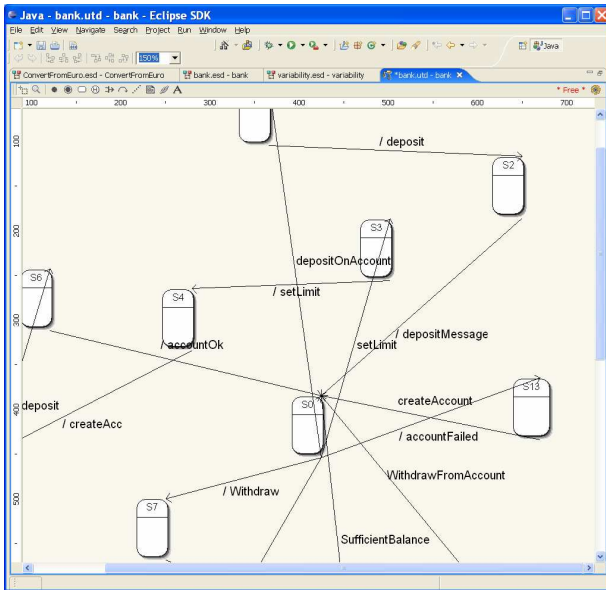
EBS1 = loop ( Deposit
  alt (
    CreateAccount
    seq (
      CreateAccountOk
      alt CreateAccountFailed)
    alt (WithdrawWithoutLimit
      seq (WithdrawOk alt WithdrawFailed)))

```

```

EBS4 = loop ( Deposit
  alt (
    CreateAccount
    seq (
      CreateAccountOk
      seq SetLimit
      seq SetCurrency ) )
    alt CreateAccountFailed)
  alt (WithdrawWithLimit
    seq (WithdrawOk alt WithdrawFailed))
  alt (ConvertFromEuro )
  alt ( ConvertToEuro ) )

```



**Figure 10. The synthesized state machine for the Bank object in the context of the BS4 product**

## 5. CONCLUSION & PERSPECTIVES

In this paper, we presented the PLiBS tool: an Eclipse-based tool for software product line behavior engineering implemented in the context of the Families project [1]. PLiBS allows software product line behavior specification using UML2 SD and implements a two-process approach to derive product behaviors. The first process is based on model specialization through algebraic expression interpretation and the second stage uses state machine synthesis. Within the PLiBS architecture the interface to the EclipseUML is independent from the core implementation. This facilitates integrating new UML tools. It is only sufficient to develop the component interface associated with the new UML tool. In addition, the separation in two independent components between PL expression derivation and UML2 state machines processes allows using PLiBS in the context of the single product development (without considering of variability). In this context, the user only uses the UML2 state machine synthesis component.

UML state machine are useful for code generation. Some tools such as Rhapsody [24] allow generating code from them. We currently working on integrating in PLiBS a code generator. This gives a good way for prototyping the derived products by generating code from product-specific state machines.

The PLiBS tool suffers from some disadvantages. It actually uses ad-oc conventions and notations to specify UML2 SD. This is due to the lack expressiveness of the EclipseUML sequence diagrams. In addition until now PLiBS do not support PL constraints checking. As a perspective, we want integrate an OCL checker. This allows specifying and checking PL constrains as formalized in [20,22].

## 6. REFERENCES

- [1] FAMILIES project, <http://www.esi.es/en/projects/families/>, 2003.
- [2] 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. 2001.
- [3] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practices. Addison-Wesley, first edition, 1998.
- [4] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use Case Description of Requirements for Product Lines. In International Workshop on Requirement Engineering for Product Line (REPL02), pages 12–18, September 2002.
- [5] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, and K. Pohl. Variability Issues in Software Product Lines. In Fourth workshop Product Family Engineering (PFE4), pages 11–19, 2001.
- [6] M. Clauß. Generic modeling using UML extensions for variability. In Workshop on Do main Specific Visual Languages at OOPSLA 2001, Tampa Bay, Florida, USA, 2001.
- [7] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. The Journal of Systems and Software, 74(2):173–194, January 2004.
- [8] O. Flege. System Family Architecture Description Using the UML. Technical Report IESE-Report No. 092.00/E, IESE, December 2000.
- [9] H. Gomaa. Object Oriented Analysis and Modeling for Families of Systems with UML. In W.B. Frakes, editor, IEEE International Conference for Software Reuse (ICSR6), pages 89–99, June 2000.
- [10] G. Halmans and K. Pohl. Communicating the variability of a software-product family to cutomers. Software and System Modeling, 2(1):15–36, 2003.
- [11] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231–274, 1987.
- [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 International Workshop on Requirement Engineering for Product Line (REPL02), pages 26–32, September 2002.
- [14] W. El Kaim. Managing variability in the LCAT SPLIT/Daisy. In Proceedings of Product Line Architecture Workshop. The First Software Product Line Conference (SPLC1), pages 21–32, 2000.

- [15] Object Management Group OMG. Unified Modeling Language specification version 2.0: Superstructure. Technical Report pct/03-08-02, OMG, 2003.
- [16] S. Robak, R. Franczyk, and K. Politowicz. Extending the UML for modeling variability for system families. *International Journal of Applied Mathematics Computer Sciences*, 12(2):285–298, 2002.
- [17] T. van der Maßen and H. Lichter. Modeling Variability by UML Use Case Diagrams. In *International Workshop on Requirement Engineering for Product Line (REPL02)*, pages 19–25, September 2002.
- [18] D.L. Webber. The Variation Point Model For Software Product Lines. Ph.D, George Mason University, George Mason University Fairfax, VA, 2001.
- [19] M.D. Weiss and C.T. Robert Lai. *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wisley, 1999.
- [20] T. Ziadi, H´ Hérouët, and J-M. Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, volume 3014 of LNCS, pages 129–139. Springer Verlag, 2003.
- [21] T. Ziadi, L. L. Hérouët, 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.
- [22] [2] T. Ziadi and J-M Jézéquel. -- *Product Line Engineering with the UML: Deriving Products*, chapter 15, *Software Product Lines: Research Issues in Engineering and Management*. Editeurs : T. Käköla et J. Duenas, 2006 Springer ISBN-10: 3-540-33252-9
- [23] EclipseUML tool home. <http://www.omondo.com/>
- [24] I-Logix. Rhapsody. <http://www.ilogix.com/products/rhapsody/index.cfm>