Journal of Universal Computer Science, vol. 16, no. 17 (2010), 2374-2393
submitted: 15/2/10, accepted: 30/8/10, appeared: 1/9/10

Systematic Management of Variability in UML-based Software Product Lines

Edson A. Oliveira Junior
(University of São Paulo, ICMC-USP, Brazil
edsonjr@icmc.usp.br)

Itana M. S. Gimenes
(State University of Maringá, DIN-UEM, Brazil
itana@din.uem.br)

José C. Maldonado
(University of São Paulo, ICMC-USP, Brazil
jcmaldon@icmc.usp.br)

Abstract: This paper presents *SMarty*, a variability management approach for UML-based software product lines (PL). *SMarty* is supported by a UML profile, the *SMartyProfile*, and a process for managing variabilities, the *SMartyProcess*. *SMartyProfile* aims at representing variabilities, variation points, and variants in UML models by applying a set of stereotypes. *SMartyProcess* consists of a set of activities that is systematically executed to trace, identify, and control variabilities in a PL based on *SMarty*. It also identifies variability implementation mechanisms and analyzes specific product configurations. In addition, a more comprehensive application of *SMarty* is presented using SEI's Arcade Game Maker PL. An evaluation of *SMarty* and related work are discussed.
Key Words: Profile, Stereotypes, UML-based Software Product Lines, Variability Management
Category: D.2, D.2.2, D.2.10

1 Introduction

The software product line (PL) engineering approach has gained increasing attention over the last years due to competitiveness in the software development segment. The economic considerations of software companies, such as cost and time to market, motivate the transition from single-product development to the PL approach, in which products are developed in a large-scale reuse perspective [Linden et al. 07].

The PL approach focuses mainly on a two-life-cycle model [Linden et al. 07]: **domain engineering**, where the PL core asset is developed for reuse; and **application engineering**, where the core asset is reused to generate specific products. The success of the PL approach depends on several principles, in particular variability management [Gomaa 05], [Korherr, List 07], [Linden et al. 07], [Pohl et al. 05], [Svahnberg et al. 05]. However, most of the existing solutions are only applied to specific PL approaches. Thus, there is a lack of an overall reasoning about variability management applied to more general approaches which take advantages of widely consolidated standard notations as, for instance, UML and its profiling extension mechanism [UML 09] for specific domain applications. Although several approaches use this mechanism as a basis to represent variability [Bragança, Machado 06], [Gomaa 05], [Korherr, List 07], [Ziadi et al. 03], most of them are not supported by a systematic process that provides guidelines to instruct the users on how to deal with variability issues in PL UML-based artifacts.

Therefore, this paper presents the *SMarty* approach for variability management in UML-based PL. This approach is supported both by a UML profile, the *SMartyProfile*, and a systematic variability management process, the *SMartyProcess*. *SMarty* makes it easier to deal with variability issues in PL. A complete example of how to apply *SMarty* to a PL is presented in section 4.

Section 2 introduces essential concepts of variability management in PL, as well as presents *SMarty*. Section 3 performs a comparison between the *SMartyProfile* and two different variability representation approaches providing some evidences of the *SMartyProfile*'s representation effectiveness. Section 4 applies our approach to the SEI's Arcade Game Maker PL. Section 5 presents inceptions and preliminary outcomes from an *SMarty* evaluation. Section 6 lists the related work, while Section 7 presents the conclusion, ongoing work, and directions for future work.

2 *SMarty*: Managing Variability in UML-based Software Product Lines

Variability is the general term used to refer to the variable aspects of the products of a PL. It is described through variation points and variants. A variation point is the specific place in a PL artifact to which a design decision is connected. Each variation point is associated with a set of variants that corresponds to design alternatives to resolve the variability [Linden et al. 07], [Pohl et al. 05]. According to [Pohl et al. 05] and [Linden et al. 07], variability management is related to every activity of a PL approach and must comprise the following activities: **variability identification**, consists of identifying the differences between products and where they take place within PL artifacts; **variability delimitation**, defines the binding time and multiplicity of variabilities; **variability implementation**, is the selection of implementation mechanisms; **variant management**, controls the variants and variation points. These activities and the its related concepts form the basis of our approach: *SMarty*.

SMarty is an approach for UML **S**tereotype-based **M**anagement of V**ari**abili**ty** in PL. It is composed of a UML 2 profile, the *SMartyProfile*, and a process, the

SMartyProcess. ***SMartyProfile*** contains a set of stereotypes and tagged values to represent variability in PL models (Section 2.1). Basically, *SMartyProfile* uses a standard object-oriented notation and its profiling mechanism [UML 09] both to provide an extension of UML and to allow graphical representation of variability concepts. Thus, there is no need to change the system design structure to comply with the PL approach. ***SMartyProcess*** is a systematic process that guides the user through the identification, delimitation, representation, and tracing of variabilities in PL models (Section 2.2). It is supported by a set of application guidelines as well as by the *SMartyProfile* to represent variabilities.

The following subsections present the *SMarty* profile and process descriptions. Section 4 provides a detailed example of its application.

2.1 The *SMartyProfile*

The *SMartyProfile* represents the relationship of major PL concepts with respect to variability management. There are four main concepts: variability [Bosch 04], variation point [Pohl et al. 05], variant [Pohl et al. 05], and variant constraints [Bosch 04].

Based on these variability management concepts, Figure 1 presents the *SMartyProfile* which is composed of the following stereotypes and respective tagged values:

≪**variability**≫ represents the concept of PL variability and is an extension of the metaclass `Comment`. This stereotype has the following tagged values: `name`, the given name by which a variability is referred to; `minSelection`, represents the minimum number of variants to be selected to resolve a variation point or a variability; `maxSelection`, represents the maximum number of variants to be selected in order to resolve a variation point or a variability; `bindingTime`, the moment at which a variability must be resolved, represented by the enumeration class `BindingTime`; `allowsAddingVar`, indicates whether it is possible or not to include new variants in the PL development; `variants`, represents the collection of variant instances associated with a variability; and `realizes`, a collection of lower-level model variabilities that realize this variability.

≪**variationPoint**≫ represents the concept of PL variation point and is an extension of the metaclasses `Actor`, `UseCase`, `Interface`, and `Class`. This stereotype has the following tagged values: `numberOfVariants`, indicates the number of associated variants that can be selected to resolve this variation point; `bindingTime`, the moment at which a variation point must be resolved, represented by the enumeration class `BindingTime`; `variants`, represents the collection of variant instances associated with this variation point; and `variabilities`, represents the collection of associated variabilities.

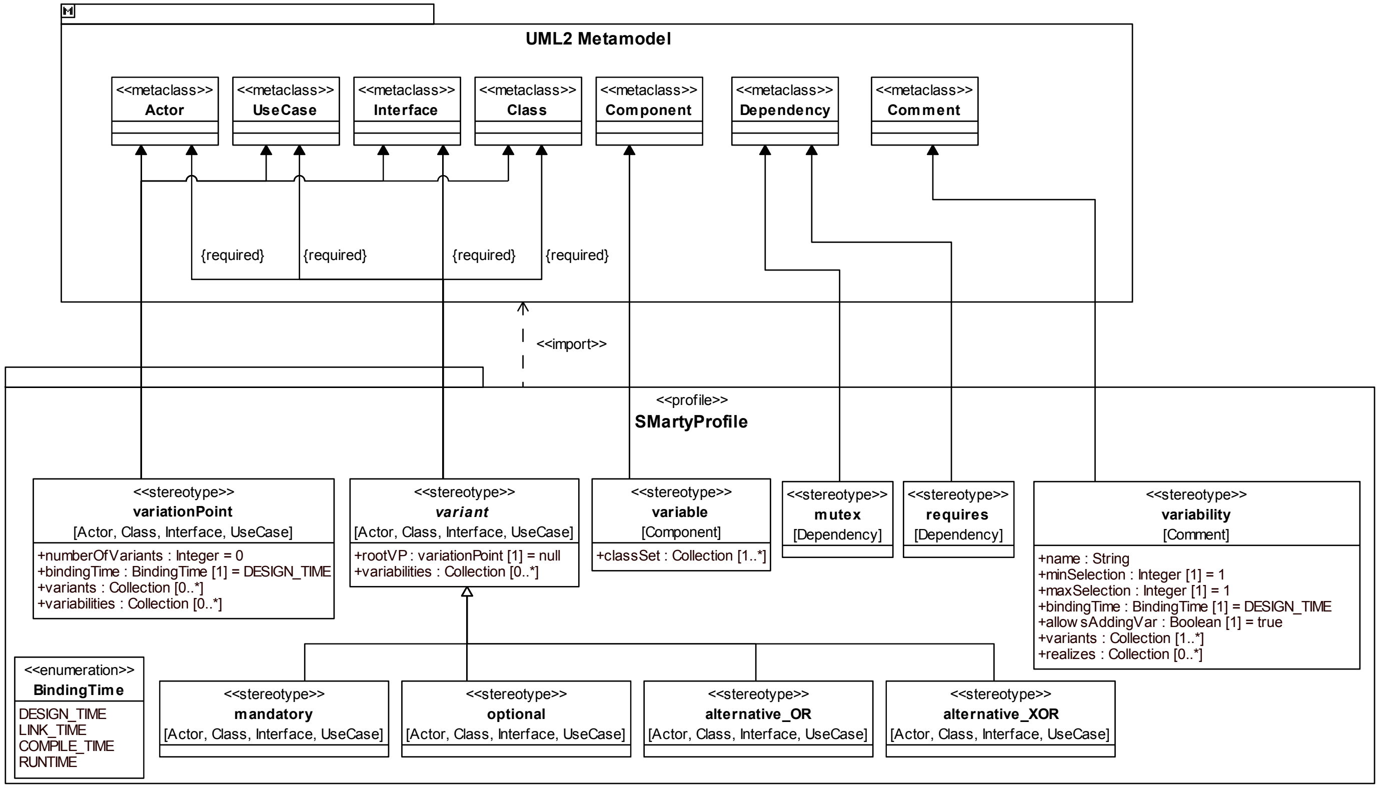

Figure 1: The *SMartyProfile* for Variability Management in PL.

≪***variant***≫ represents the concept of PL variant and is an abstract extension of the metaclasses `Actor`, `UseCase`, `Interface`, and *Class*. This stereotype is specialized in four other non-abstract stereotypes which are: ≪mandatory≫, ≪optional≫, ≪alternative_OR≫, and ≪alternative_XOR≫. The stereotype ≪variant≫ has the following tagged values: `rootVP`, represents the variation point with which this variant is associated; and `variabilities`, the collection of variabilities with which this variant is associated.

≪**mandatory**≫ represents a compulsory variant that is part of every PL product.

≪**optional**≫ represents a variant that may be selected to resolve a variation point or a variability;

≪**alternative_OR**≫ represents a variant that is part of a group of alternative inclusive variants. Different combinations of this kind of variants may resolve variation points or variabilities in different ways.

≪**alternative_XOR**≫ represents a variant that is part of a group of alternative exclusive variants. This means that only one variant of the group can be selected to resolve a variation point or variability;

≪**mutex**≫ represents the concept of PL variant constraint and is a mutually exclusive relationship between two variants. This means that when a variant is selected another variant must not also be selected;

≪**requires**≫ represents the concept of PL variant and is a relationship between two variants in which the selected variant requires the choice of another specific variant;

≪**variable**≫ is an extension of the metaclass `Component`. It indicates that a component has a set of classes with explicit variabilities. This stereotype has the tagged value `classSet` which is the collection of class instances that form a component.

2.2 The *SMartyProcess*

The *SMartyProcess* activities are directly related to a general PL development process [Linden et al. 07], [Pohl et al. 05]. Figure 2 illustrates the interaction between this process, represented by the activities vertically aligned on the left side, and the *SMartyProcess*, represented by the activities in the rectangle on the right side. [Oliveira Junior et al. 05] initially proposed a variability management process as a proof of concept to identify prospective stereotypes and activities for managing variabilities in PL. In this paper, we incorporate formal definitions of such stereotypes by proposing *SMartyProfile*. In addition, *SMarty* is formalized by combining *SMartyProfile* and *SMartyProcess* in an overall approach driven by guidelines to systematically manage PL variabilities.

SMartyProcess is iterative and incremental. It runs in parallel with the development of a PL. *SMartyProcess* progressively uses the outputs of the PL

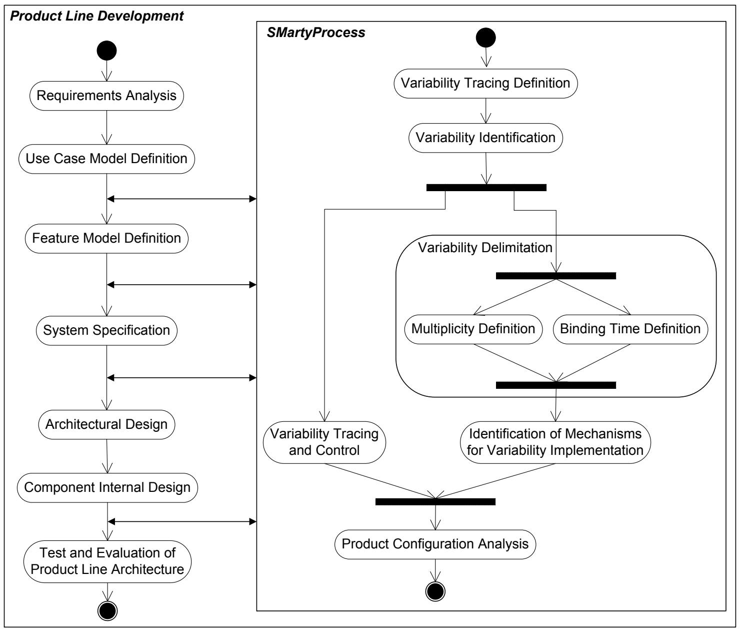

Figure 2: Interaction Between PL Development Activities and *SMartyProcess*.

development activities as inputs. Throughout the execution of activities, the number of variabilities tends to increase.

SMartyProcess is iterative, thus variability updates are allowed from any of its activities. Its activities and their respective inputs and outputs are presented in the next paragraphs. *SMartyProcess* consumes artifacts from the PL core asset, as well as providing it with information. The use case and class models, for instance, feed the *SMartyProcess* and return to the core asset with the variabilities identified and delimited. However, some models, such as the variability tracing and implementation ones, are originated in *SMartyProcess* itself.

Variability Tracing Definition receives as input the use case and the feature model built in the PL development process. A variability tracing model is built according to the following guidelines: (i) the captured features of the PL are listed; (ii) the captured use cases are listed; (iii) the relationship between features and use cases are re-analyzed; and (iv) crossing relations between use cases and features are marked with a blob. This model is represented in a tabular form. The model supports the tracing from the features to all PL UML models.

In each interaction between the PL development and the *SMartyProcess*, the **Variability Identification** activity progressively receives as input the use case, feature, class, and component models. It aims at progressively identifying the variability associated with the models. The *SMartyProfile* strongly supports the development of this activity by applying its stereotypes to the PL models. Variability identification is a domain-dependent activity which requires abilities of the PL managers and analysts. Therefore, guidelines are offered in order to support this activity, which include:

1. elements of use case models related to the extend and extension points mechanism[1] [UML 09] suggest variation points with associated variants which might be inclusive or exclusive alternative.
2. in class models, variation points and their variants are identified in the following relationships [UML 09]: **generalization**, the most general classifiers are the variation points and the most specific ones are the variants; **interface realization**, the suppliers (specifications) are variation points and the implementations (clients) are the variants; **aggregation association**, the typed instances with hollow diamonds are the variation points and the associated typed instances are the variants; and **composite aggregation**, the typed instances with filled-in diamonds are the variation points and the associated typed instances are the variants.
3. elements of use case models related to the include (dependency) or association from actors relationships [UML 09] suggest either mandatory or optional variants. Section 4 gives examples of these elements;
4. elements of class models related to the association relationship in which the `aggregationKind` attribute has value `none` [UML 09], i.e., neither an aggregation nor a composition suggest either mandatory or optional variants.
5. components, in component models, with variation point or variant classes are stereotyped as ≪variable≫.

Variability Delimitation aims at defining the following attributes of a variability: (i) multiplicity; (ii) binding time, and (iii) possibility, or not, of adding new elements to the associated variant set. The multiplicity of a variability indicates the minimum (`minSelection` attribute) and the maximum (`maxSelection` attribute) number of elements of its `variants` attribute (Section 2.1) that must be selected to resolve it. The following rules are applied: (i) variabilities with optional variants have multiplicity `minSelection = 0`, and `maxSelection = 1`; (ii) variabilities with exclusive alternative variants have multiplicity `minSelection = maxSelection = 1`; (iii) variabilities with inclusive

[1] In our approach, use case generalization is not used to represent variation points and variants as it does not represent the adding of specific actions from specialized use cases, as the extend relationship does [Bragança, Machado 06].

alternative variants have multiplicity `minSelection = 1`, and `maxSelection = length(variants)`.

The definition of the binding time is essential to determine the choice of implementation mechanisms, described in the next activity.

Identification of Mechanisms for Variability Implementation aims at selecting mechanisms to be used to implement variabilities. The inputs are class and component models with their respective variabilities represented and delimited as a result of the previous activity. The output of this activity is an implementation model represented as a table. Each row of the table indicates the name of a variability, the element in which it occurs, the binding time, the implementation mechanism, and the implementation strategy. The model is based on the variability implementation techniques proposed by [Jacobson et al. 97] and [Svahnberg et al. 05]. Among these techniques are generalization, extension, and parameterization.

Variability Tracing and Control uses a variability metamodel for describing the relationships between variant artifacts of a PL and their variation points, variants, binding time, and implementation mechanisms. Together with the variability tracing model, this metamodel allows the association of a feature with the related use cases and, therefore, with the elements of class and component models. The execution of this activity consists of the instantiation of the metamodel for a PL.

Configuration Analysis of Specific Products aims at investigating the impact of selecting possible features of the PL artifacts in order to analyze the feasibility of the production of specific PL products. The selection of a feature may imply the selection of a product configuration according to the variation points described. Moreover, whether the product requires an additional feature, the introduction of this feature must be analyzed in the PL artifacts.

Next section presents a case of comparison between *SMartyProfile* and two well-known approaches to represent variability in PL in order to show evidences of *SMartyProfile*'s effectiveness.

3 *SMartyProfile* and Variability Representation Approaches

In this section we compare the *SMarty* approach, specifically *SMartyProfile* representation, with the well-known [Halmans, Pohl 03] and [Gomaa 05] variability representation approaches.

[Halmans, Pohl 03] extend UML by adding the stereotype ≪variant≫ to represent PL variants, as *SMartyProfile* does. [Halmans, Pohl 03] claim that the representation of variation points in use case diagrams is not possible, thus they propose the triangle notation, as showed in Figure 3.

Figure 4 presents an excerpt of a Flight Booking System according to the approach proposed by [Halmans, Pohl 03].

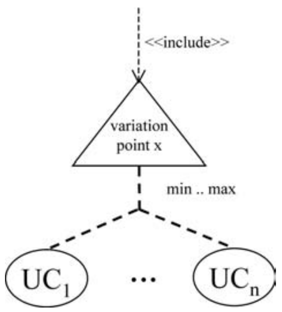

Figure 3: Representation of Variation Points [Halmans, Pohl 03].

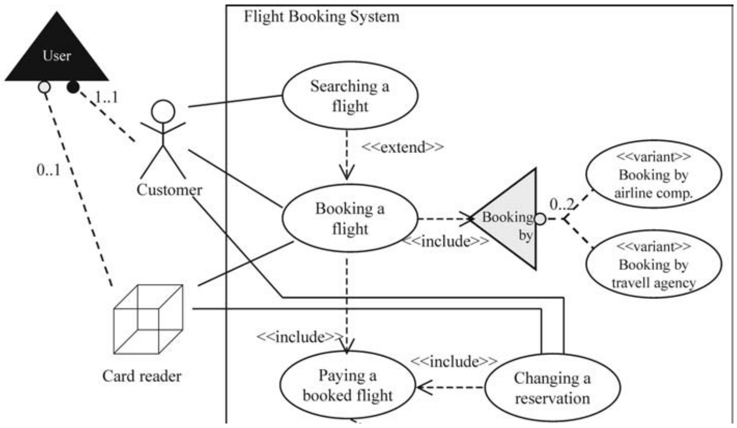

Figure 4: Excerpt of a Flight Booking System [Halmans, Pohl 03].

In [Halmans, Pohl 03] representation, the stereotype ≪variant≫ does not indicate the type of variant that we are dealing with. For instance, the use case `Booking by airline comp.` might be an optional, inclusive or exclusive variant. In *SMartyProfile*, ≪variant≫ is an abstract stereotype which is specialized by the stereotypes ≪optional≫, ≪mandatory≫, ≪alternative_OR≫, and ≪alternative_XOR≫, thus making explicit variant modeling in use cases. In addition, the four stereotypes inherit tagged values from the abstract one.

On the other hand, *SMartyProfile* uses the stereotype ≪variationPoint≫ to represent a variation point use case or class. A variation point is related to a UML comment with the stereotype ≪variability≫ representing a variability. The UML comment contains all the tagged values of the ≪variability≫ stereotype, representing essential information to resolve variabilities.

Although the triangle is an interesting notation for variation points, it is not part of the UML metamodel and, consequently, it is not supported by automated tools that provide XMI file features. *SMartyProfile* takes advantage of the UML extension points concept in use cases to represent the relations between variation point use cases and their variants. Moreover, variants cardinality is represented as a tagged value. It makes our representation fully compliant to the UML metamodel, thus it is supported by UML tools. Figure 5 presents the [Halmans, Pohl 03] example according to *SMartyProfile*. Note that the triangle `Booking by` (Figure 4) is replaced with the variability `Booking by` (Figure 5) keeping its original cardinality.

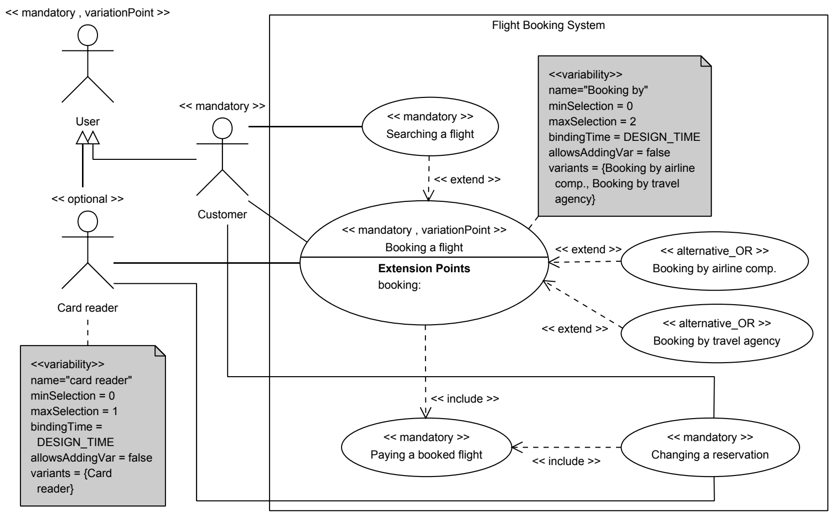

Figure 5: The Flight Booking System according to *SMartyProfile*.

[Gomaa 05] uses the UML extension points concept to represent variation points and their variants in use case diagrams, as *SMartyProfile* does. Figure 6 presents an example of [Gomaa 05] representation for a `Check Out Customer` feature.

Although [Gomaa 05] representation is fully compliant to the UML metamodel, it uses only ≪kernel≫ and ≪optional≫ stereotypes to represent variants. In addition, it does not make explicit variation points by means of stereotypes, as well as the information with regard to variation points resolution, as *SMartyProfile* does. For example, one might ask: what is the binding time of `Check Out Customer`? Thus, Figure 7 presents the `Check Out Customer` feature according to *SMartyProfile*.

We can easily see in Figure 7 which use case is a variation point, and its

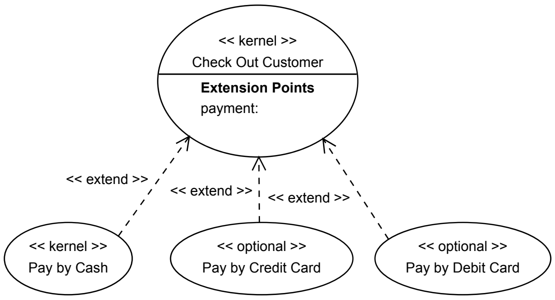

Figure 6: `Check Out Customer` Feature according to [Gomaa 05].

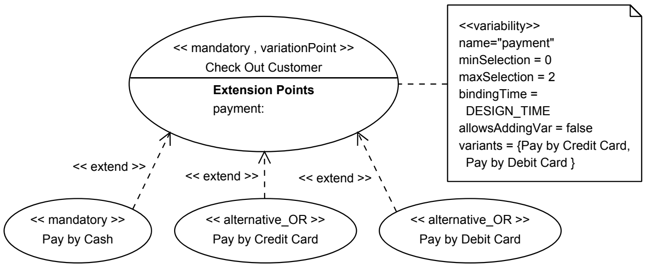

Figure 7: `Check Out Customer` Feature According to *SMartyProfile*.

information represented in the related variability, as well as the use cases which are either mandatory or alternative inclusive. In addition, the variation point `Check Out Customer` is bound at DESIGN_TIME, and it does not allow the addition of new variants.

These comparisons give us some evidences of the *SMarty* approach with relation to its effectiveness of variability representation. However, we understand that an overall experimental approach should be taken into account to claim such an effectiveness.

Next section presents a more detailed PL according to the *SMartyProfile*, its stereotypes and tagged values, for representing variability, and the *SMartyProcess* guidelines to identify and delimit variability.

4 The Arcade Game Maker PL According to *SMarty*

The Arcade Game Maker (AGM) [SEI-AGM 09] is a pedagogical and exemplary PL created by the SEI to support learning and experimenting based on PL concepts. It has a complete set of documents and UML models, as well as a set of tested classes and the source code for three different games: Pong, Bowling, and

Brickles. Although AGM is not a commercial PL, it has been used to illustrate the concepts of several different PL approaches, as well as PL and architecture evaluation case studies.

The essential AGM UML models are the feature model (Figure 8), the use case model (Figure 9) and the core asset class model (Figure 10).

The **AGM feature model** is composed of four main features: `services` which defines its sub-features `play`, `pause`, and `save`; `rules` which defines the sub-features `brickles`, `pong`, and `bowling`; `action` which defines its sub-features `movement` and `collision`; and `configuration`. The sub-features `save`, `brickles`, `pong`, and `bowling` are optional.

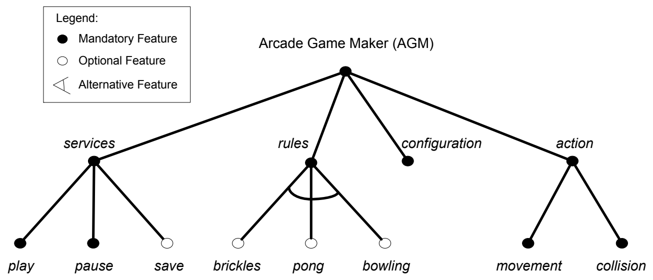

Figure 8: AGM Top-Level Feature Model.

The **AGM use case model** has two actors, `GamePlayer` and `GameInstaller` which trigger several use cases such as `Save Game`, `Exit Game`, and `Play Selected Game`. The use cases `Check Previous Best Score` and `Save Score` are triggered by the `GamePlayer` actor, whereas `Install Game` is triggered by `GameInstaller`.

The use case `Play Selected Game` is the most important use case. It has two extension points: `initialization_ext_point` and `animation_ext_point`. The former is responsible for allowing specific actions from the use case `Initialization`, whereas the later is responsible for specific actions from `Animation Loop` which can be realized by different games.

The **AGM core asset class model** has several concrete and abstract classes to generate products from the AGM PL. However, the most important classes are: `GameSprite` which represents an element of a game and has an associated `Rectangle` composed of a `Size` and a `Point`; `MovableSprite` and `StationarySprite` which represent, respectively, moving and non-moving game elements; `GameMenu` which represents the menu of a specific game and is composed of a `Board`.

In order to show how the *SMarty* approach can be applied, we followed

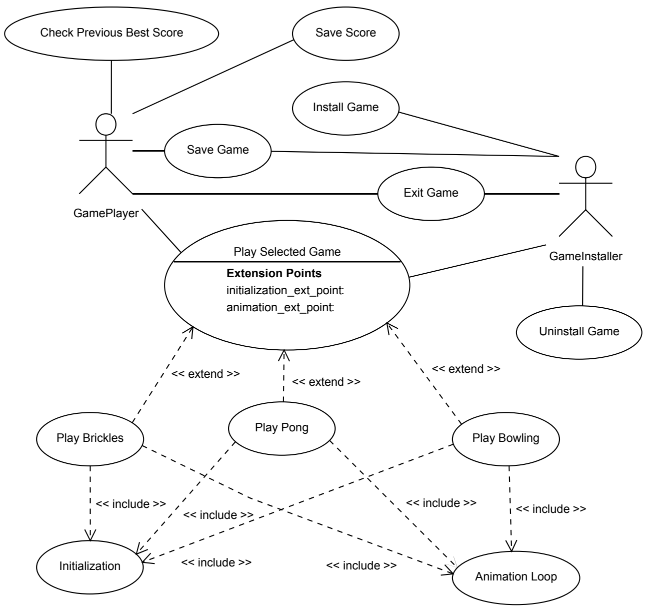

Figure 9: AGM Use Case Model.

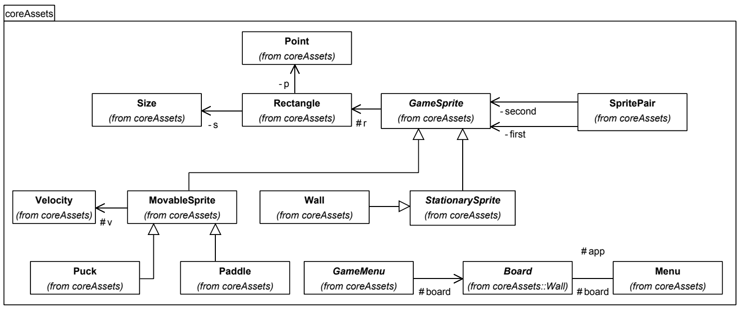

Figure 10: AGM Core Asset Class Model.

the *SMartyProcess* activities to the AGM PL, supported by *SMartyProfile*, as follows:

– **AGM Variability Tracing Model:** we built the AGM tracing model, presented in Table 1, taking into account the AGM feature and use case models. As we can see in this table, AGM features and use cases are listed and their crossing relations are marked with a blob. For instance, the feature `save` is related to the use cases `Save Score` and `Save Game`. It allows the tracing from a given feature to its related use cases, whereas feature selection operations are realized during PL product derivation activities. For example, if the feature `save` is removed from a certain AGM game, the related use cases `Save Game` and `Save Score` must also be removed.

Use Case	Feature								
	services			*rules*			*action*		*configuration*
	play	*pause*	*save*	*Brickles*	*Pong*	*Bowling*	*movement*	*collision*	
Check Previous Best Score		•							
Save Score			•						
Install Game									•
Save Game			•						
Exit Game		•							
Play Selected Game	•			•	•	•			
Uninstall Game									•
Play Brickles	•			•					
Play Pong	•				•				
Play Bowling	•					•			
Initialization							•	•	
Animation Loop							•	•	

Table 1: AGM Tracing Model According to *SMarty*.

– **AGM Variability Models:** by following the variability identification and delimitation guidelines, we identified and delimited several variabilities for the AGM use case and core asset class models. Figures 11 and 12 present the AGM original use case and class models, respectively, with variabilities identified and delimited according to respective stereotypes (Section 2.1) and guidelines (Section 2.2).

  The **AGM use case variability model**, according to Section 2.2, guideline #1, has a mandatory variation point `Play Selected Game` which has the extension points `initialization_ext_point` and `animation_ext_point`, and the associated variability named `play game`. These extension points allow the inclusive alternative variants `Play Brickles`, `Play Pong`, and `Play Bowling` to add specific initialization and animation loop game actions to the variation point. In addition, the variability `play game` indicates that at least one and at most three variants must be selected, and that it is possible to include new variants during the design of the PL. The AGM use case model also has, according to Section 2.2, guideline #3, two optional use cases, `Check Previous Best Score` and `Save Score`, with respective variabilities. The selection of the former use case forces the selection of the later due to the `requires` constraint (Section 2.1). Remaining use cases and the actors are mandatory variants of the AGM use case model.

  The **AGM core asset variability model** according to Section 2.2, guideline #2, has a mandatory variation point, the abstract class `GameSprite`

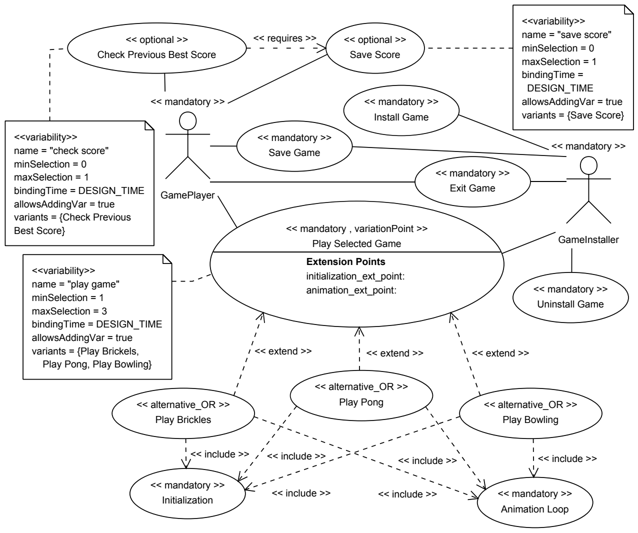

Figure 11: AGM Use Case Variability Model According to *SMarty*.

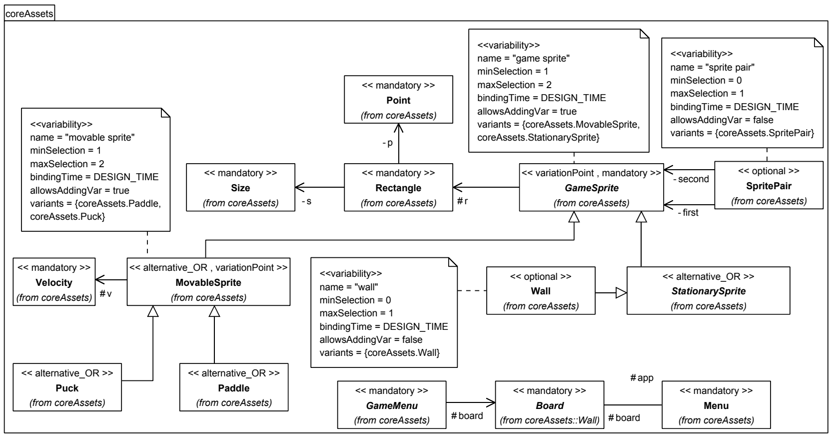

Figure 12: AGM Class Variability Model According to *SMarty*.

which has the alternative inclusive variants `MovableSprite` and `Stationary-Sprite`, and the associated variability `game sprite`. This variability indicates that at least one and at most two variants must be selected, and that it is possible to include new variants at design time. The former variant is also a variation point which has the alternative variants `Puck`, and `Paddle`, and the associated variability named `movable sprite`, indicating that at least one and at most two variants must be selected, and it is possible to include new variants at design time. The AGM core asset class model according to Section 2.2, guideline #4, also has two optional variants, `Wall` and `SpritePair`, with respective variabilities. The other classes are mandatory variants of the AGM core asset class model.

All classes from the AGM class variability model form, according to Section 2.2, guideline #5, the variable component `Game`.

– **AGM Mechanisms for Variability Implementation:** the definition of the variability implementation model can be done based on variability implementation mechanisms and strategies proposed by [Svahnberg et al. 05]. Thus, Table 2 presents the variability mechanisms defined for the AGM PL. For instance, the variability `wall` occurs in the class `Wall` and it is bound at design time via `Variant Class Specialization`, implemented using `Strategy` and `Template` patterns.

Variability Name	Level Class / Component	Binding Time	Implementation Mechanism	Implementation Strategy
movable sprite	MovableSprite	Design Time	Variant Class Specialization	Strategy and Template Patterns
wall	Wall	Design Time	Variant Class Specialization	Strategy and Template Patterns
game sprite	GameSprite	Design Time	Variant Class Specialization	Strategy and Template Patterns
sprite pair	SpritePair	Design Time	Variant Class Specialization	Strategy and Template Patterns

Table 2: AGM Variability Implementation Model.

We are currently developing an automated environment for PL architecture evaluation which will comprise a module to allow the tracing and control of variabilities of a certain PL. In addition, such a tool will provide a feature for configuration analysis of specific products derived from a PL. Thus, the *SMartyProcess*' activities **Variability Tracing and Control** and **Configuration Analysis of Specific Products** are not showed in this paper.

5 Initial Evaluation of *SMarty*

This section presents insights and outcomes of an initial evaluation of the *SMarty* approach.

The key issue of a PL is the way it articulates and manages its variable aspects. A variability management approach must coexist with any PL core asset development and product development in order to support the clear specification, tracing, and control of variabilities. Our initial studies, [Oliveira Junior et al. 05] and [Oliveira Junior et al. 08], as well as Section 3 provide evidences that *SMarty* enables better representation and, therefore, better control of variabilities, compared to existing PL approaches. Moreover, it makes explicit the most important decisions, such as the number of variants associated with a variation point and the type of choices allowed, the binding time, and the implementation mechanisms.

An initial evaluation of *SMarty* was carried out based on an empirical study in the form of a quantitative case study [Oliveira Junior et al. 05] which enabled the demonstration of the need of a complete UML-based variability management approach. The case study controlled the following variables: (i) the number of artifacts in each activity of the PL process; (ii) the number of variabilities in each artifact, determined by the sum of variable elements in each artifact; and, (iii) the variability types associated with each variation point. The *SMartyProfile* representation allowed a more precise estimation of the number of products that can be derived from a PL, thus offering a better support for PL configuration analysis. However, investigations have to proceed in order to increase the volume of data available for formal experiments. Ongoing work aims at building an experimental basis for UML-based PLs that allows the inference of quality attributes of both the PL itself and potential products.

SMarty introduces the idea of using UML comments to make explicit variability meta-attribute values such as multiplicity and binding time. One of the advantages of using comments is that they belong to the UML standard metamodel and, thus, can be read from any commercial tool that supports UML modeling.

6 Related Work

The *SMarty* approach takes into account previous work on PL approaches mainly related to the following issues: management activities, artifact notation, variability attributes, metadata modeling, and experimental software engineering. The management activities defined for *SMartyProcess* are aligned with the essential PL engineering activities [Linden et al. 07]. Variability management is considered a subprocess of the PL management activity, thus it has a close interaction with domain and application engineering activities.

The activities defined in our approach were initially based on [Linden et al. 07] and [Pohl et al. 05] suggestions of variability management activities. However, their work only lists the activities within the context of a well-defined process with roles, inputs, and outputs. Our *SMartyProcess* activities and their associated roles and artifacts are fully specified.

SMarty uses the *SMartyProfile* as its basis to represent the PL artifacts in terms of variability concepts and its attributes. It takes into consideration similar approaches such as [Bragança, Machado 06], [Gomaa 05], [Korherr, List 07], and

[Ziadi et al. 03]. [Chen et al. 09] presents a systematic review with respect to chronological variability background which includes most of related work of this paper.

[Bragança, Machado 06] offer an extension of the UML metamodel for the extend relationship in use case models and its formalization using activity diagrams. The extension occurs at the description level of the use case steps and is concerned with the type of sequence of steps. To represent this type sequence, the authors propose a set of new metaclasses and stereotypes, such as: Location, Rejoin, ExtensionFragment, and InclusionPoint, ≪before≫, ≪after≫, ≪rejoin_point≫, and ≪inclusion_point≫. Although *SMarty* also takes into account use case extend relationships to represent variability, it does not do it at the use case steps description level as all the information needed is represented graphically using stereotypes and UML comments. The advantage of *SMarty* is that it does not require variability information at description level of use case steps.

[Ziadi et al. 03] propose a UML profile for representing variability in class diagrams. The stereotypes defined for class diagrams are: ≪optional≫ to specify optionality, ≪variation≫ to specify variation points as abstract classes, and ≪variant≫ for subclasses. *SMartyProfile* also defines the ≪optional≫ stereotype. In our approach, however, we consider the use of the ≪variationPoint≫ stereotype not only for abstract classes, but also for interfaces as they both represent the concept of polymorphism. In addition, because there are many kinds of relationships between variants and variation points, our approach defines several different stereotypes to represent variants in class diagrams. As a result, the user has a set of choices to solve its variability modeling issues. [Ziadi et al. 03] proposed stereotypes, however, do not have tagged values which makes it difficult to understand the relationships/constraints between variability, variation point, and variant.

[Korherr, List 07] propose a UML profile for representing variability in class models and a mapping between its stereotypes and activity diagrams. The following stereotypes are part of this profile: ≪variation point≫ for superclasses, ≪variant≫ for subclasses, ≪mandatory≫ for compulsory variants, ≪optional≫, and ≪alternative choice≫. Although this profile resembles our profile, the *SMartyProfile* allows to explicitly represent inclusive and exclusive alternative variants by means of different stereotypes, making variability modeling more precise and intuitive. Furthermore, *SMartyProfile* is also concerned with representing variability in use case and component models.

[Clauß 01] presents an approach for modeling variability using UML. Although *SMartyProfile* is partially based on this work, Clauss' approach defines two different stereotypes for alternative variants - inclusive and exclusive alternative variants - and uses UML comments to distinguish between them. *SMar-*

tyProfile takes advantage of the comments to represent variability and its attributes, such as binding time and multiplicity. Furthermore, *SMartyProfile* is not defined only at the class level, but also at the use case and component levels.

7 Conclusion and Future Work

The *SMarty* approach presented in this paper: (i) defines a UML profile with stereotypes, tagged values, and relationships/constraints among its elements to better represent variability issues in PL models, making explicit variability attributes, such as multiplicity and binding time; (ii) defines a process composed of activities, artifacts, and roles necessary to control variability in UML-based PLs, supported by guidelines for identifying variabilities in PL models; (iii) makes explicit important design decisions, such as the alternatives for variation points, binding times and implementation mechanisms; and (iv) allows a clear relationship between features and PL architecture through the use of variability tracing models.

SMarty brings out advantages to the PL engineer, such as: (i) graphical identification of variabilities in PL models; (ii) better management of variabilities as they are identified, represented, and organized in a systematic way allowing more effective PL documentation; (iii) facilitate the generation of PL products by automated tools that provide support for UML metamodels; and (iv) decrease the time to market by dealing with less complex variabilities as consequence of a systematic approach. However, we understand that the major difficult in applying *SMarty* is to establish an automated environment to support the *SMarty* activities and their generated data.

We are currently investigating how to extend our *SMartyProfile* to represent variability in sequence and activity diagrams, as well as proposing guidelines to give directions to the user. We are also working on improving our variability representation in component models and extracting metrics to measure, via formal experiments, the effectiveness of this representation. We have already proposed metrics to evaluate PL architectures [Oliveira Junior et al. 08] by means of measuring use case, class, and component models according to *SMartyProfile*. These support the definition of a systematic evaluation method for PL architectures. In addition, we are working on an automated PL architecture evaluation environment which will support the application of *SMarty* and its *SMartyProfile* and *SMartyProcess*, as well as the collection of metrics and the execution of experiments.

Further work includes the design and implementation of a support tool to make automated product configuration analysis possible, thus providing organizations with mechanisms to evaluate the adoption and evolution of PLs. Furthermore, we plan to conduct experiments to validate empirically the *SMartyProfile* by applying it to several different PL.

Acknowledgments

The authors would like to thank CAPES-Brazil for funding Edson's visiting scholar term at the University of Waterloo, Ontario, Canada.

References

[Bosch 04] Bosch, J.: Preface. In: Proceedings of the 2nd Groningen Workshop on Software Variability Management: Software Product Families and Populations, Groningen, The Netherlands (2004)

[Bragança, Machado 06] Bragança, A., Machado, R. J.: Extending UML 2.0 Metamodel for Complementary Usages of the ≪extend≫ Relationship within Use Case Variability Specification. In: Proceedings of the 10th International Software Product Line Conference, pp. 123–130. (2006)

[Chen et al. 09] Chen, L., Babar, M. A. and Ali, N.: Variability Management in Software Product Lines: A Systematic Review. In: Proceedings of the XIII Software Product Line International Conference, pp. 81–90. (2009)

[Clauß 01] Clauß, M.: Generic Modeling Using UML Extensions for Variability. In: Proceedings of Workshop on Domain Specific Visual Languages, pp. 11–18. Tampa Bay (2001)

[Gomaa 05] Gomaa, H.: Designing Software Product Lines with UML: from Use Cases to Pattern-based Software Architectures. Addison-Wesley, (2005)

[Halmans, Pohl 03] Halmans, G., Pohl, K.: Communicating the Variability of a Software-Product Family to Customers. Springer-Verlag, New York (2003)

[Jacobson et al. 97] Jacobson, I., Griss, M. L., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, (1997)

[Korherr, List 07] Korherr, B., List, B.: A UML 2 Profile for Variability Models and their Dependency to Business Processes. In: Proceedings of the 18th International Conference on Database and Expert Systems Applications, pp. 829–834 (2007)

[Linden et al. 07] Linden, F. J. van der, Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer-Verlag, New York (2007)

[Oliveira Junior et al. 05] Oliveira Junior, E. A., Gimenes. I. M. S., Huzita, E. H. M., Maldonado, J. C.: A Variability Management Process for Software Product Lines. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 225–241. IBM Press, Toronto, Ontario, Canada (2005)

[Oliveira Junior et al. 08] Oliveira Junior, E. A., Gimenes, I. M. S., Maldonado, J. C.: A Metric Suite to Support Software Product Line Architecture Evaluation. In: Proceedings of the 34th Conferencia Latinoamericana de Informática, pp. 489–498. Santa Fé, Argentina (2008)

[Pohl et al. 05] Pohl, K., Böckle, G., Linden, F. J. van der: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, New York (2005)

[SEI-AGM 09] SEI - Arcade Game Maker Pedagogical Product Line, `http://www.sei.cmu.edu/productlines/ppl`

[Svahnberg et al. 05] Svahnberg, M., van Gurp, J., Bosch, Jan: A Taxonomy of Variability Realization Techniques: Research Articles. Software Practice and Experience. v.35, n.8, pp. 705–754 (2005)

[UML 09] OMG - UML Specification - Superstructure v2.2, `http://www.omg.org/cgi-bin/doc?formal/09-02-02`

[Ziadi et al. 03] Ziadi, T., Hélouët, L., Jézéquel, J. M.: Towards a UML Profile for Software Product Lines. In: Proceedings of the Product Family Engineering, pp. 129–139. (2003)