Towards Metamodel Support for Variability and Traceability in Software Product Lines

Yguaratã Cerqueira Cavalcanti
Federal University of Pernambuco
Reuse in Software Engineering
Recife, Brazil
ycc@cin.ufpe.br

Ivan do Carmo Machado
Federal University of Bahia
Reuse in Software Engineering
Salvador, Brazil
ivanmachado@dcc.ufba.br

Paulo Anselmo da Mota Silveira Neto
Reuse in Software Engineering
pamsn@rise.com.br

Luanna Lopes Lobato
Federal University of Pernambuco
Reuse in Software Engineering
Recife, Brazil
lll@cin.ufpe.br

Eduardo Santana de Almeida
Federal University of Bahia
Reuse in Software Engineering
Salvador, Brazil
esa@dcc.ufba.br

Silvio Romero de Lemos Meira
Federal University of Pernambuco
Reuse in Software Engineering
Recife, Brazil
srlm@cin.ufpe.br

ABSTRACT

In Software Product Lines (SPL), where a greater variety of products are derived from a common platform and constantly changed and evolved, it is important to manage the SPL variability and the traceability among its artifacts. This paper presents a metamodel which aims to coordinate SPL activities, by managing different SPL phases and their responsibles, and to maintain the traceability and variability among different artifacts. The metamodel was built for a SPL project in a private company working in the medical information management domain, which includes four products encompassing 102 different modules and 840 features. The metamodel is divided into five sub-models: project and risk management, scoping, requirements and testing. It is represented in the UML notation. Organizations using this metamodel as basis for their approaches, can easily understand the relationships between the SPL assets, communicate to the stakeholders, and facilitate the evolution and maintenance of the SPL. The metamodel can also be adapted to the single system development context.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Design

General Terms

Design, Documentation, Management

Keywords

design, software engineering, variability, software product lines, metamodel

VaMoS '11, January 27-29, 2011 Namur, Belgium

1. INTRODUCTION

SPL has proven to be a successful approach in many business environments [9, 18]. Nevertheless, the SPL advantages do not come for free. They demand mature software engineering, planning and reuse, adequate practices of management and development, and also the ability to deal with organizational issues and architectural complexity. If these points are not considered, the product line success could be missed [6]. Therefore, the development should be supported by auxiliar methods and tools, specially due to the complexity of the software systems that a SPL is supposed to deal with, represented by the variabilities.

Modeling can be used as a support mechanism to define and represent the variability involved in a SPL in a controlled and traceable way, as well as the mappings among the elements that compose a SPL. Many SPL projects are developed and maintained using model-based approaches [10]. In this context, this work proposes a metamodel representing the interactions among the assets of a SPL, developed in order to provide a way of managing traceability and variability. The proposed metamodel consists of representing diverse reusable assets involved in a SPL project, ranging from scoping to test artifacts, and also documentation.

The motivation to build the metamodel emerged from the experience gained during an industrial SPL project development that we have been involved in. This project consists of introducing SPL practices in a company working in the medical information management domain, placed in Sal-

vador, Brazil. The company currently has a single software development process and develops four different products: *SmartHealth*, a product composed by 35 modules (or subdomains), which has the capability of managing a whole hospital, including all areas, ranging from financial management to issues related to patient's control; *SmartClin*, composed by 28 modules, is responsible to perform the clinical management, supporting activities related to medical exams, diagnostics and so on; *SmartLab* is a product composed by 28 modules, which integrates a set of features to manage labs of clinical pathology; all these are desktop-based products, the only web-based product is the *SmartDoctor*, which is composed by 11 modules, and is responsible to manage the tasks and routines of a doctor's office.

Throughout this project, during the *scoping phase* [14] of the SPL life cycle, eight hundred and forty features (840) were consolidated. According to the time recorded in the management tool used in the project, dotProject[1], the scoping phase took approximately 740 man/hours of work. After this phase, *requirements engineering* started and the challenge faced during this phase was how to trace the variability and evolution among several assets such as product map [9], sub-domain documentation, features documentation, besides requirements, use cases, test cases and all the relationships among these assets. Although this work could be done with conventional tools, such as text and spreadsheet processors, it might be very error prone and not efficient. To understand such difficulties, consider the scenario where features should be retrieved from the existing products and then integrated with the products' requirements; afterwards, use cases must be linked to requirements, test cases linked to use cases, and so on. Clearly that would be not trivial to be performed with conventional tools.

Although diverse metamodels have been proposed in the literature [4, 21, 3, 19, 8, 16, 2] in order to address variability and traceability aspects, they generally cover the SPL phases partially or do not treat such issues together through all the SPL disciplines. In this paper, we propose a metamodel which provides support for several SPL aspects, such as scoping, requirements, tests, and project and risk management. Furthermore, the metamodel was built upon a set of requirements concerning different issues identified in the literature, which is further discussed in Section 2. It also serves as a basis to understand the mappings among the SPL assets, communicate them to the stakeholders, facilitate the evolution and maintenance of the SPL, as well as it can be adapted to other contexts.

The remainder of this paper is structured as follows: Section 2 specifies the requirements for a SPL metamodel; Section 3 describes the actual proposed metamodel, detailing its building blocks; in Section 4 an initial validation of the proposal is presented; Section 5 presents the related work; and finally, Section 6 concludes this work and outlines future work.

2. REQUIREMENTS FOR THE SPL METAMODEL

We identified some work [4, 19, 8, 22] that elicited different requirements which a metamodel for SPL should be in conformance with, in order to support properly the SPL development characteristics. The requirements are following described.

In [19], the authors propose four (4) requirements that a framework for modeling variability in SPL should have to support product derivation. However, the product derivation phase in SPL depends on how properly the previous phases are done. Therefore, such requirements in our metalmodel are considered from the initial SPL phases:

Uniform and first-class representation of variation points in all abstraction levels. [19] argues that "*uniform and first-class representation of variation points facilitates the assessment of the impact of selections during product derivation and changes during evolution*".

Hierarchical organization of variability representation. In [19], it is also argued that explicitly representing these variation points hierarchically reduces the cognitive complexity during the product derivation process.

Dependencies, including complex ones, should be treated as first-class citizens in the modeling approach. "*First class representation of dependencies can provide a good overview on all dependencies*". In our metamodel, we provide mechanisms that enable all the SPL assets, even as all the dependencies, be treated as first-class citizens.

The interactions between dependencies should be represented explicitly. All the assets in a SPL should be linked in order to preserve their dependencies. By doing that, the product derivation, maintenance and evolution of the SPL can be performed in an efficient and effective way. Thus, the metamodel should have a very high degree of linkage among its entities.

In [8], three (3) essential requirements were defined to support variability documentation across different SPLs. Although our metamodel is not intended to support multiple SPLs yet, one of those requirements may be useful:

Facilitate the creation of views on the documented variability and constraints. [8] argues that an approach to document and manage variability across SPLs needs "*to support selective retrieval of variability and commonality information, including variability constraints*". It is not a specific cause when supporting different SPLs, but it is also important in a single SPL.

2.1 Traceability Requirements

While analyzing all the previous requirements, we have noticed that *traceability* is the fundamental building block that enables those requirements. Thus, the basis for constructing a SPL metamodel is a strong linkage among all elements/assets involved in a SPL development. These elements, such as features, requirements, design, source code, and test artifacts, should be linked in a way that their building, evolution and maintenance could be more effectively controlled, and the impact of changes and addition of new features in different products from the SPL could be analyzed.

According to [22], tracing approaches should capture and manage relationships among the different documents built during development phase. It enables to support various stakeholders, such as maintainer, project manager, customers, developers, testers, etc., in performing their tasks. For example, project planners use traceability to perform impact analysis, while designers use it to understand dependencies

[1]**dotProject** is an open source, web-based project management application, and was used in the project reported in this work.

between requirement and design [22]. Thus, we believe that without enabling traceability as a basis of the metamodel, the realizations of the requirements previously described are not feasible, becoming the SPL development a complete disorder.

In [4], some requirements are set that consider traceability issues: (a) *base the traceability on the SPL metamodel*; (b) *it should be customizable in terms of available trace types*; (c) *it should be capable of handling variability in the product line infrastructure*; (d) *the number of traces should be as small as possible*; and (e) *it should be automatable.*

Although we presented a large set of requirements along this section, we have not found studies that implement all these together. Thus, in our proposal, we grouped all these requirements to fit them in our metamodel, as detailed in next section. Furthermore, such requirements can also serve as a guideline for building SPL models.

3. THE SPL METAMODEL

In this section, we describe the initial metamodel developed in order to fulfill the requirements specified in previous sections. It was developed using the UML notation [7]. The metamodel was initially divided into *SPL management*, which currently includes the risk management submetamodel, and the *SPL core development*, which are the scoping, requirement, and tests sub-metamodels. Henceforth, we will call the submetamodels just by models.

Figure 1 shows the overall view of the metamodel, where the dashed boxes specify the models of the metamodel. The variability model is strongly based on the generic metamodel proposed by [3]. We split the metamodel into small diagrams in the next subsections in order to explain it in details. It starts by detailing the Asset UML Profile, scoping model, then moving towards requirement, tests, and management models.

3.1 Asset UML Profile

This is the metamodel core entity, called *Asset*, as shown in Figure 2. This entity is used as a UML profile[2] for the other entities of the metamodel which should behave as an *Asset*. Thus, whenever an entity of the SPL metamodel have the properties of an *Asset*, it is extended using the *asset* profile tag. The usage of such profile has three main reasons: (1) to enable the evolution and maintenance of the metamodel without the need to modify the entire metamodel; (2) to transform the metamodel entities into first class entities; and (3) to keep the metamodel views clean and understandable.

The *Asset* entity is the metamodel core since it has properties that make feasible some of the requirements aforementioned. For example, it is related to a *History* entity, which is responsible to keep track of the changes that are performed in some *Asset* object. Thus, a *History* object records the *Asset* object which was modified, what kind of modification was performed, and who did it. Recording such modifications enables, for example, to calculate the probability that an *Asset* object has to be modified – such probability could impact directly in the SPL architecture design [22].

The *Asset* entity is also related to a set of metrics (*Metric* entity). During the SPL development, it is important to have information quantifying the metrics. For example, we could measure the effort for scoping analysis and requirement analysis, thus it would be possible to estimate the value for each feature or requirement. We could also set a metric for number of LOC (Lines of Code) for each feature in the SPL, or how many requirements and use cases some feature has. The granularity of the metrics can vary from the very high level to the very low level due to the strong tracking capability of the metamodel.

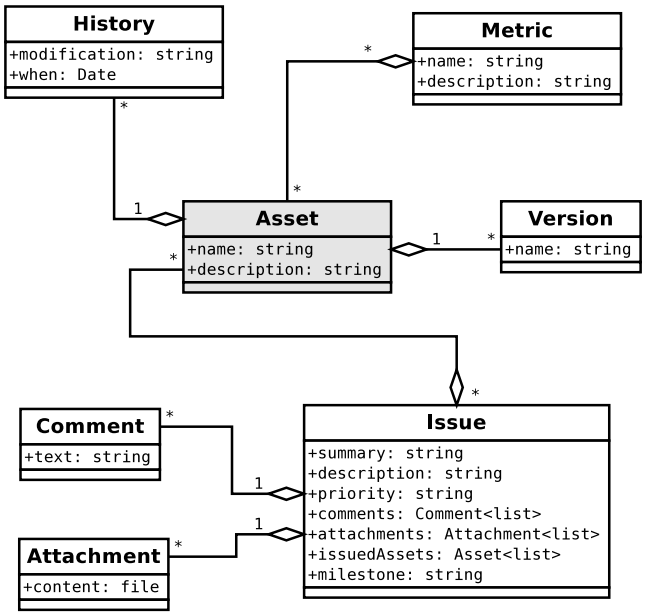

Figure 2: The metamodel core UML profile.

Furthermore, the *Asset* entity is also related to a *Version* entity. It enables the *Asset* objects to be versioned. This characteristic is important due to the variability nature of a SPL project. The presence of the *Version* entity means that for each modification in an *Asset* object, a new version of this should be created. Thus, integrating a versioning mechanism inside the metamodel will enable easy maintenance of different versions of the same product. If the metamodel is extended to support different SPLs, it becomes more critical. The *History* entity should not be confused with the *Version* entity; the former holds metadata information for the *Asset* object, while the later keeps different copies of an *Asset* object.

Last, but not least, the metamodel also integrates a mechanism for issues reporting. This mechanism enables any *Asset* object to be associated with an *Issue* object. It also means that someone could report an issue for different versions of the same *Asset* object. As direct impact of this mechanism, it will provide easy maintenance and evolution of different *Asset* object versions. However, due to better understanding reasons, the issue mechanism showed in the Asset UML Profile is a simplification of what an issue mechanism should be. Therefore, the metamodel can be extended with other specialized entities to support a complete issue tracker system, even as it could be done for versioning, metrics, and history mechanisms.

[2]UML profiles are an extension mechanism of the UML language [7] that allow models to be customized for specific domains.

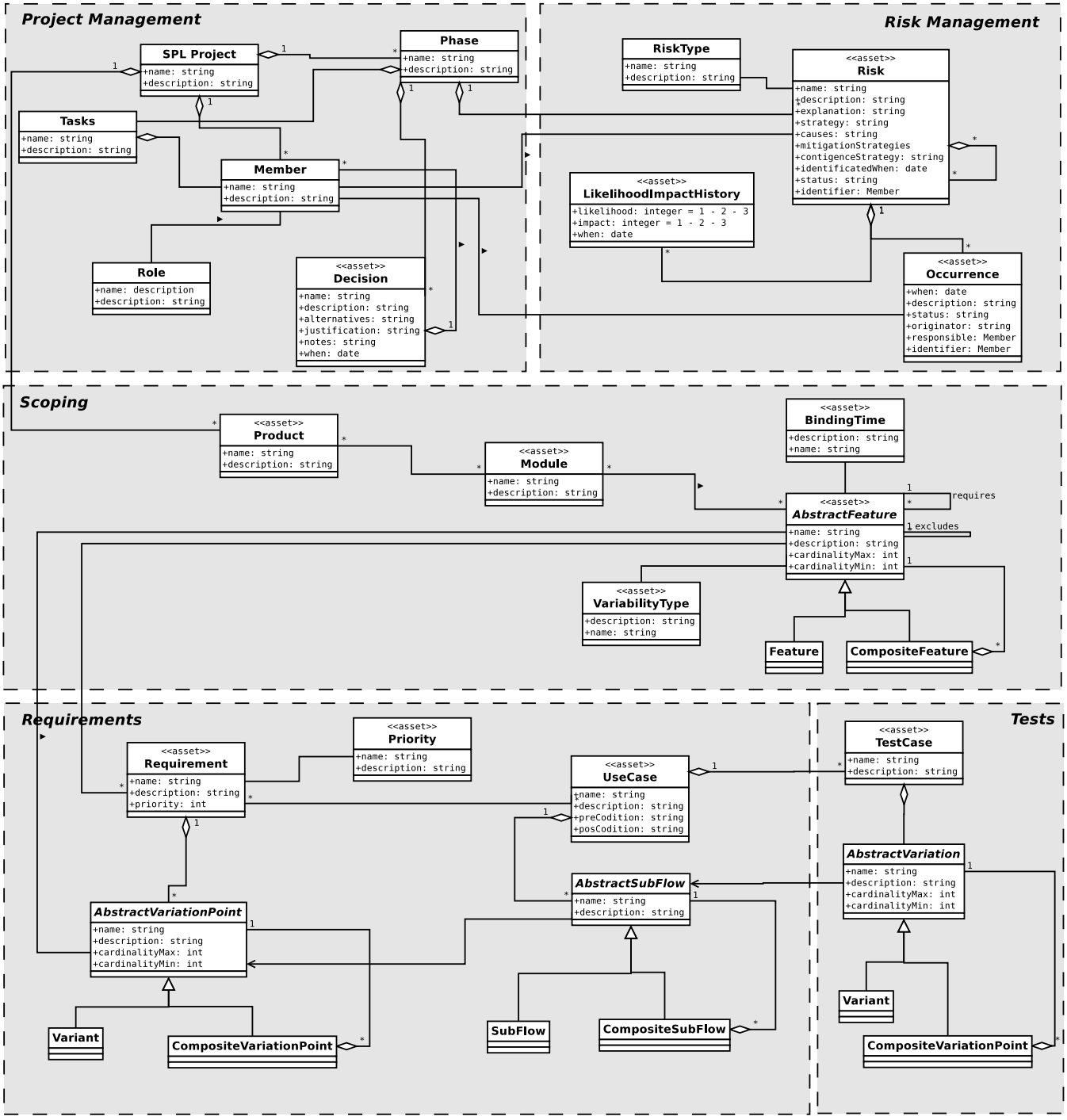

Figure 1: Overview of the proposed SPL metamodel.

3.2 Scoping Model

During the scoping phase [14], the feature model is assembled based on the method to identify those features [15]. In some cases, a tree is built containing all feature types, such as mandatory, alternative and optional. The scoping model is shown in Figure 3.

We used the Composite design pattern [12] to represent the feature model, since it is a good representation using UML diagrams for the feature model proposed by [15]. It enables the features and their dependencies to be represented in a form of a tree, where features can have sub-features (children) recursively. The feature model proposed in the scoping model also enables other relationship between features, e.g. it is possible to specify what are the required and excluded features when choosing any feature. Moreover, the metamodel can be extended to support other relationships.

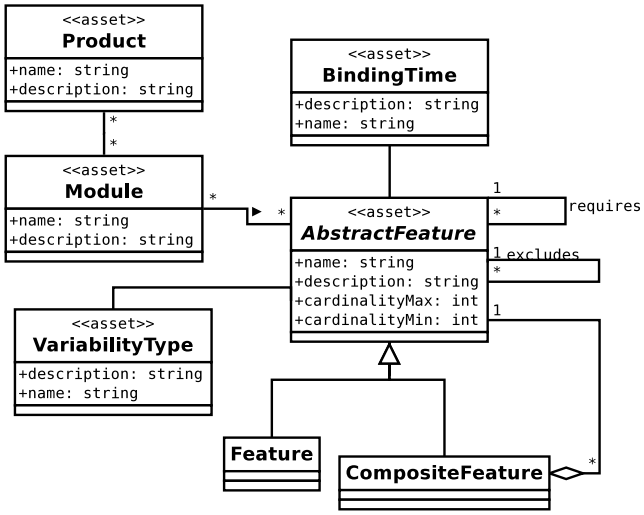

Figure 3: The model for scoping phase.

In the scoping model, a *Feature* object has *BindingTime*, *VariabilityType*, *Module* and *Product* entities associated with it. We did not specify the binding times and variability types for the features, because it must be done when instantiating the metamodel. According to [15], examples of binding time are *before compilation*, *compile time*, *link time*, *load time*, *run-time*; and examples of variability type can be *Mandatory*, *Alternative*, *Optional*, and *Or*.

In the scoping model, the *Feature* objects are grouped into *Module* objects, and *Module* objects are then grouped into *Product* objects. The *Module* objects can be viewed also as the sub-domains of the *Product* objects. It was decided to structure the metamodel in this way since it better represents the SPL project we are developing in the mentioned private company. However, if it is not necessary to have a *Module* entity, it is easy to remove that from the metamodel, since the other phases are not directly linked to the *Module* entity.

3.3 Requirements Model

The metamodel also involves the requirement engineering traceability and interactions issues, considering the variability and commonality in the SPL products. The two main work products from this SPL phase are the requirements and use cases. Figure 4 presents the model regarding the requirements phase.

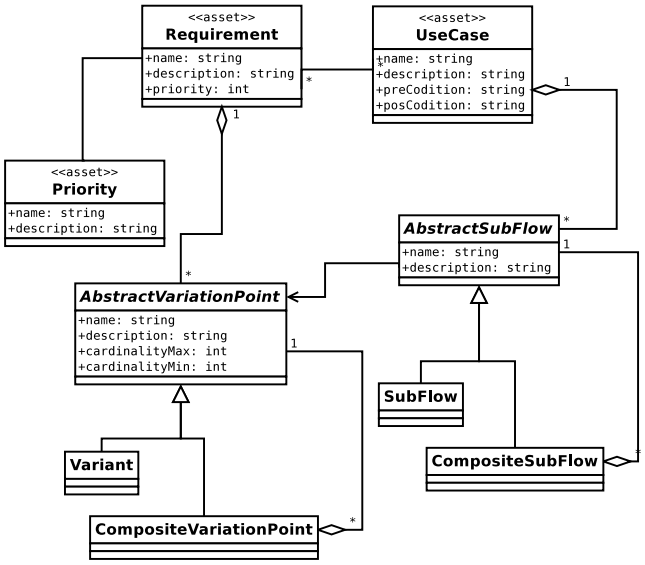

Figure 4: The metamodel for requirements phase.

The *Requirement* object is composed by *name*, *description*, *bindingTime*, *priority*. During its elicitation, it should envisage the variations over the foreseeable SPL life-cycle [9], considering the products requirements and the SPL variations. Briefly, it presents what the systems should do and how they are supposed to be.

Some scoping outputs serve as source of information in this phase. For instance, during the requirements elicitation the feature model is one of the primary artifacts. Features and requirements have a many-to-many relationship, which means that one feature can encompass many requirements and different requirements can encompass many features [17].

It is important to highlight that the metamodel was built in order to address the behavior of SPL projects. In this sense, there are three scenarios where a requirement may be described: *(i)* the requirement is a variation point; *(ii)* the requirement is a variant of a variation point; and *(iii)* the requirement has variation points.

The same scenarios are used when eliciting use cases, it also can be composed by variation points and variants, represented in the model by flow and sub-flow, respectively. In addition, the same many-to-many association is used between requirements and use cases, in which one requirement could encompass many use cases, and a use case could be encompassed by many requirements. The *UseCase* model is composed by *name*, *description*, *preCondition* and *postCondition*. The alternative flows are represented by the flows and sub-flows.

3.4 Tests Model

The metamodel encompasses testing issues in terms of how system test cases interact with other artifacts. The *Test Cases* are derived from *Use Cases*, as can be seen in the metamodel and separated in Figure 5. This model expands on the abstract use case definition, in which variability is represented in the use cases. A use case is herein composed by the entity *AbstractFlow*, that comprises the subentity

Flow, which actually represent the use case steps. Every step can be associated with a *subflow*, that can represent a variation point.

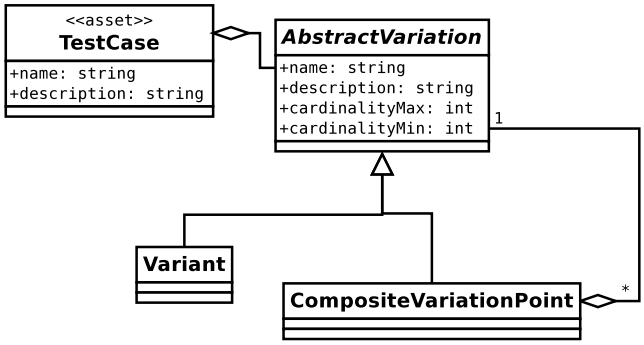

Figure 5: The metamodel for tests.

Figure 6 illustrates the dependency between test objective and test case when variability is considered [23]. Consider that in (A) the component is variable as a whole, and in (B) only part of the component is variable. They will turn, respectively, into (A') and (B'), the former as a new test case, which is variable as a whole, and the latter, a test case in which only the corresponding part of the test case is variable.

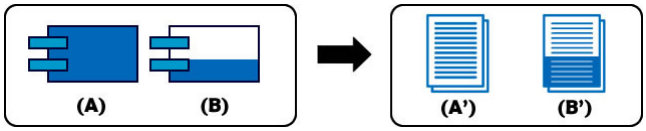

Figure 6: Dependency between test objective and test cases considering variability.

Hence, the challenge is how to optimally build test cases that take into consideration variability aspects so that the reuse of the parts of test cases will emerge easily.

The case illustrated in the Figure 6 is a typical case, in which only part of a use case varies. It is not necessary to create different use cases to represent the variability, but rather reuse part of it. According to the requirements model of the metamodel (Figure 4), such a representation is feasible, since every step in a use case can make reference to a *subflow*.

Consider a hypothetical situation in which there are two variation points, the first representing an *optional feature* (white diamond), and the second representing an *alternative feature* (black diamond), as depicted by the diagram shown in Figure 7. This diagram represents five possible scenarios: (1) [A-B-C-D], (2) [A-B-C-D-E-F], (3) [A-B-C-D-E-G], (4) [A-E-F], (5) [A-E-G]. In this case, if we consider the first three possible scenarios, we could create the test case for scenario (1), and then reuse the flow of this scenario and the results for the remaining scenarios (2) and (3). This idea is brought from the control-flow coverage criteria, based on graph representation. We applied this same representation, but now considering aspects of variability.

Considering that a use case can generate several test cases, the strategy to represent each step in a use case and enable it to be linked to a variation point enables the building of test cases which also consider the variation points. This

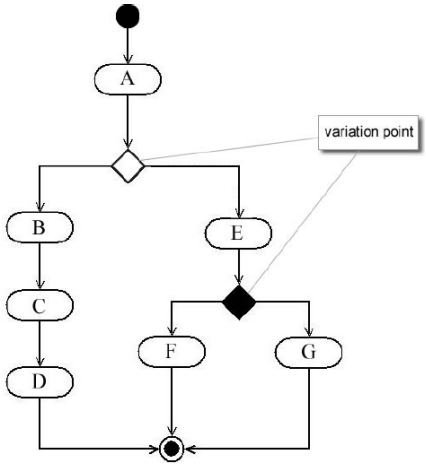

Figure 7: Hypothetical diagram representing variability.

way, several test cases can be instantiated from a use case, that have represented the variation points. Variability is preserved in the core asset test artifacts to facilitate reuse.

This strategy allows that every change in the use case and/or the variation points and variants to be propagated to the test cases and their steps (variable or not), due to the strong traceability represented in the metamodel.

3.5 Initial Management Model

In this section, the main characteristics of the management model are presented. The main objective of this model is to coordinate the SPL activities. Through this model it is possible to manage the SPL project and consequently keep track of its different phases and the staff responsible for that. Thus, it is possible to maintain the mappings and traceability between every artifact.

As illustrated in Figure 8, the management model can be viewed as the starting point to the SPL metamodel. Hence, through this model we can define information about the SPL project as well as details such as the SPL phases, the tasks to be performed, the members responsible for the tasks and their associated roles. In addition, the decisions about the SPL project can be documented using the Decision entity, by describing the alternatives, justification, notes, when it occurred, and the involved staff.

3.5.1 Risks Model

According to [20], Risk Management (RM) is particularly important for software projects because of the inherent uncertainties that most projects face. These stem from loosely defined requirements, difficulties in estimating the effort and resources needed for software development, and dependence on individual skills and requirements changes due to changes in customer needs.

Thus, our RM model for SPL involves activities which must be performed during the RM process that involves all SPL phases. These activities are based on the definition proposed by [20], however it was adapted to our context, based upon the needs with feedback in risk identification stage. The following activities for RM should be performed:

1. *Risk identification:* it is possible to map the risks in the

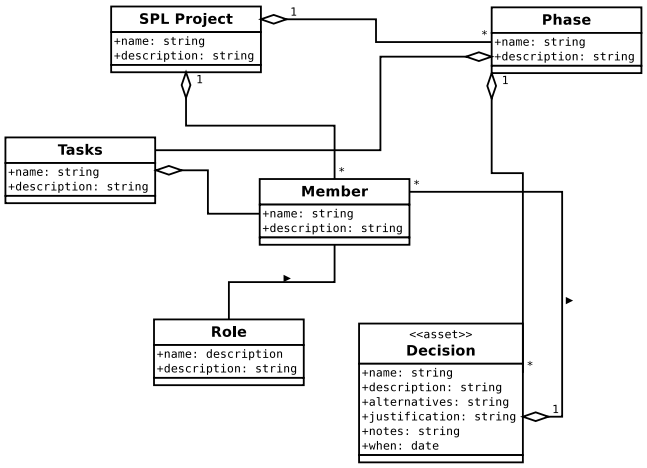

Figure 8: The model for project management.

project, considering product and business risks identification;

2. *Risk Documentation:* identified risks are documented in order to provide the assessment of them;
3. *Risk analysis:* the likely to occur and the consequences of these risks are assessed;
4. *Risk planning:* plans to address the risk either by avoiding it or minimizing its effects on the project are drawn up;
5. *Risk monitoring:* the risk is constantly assessed and plans for mitigation are revised as more information about the risk becomes available.

These activities compose the base for the RM model proposed in our metamodel in a way that may support an automated risk management strategy. Figure 9 shows the functionalities encompassed by the RM model.

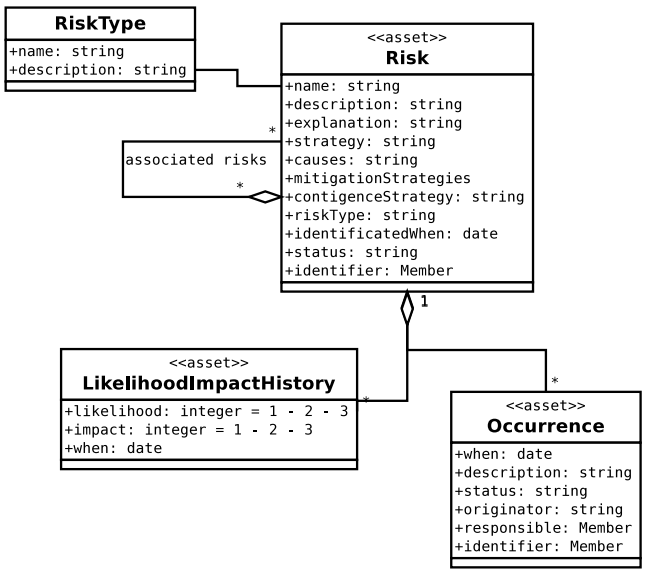

Figure 9: The metamodel for risks management.

According to the RM model, the risks are identified and then their main characteristics are documented. The characteristics are: the risk description, type, status, mitigation strategy, and contingency plans. In addition, as the RM process is a continuous activity, it is necessary to keep track of the history about the management of these risks. Thus, the risks' likelihood and impact are documented according to their occurrence, which can happen in different moments throughout the project development.

4. INITIAL METAMODEL VALIDATION

In order to validate our metamodel, we have implemented a web tool where all the entities from the metamodel are provided and specialized in the tool for the company settings, and the intended metamodel traceability is fully supported. We implemented the tool using the Django[3] framework, which enabled the fast development of a functional prototype. With Django implementation, we mapped the metamodel entities and their relationship within Python[4] classes, and then a relational database for these entities is automatically created. Finally, Django generates a Web application where it is possible to test the mapping by inserting some test data – in our case, the documentation regarding features, requirements and so on.

After this initial application generated by Django, we can extend it by developing new functionalities for the tool. Currently, it is possible to use the tool to document all the assets regarding the metamodel, however the test cases derivation from use cases is not supported yet. Furthermore, the tool is able to produce feature models visualizations, as well as product maps and other type of project documentation. For example, there are several types of reports that can be generated, such as: features per modules/products; all the features, requirements and use cases per modules/products; and the traceability matrix among these different assets.

Additionally, the tool also aids in the inspection activity – static quality assurance activities carried out by a group composed by the authors of the artifacts, a reader and the reviewers – through the generation of different reports and providing a way to gather information about the change requests that should be performed in the inspected artifacts. Moreover, it is important to mention that the tool provides a bug tracker, as proposed by the metamodel, in order to manage the change requests. This also enables the tool to store all the historical data to be further reused and serve as lessons learned in project future activities.

Since the tool provides the traceability proposed in the metamodel, it has also a way to measure the impact analysis of each change request. For example, given a change in a use case, the user opens a change request referencing that asset in order to solve that problem, then the Change Control Board will investigate the change and assign a responsible to fix it. As soon as someone starts to investigate the defect/enhancement, the tool can be asked to inform the impacted artifacts associated with that change in that specific use case. In this way, it can also help the decision regarding when a change should be done in the common platform or in a single SPL product.

The tool has been used inside the company mentioned in the initial sections, since July 2010. As previously stated,

[3] `http://www.djangoproject.com`
[4] `http://www.python.org`

the project goal is to change their single software development process to a SPL process. The tool is currently being used by SPL consultants working embedded in the company to perform the scoping and requirements engineering phases. So far, it was documented 4 products, 9 modules, 97 features, 142 requirements and 156 use cases using the tool. In parallel, designing, implementation and testing are being started.

5. RELATED WORK

We searched the literature related work which addressed *metamodels for SPL* and/or *approaches and tools derived from the metamodels*. Hence, we briefly describe our findings in the following subsections.

5.1 Metamodels

In [21], it is proposed a metamodel to integrate requirements to the feature model, such as the one proposed in [15]. A requirement, in this case, is an indivisible piece of text describing some portion of the system to be developed. The requirements can be described hierarchically, thus achieving variability. Furthermore, the traceability among the entities is based on the metamodel itself.

In [3], a high level metamodel for representing variability in SPL is proposed. The major goal of the metamodel was to "*separate out the representation of variability from the representation of various assets developed in the product development lifecycle while maintaining the traceability between them*".

The authors in [5] proposed a conceptual variability model in order to address traceability of variations at different abstraction levels and across different generic artifacts of the SPL. To achieve that, it is proposed to represent the variability in a third dimension. In such dimension, the variations points would be linked to the generic artifacts (such as requirements, use cases, architecture, etc.) to keep the traceability.

In [8], a metamodel is described to structure variability information across different SPLs. It also based the metamodel in a set of requirements to document requirements variability. [16] introduced two metamodels representing domain requirements and domain architecture with variability, and the traceability between these artifacts are based upon the metamodel.

5.2 Approaches and Tools

In [11], it is briefly described an integrated tool support to develop SPL. Such tool follows some requirements, such as: Domain-specific adaptations, Mining existing assets, Involving multiple teams, Project-specific adaptations, Support for product derivation, Capturing new requirements during derivation, and Supporting product line evolution. However, it is not discussed the metamodel behind it.

Alférez at. al. [1] proposed a model-driven approach for linking features and use cases, along with its activities, and an example about how to apply the approach was showed. The authors did not differentiate between requirements model and use case model, and it is not explicitly described how the flows and sub-flows of the use cases should be handle.

Jirapanthong and Zisman [13] present the XTraQue, which is a rule-based approach to support automatic generation of traceability relations. In the approach, the feature model, use cases and some design documents are represented using XML. Thus, the approach extracts the relationships from these documents automatically using predefined rules.

As it can be seen, the literature basically proposes metamodels or techniques that address only a specific portion of SPL development. In addition, traceability and variability are not always considered together, or the description and the details of them are very simplified. Thus, the main difference between the previous described work and our proposal, is that we present a metamodel, and the implementation of it, for SPL development concerning variability and traceability aspects together along the early phases of SPL, and providing a level of details that simplifies the adoption of the metamodel. Furthermore, the tool that implements our metamodel is currently being used in a industrial project.

6. CONCLUSION AND FUTURE WORK

In this work, we proposed a metamodel for Software Product Lines that encompasses several phases of a SPL development, representing the interactions among the assets of a SPL, developed in order to provide a way of managing traceability and variability, and its initial validation inside a private company. The metamodel was built based upon a real-world SPL project being conducted in a private software company from Brazil. The organization works in the medical information management domain, and have essentialy four products, counting a total of 102 modules (sub-domains), with encompass about 840 features.

The phases currently supported by the metamodel include: scoping, requirements engineering, and tests. Additionally, the metamodel also supports different management aspects of a SPL project. For example, in the current version of the metamodel it is possible to manage different SPL projects concerning the staff, phases and activities, and it is proposed a model for managing risks in SPL.

In the way that we conceived the metamodel, the assets are treated as first-class citizens, which means that we can trace the linkage of any asset to others. Furthermore, treating assets as first-class citizens in our metamodel also enables a set of issues: to keep different versions of the same asset concerning different products in the SPL; to keep the history of assets modifications; to associate any metrics to assets; and it is also possible to manage the defects for different versions of the assets. Furthermore, by incorporating the management model, it is also possible to keep track of different assets and their responsible.

Although the metamodel was conceived to represent the SPL project for a specific company, in which we have been working jointly, it was built to be adaptable to other contexts. For example, the metamodel can be easily changed to support single system development by removing SPL specific entities.

For future work, we intend to extend the metamodel to support more aspects of SPL development. Specifically, we are planning to extend the model for metrics management, detailed software configuration aspects, integrate a model for SPL architecture, and establish mechanisms to link all these artifacts to source code.

We also plan to provide some way to enable the products derivation. The reuse between different SPLs [8] is also intended to be implemented in future releases of the metamodel. In addition, formalized evaluations will be performed, that consider aspects of empirical software engineer-

ing, in order to assess the metamodel effectiveness. Finally, the prototype initially created to support the metamodel should be evolved, as well as formally validated.

Acknowledgment

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[5]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08 and CNPq 474766/2010-1.

7. REFERENCES

[1] M. Alférez, U. Kulesza, A. Moreira, J. Araújo, and V. Amaral. Tracing from features to use cases: A model-driven approach. In *VaMoS'08: Proc. of the 2nd International Workshop on Variability Modeling of Software-intensive Systems*, pages 81–87, 2008. 5.2

[2] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, and A. Sousa. A model-driven traceability framework for software product lines. *Software and Systems Modeling*, 9:427–451, 2010. 1

[3] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A metamodel for representing variability in product family development. In *PFE'03: Proc. of the 5th International Workshop on Software Product-Family Engineering*, pages 66–80, 2003. 1, 3, 5.1

[4] J. Bayer and T. Widen. Introducing traceability to product lines. In *PFE'01: Proc. of 3th International Workshop on Software Product-Family Engineering*, pages 409–416, 2001. 1, 2, 2.1

[5] K. Berg, J. Bishop, and D. Muthig. Tracing software product line variability: from problem to solution space. In *SAICSIT'05: Proc. of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, pages 182–191, Republic of South Africa, 2005. SAICSIT. 5.1

[6] A. Birk and G. Heller. Challenges for requirements engineering and management in software product line development. In *REFSQ'07: Proc. of the 11th International Working Conference on Requirements Engineering*, pages 300–305, Berlin, Heidelberg, 2007. Springer-Verlag. 1

[7] G. Booch, J. E. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, second edition, 2005. 3, 2

[8] S. Bühne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *RE'05: Proc. of the 13th International Conference on Requirements Engineering*, pages 41–52, 2005. 1, 2, 5.1, 6

[9] P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley, Boston, MA, USA, 2001. 1, 3.3

[10] D. Dhungana, P. Grnbacher, and R. Rabiser. The dopler meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, pages 1–38, 2010. 1

[11] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *ASE'07: Proc. of the IEEE/ACM International Conference on Automated Software Engineering*, pages 533–534, 2007. 5.2

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, USA, 1995. 3.2

[13] W. Jirapanthong and A. Zisman. Xtraque: traceability for product line systems. *Software and System Modeling*, 8(1):117–144, 2009. 5.2

[14] I. John and M. Eisenbarth. A decade of scoping: a survey. In *SPLC'09: Proc. of the 13th International Conference on Software Product Lines*, pages 31–40, 2009. 1, 3.2

[15] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. 3.2, 3.2, 5.1

[16] M. Moon, H. S. Chae, T. Nam, and K. Yeom. A metamodeling approach to tracing variability between requirements and architecture in software product lines. In *CIT'2007: Proc. of the 7th IEEE International Conference on Computer and Information Technology*, pages 927–933, University of Aizu, Fukushima Japan, 2007. 1, 5.1

[17] D. F. S. Neiva, E. S. de Almeida, and S. R. de Lemos Meira. An experimental study on requirements engineering for software product lines. In *EUROMIRCRO-SEAA'09: Proc. of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 251–254, 2009. 3.3

[18] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 1

[19] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *SPLC'04: Proc. of the 9th International Software Product Line Conference*, pages 197–213, 2004. 1, 2

[20] I. Sommerville. *Software Engineering*. Addison Wesley, 8 edition, 2007. 3.5.1

[21] D. Streitferdt. Traceability for system families. In *ICSE'01: Proc. of the 23rd International Conference on Software Engineering*, pages 803–804, 2001. 1, 5.1

[22] A. von Knethen and B. Paech. A survey on tracing approaches in practice and research. Technical report, Fraunhofer Institute of Experimental Software Engineering, 2002. 2, 2.1, 3.1

[23] A. Wübbeke. Towards an efficient reuse of test cases for software product lines. In *SPLC'08: Proc. of the International Conference on Software Product Lines*, pages 361–368, 2008. 3.4

[5]http://www.ines.org.br/