We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

1

Handling Variability and Traceability over SPL Disciplines

Yguaratã Cerqueira Cavalcanti[1], Ivan do Carmo Machado[2], Paulo Anselmo da Mota Silveira Neto[1] and Luanna Lopes Lobato[1]
[1]*Federal University of Pernambuco, Reuse in Software Engineering – RiSE*
[2]*Federal University of Bahia, Reuse in Software Engineering – RiSE*
Brazil

1. Introduction

SPL has proven to be a successful approach in many business environments (Clements & Northrop, 2001a; Pohl et al., 2005a). 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 (Birk & Heller, 2007). 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 (Dhungana et al., 2010). In this context, this chapter[1] 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 healthcare management systems domain, placed in Salvador, Brazil. The company currently has a single software development process and develops four different products: *SmartHealth*, a product composed by 35 modules (or sub-domains), 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

[1] This chapter is an extension of our previous work published in VaMoS'11 conference (Cavalcanti et al., 2011)

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* (John & Eisenbarth, 2009) 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[2], 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 (Clements & Northrop, 2001a), 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 (Anquetil et al., 2010; Bachmann et al., 2003; Bayer & Widen, 2001; Bühne et al., 2005; Moon et al., 2007; Sinnema et al., 2004; Streitferdt, 2001) 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 4. 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 chapter is structured as follows: Section 2 presents an overview regarding SPL and its main concepts; Section 3 introduces the traceability conept; Section 4 specifies the requirements for a SPL metamodel; Section 5 describes the actual proposed metamodel, detailing its building blocks; in Section 6 an initial validation of the proposal is presented; Section 7 presents the related work; and finally, Section 8 concludes this work and outlines future work.

2. SPL essential activities

Software Product Lines combine three essential and highly iterative activities that blend business practices and technology. Firstly, the *Core Asset Development (CAD)* activity that does not directly aim at developing a product, but rather aims to develop assets to be further reused in other activities. Secondly, *Product Development (PD)* activity which takes advantage of existing, reusable assets. Finally, *Management* activity, which includes technical and organizational management (Linden et al., 2007). Figure 1 illustrates this triad of essential activities.

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

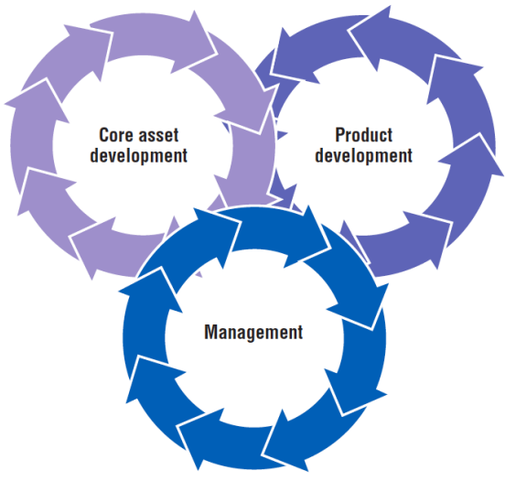

Fig. 1. Essential product line activities (Northrop, 2002).

2.1 Core Asset Development

Core Asset Development is the life-cycle that results in the common assets that in conjunction compose the product line's platform (Linden et al., 2007). The key goals of this activity are (Pohl et al., 2005b):

- Define variability and commonality of the software product line;
- Determine the set of product line planned members (scope); and
- Specify and develop reusable artifacts that accomplish the desired variability and further instantiated to derive product line members.

This activity (Figure 2) is iterative, and its inputs and outputs affect each other. This context influences the way in which the core assets are produced. The set of inputs needed to accomplish this activity are following described (Northrop, 2002). *Product constraints* commonalities and variations among the members that will constitute the product line, including their behavioral features; *Production constraints* commercial, military, or company-specific standards and requirements that apply to the products in the product line; *Styles, patterns, and frameworks* relevant architectural building blocks that architects can apply during architecture definition toward meeting the product and production constraints; *Production strategy* the whole approach for realizing the core assets, it can be performed starting with a set of core assets and deriving products (top down), starting from a set of products and generalizing their components in order to produce product line assets (bottom up) or both ways; *Inventory of preexisting assets* software and organizational assets (architecture pieces, components, libraries, frameworks and so on) available at the outset of the product line effort that can be included in the asset base.

Based on previous information (inputs), this activity is subdivided in five disciplines: *(i)* domain requirements, *(ii)* domain design, *(iii)* domain realization (implementation), *(iv)* domain testing and *(v)* evolution management, all of them administered by the management activity (Pohl et al., 2005b). These disciplines are responsible for creating the core assets, as well as, the following outputs (Figure 2) (Clements & Northrop, 2001b): *Product line scope* the description of the products derived from the product line or that the product line is capable of

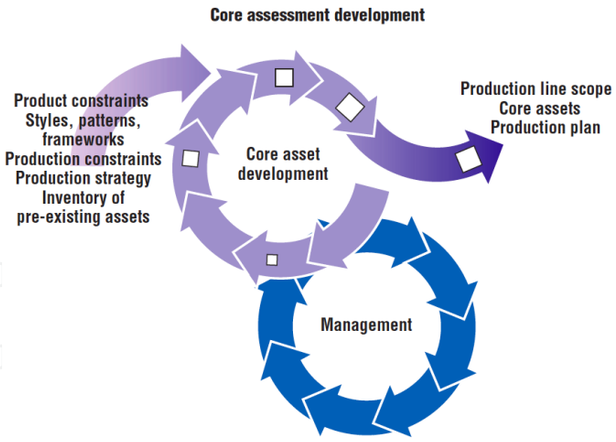

Fig. 2. Core Asset Development (Northrop, 2002).

including. The scope should be small enough to accommodate future growth and big enough to accommodate the variability. *Core assets* comprehend the basis for production of products in the product line, besides the reference architecture, that will satisfy the needs of the product line by admitting a set of variation points required to support the spectrum of products, theses assets can also be components and their documentation. The *Production plan* describes how the products are produced from core assets, it also describe how specific tools are to be applied in order to use, tailor and evolve the core assets.

2.2 Product development

The product development main goal is to create individual (customized) products by reusing the core assets previously developed. The CAD outputs (product line scope, core assets and production plan), in conjunction with the requirements for individual products are the main inputs for PD activity (Figure 3).

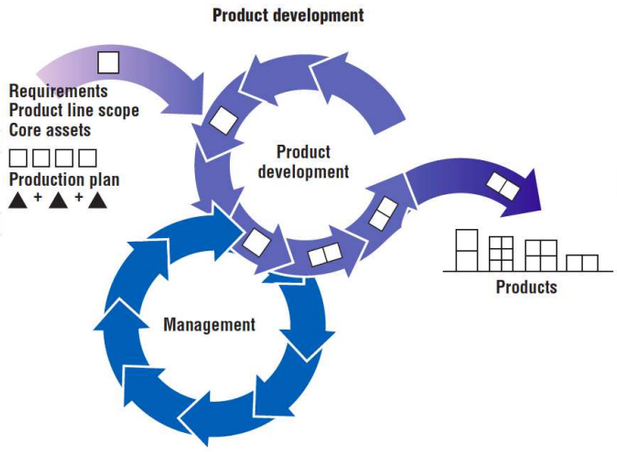

Fig. 3. Product Development (Northrop, 2002).

In possession of the production plan, which details how the core assets will be used in order to build a product, the software engineer can assemble the product line members. The

product requirement is also important to realize a product. Product engineers have also the responsibility to provide feedback on any problem or deficiency encountered in the core assets. It is crucial to avoid the product line decay and keep the core asset base healthy.

2.3 Management

The management of both technical and organizational levels are extremely important to the software product line effort. The former supervise the CAD and PD activities by certifying that both groups that build core assets and products are engaged in the activities and to follow the process, the latter must make sure that the organizational units receive the right and enough resources. It is, many times, responsible for the production strategy and the success or failure of the product line.

2.4 SPL variability management

During Core Asset Development, variability is introduced in all domain engineering artifacts (requirements, architecture, components, test cases, etc.). It is exploited during Product Development to derive applications tailored to the specific needs of different customers.

According to Svahnberg et al. (2005), variability is defined as *"the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"*. It is described through variation points and variants. While, the variation point is the representation of a variability subject (variable item of the real world or a variable property of such an item) within the core assets, enriched by contextual information; the variant is the representation of the variability object (a particular instance of a variability subject) within the core assets (Pohl et al., 2005b).

The variability management involve issues, such as: variability identification and representation, variability binding and control (de Oliveira et al., 2005). Three questions are helpful to variability identification, *what vary* the variability subject, *why does it vary* the drivers of the variability need, such as stakeholder needs, technical reasons, market pressures, etc. The later, *how does it vary* the possibilities of variation, also known as variability objects.

The variability binding indicates the lifecycle milestone that the variants related with a variation point will be realized. The different binding times (e.g.: link, execution, post-execution and compile time) involves different mechanisms (e.g.: inheritance, parameterization, conditional compilation) and are appropriate for different variability implementation schemes. The different mechanisms result in different types of defects, test strategies, and test processes (McGregor et al., 2004).

Finally, the purpose of variability control is to defining the relationship between artifacts in order to control variabilities.

3. SPL traceability

It most organizations and projects with mature software development processes, software artifacts created during the application of these processes end up being disconnected from each other. There are several factors leading to the lack of traceability among artifacts, as stated by Rilling et al. (2007). They are: (i) these artifacts may be written in different languages (natural language vs. programming language); (ii) the system is described by considering

various abstraction levels/views (scoping and requirements vs. design or implementation); (iii) software processes do not consider maintenance of existing traceability links as a "must have" practice; and also (iv) there is a lack of adequate tool support to create and maintain traceability among software artifacts.

The lack of traceability among artifacts can be considered a major challenge for many software maintenance activities (Rilling et al., 2007). As a result, during the comprehension of existing software systems, software engineers are likely to spend an enormous effort on synthesizing and integrating information from various sources to establish links among these artifacts.

Indeed, by establishing traceability, engineers have the opportunity to understand how software artifacts interact with each other, in terms of relations and dependencies. Traceability links are indeed helpful when considering the evolutionary characteristic of the software artifacts, that are likely to be changed during its lifecycle.

According to Anquetil et al. (2010), establishing traceability yield a series of benefits, such as: (i) to relate software artifacts and corresponding design decisions, (ii) to give feedback to architects and designers about the current state of the development, allowing them to reconsider alternative design decisions, and to track and understand errors, and (iii) to ease communication between stakeholders, among others.

In SPL, where assets can be used by many products, traceability is even more important. A change in an artifact in a product line may lead to resultant changes in all products developed reusing such an artifact. Hence, it is necessary to define dependency relationships between the relevant artifacts to support consistent change integration (Moon et al., 2007).

Traceability practices vary widely-from high-end to low-end (Ramesh & Jarke, 2001). The prior use customized traceability knowledge for managing variability. Project managers may select traceability practices based on project characteristics such as system complexity, product line vs. single system software development, and degree of variety involved. The later take more of a "one-size fits all" approach to documenting traceability knowledge, and create simple traceability links between customer requirements, design, and code modules. While high-end practices aim at customer satisfaction and system quality, low-end are used just to meet organizational or customer-mandated quality requirements (Mohan & Ramesh, 2007).

4. Requirements for the SPL metamodel

We identified some work (Bayer & Widen, 2001; Bühne et al., 2005; Sinnema et al., 2004; von Knethen & Paech, 2002) 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 (Sinnema et al., 2004), 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 metal-model are considered from the initial SPL phases:

Uniform and first-class representation of variation points in all abstraction levels. Sinnema et al. (2004) stated 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 (Sinnema et al., 2004), 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 (Bühne et al., 2005), 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. Bühne et al. (2005) stated 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.

4.1 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 von Knethen & Paech (2002), 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 between requirement and design (von Knethen & Paech, 2002). 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 (Bayer & Widen, 2001), 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.

5. 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 (Booch et al., 2005). The metamodel was initially divided into *SPL management*, which currently includes the risk management sub-metamodel, and the *SPL core development*, which are the scoping, requirement, and tests sub-metamodels. Henceforth, we will call the submetamodels just by models.

Figure 4 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 Bachmann et al. (2003). 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.

5.1 Asset UML profile

This is the metamodel core entity, called *Asset*, as shown in Figure 5. This entity is used as a UML profile[3] 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 (von Knethen & Paech, 2002).

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.

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

[3] UML profiles are an extension mechanism of the UML language (Booch et al., 2005) that allow models to be customized for specific domains.

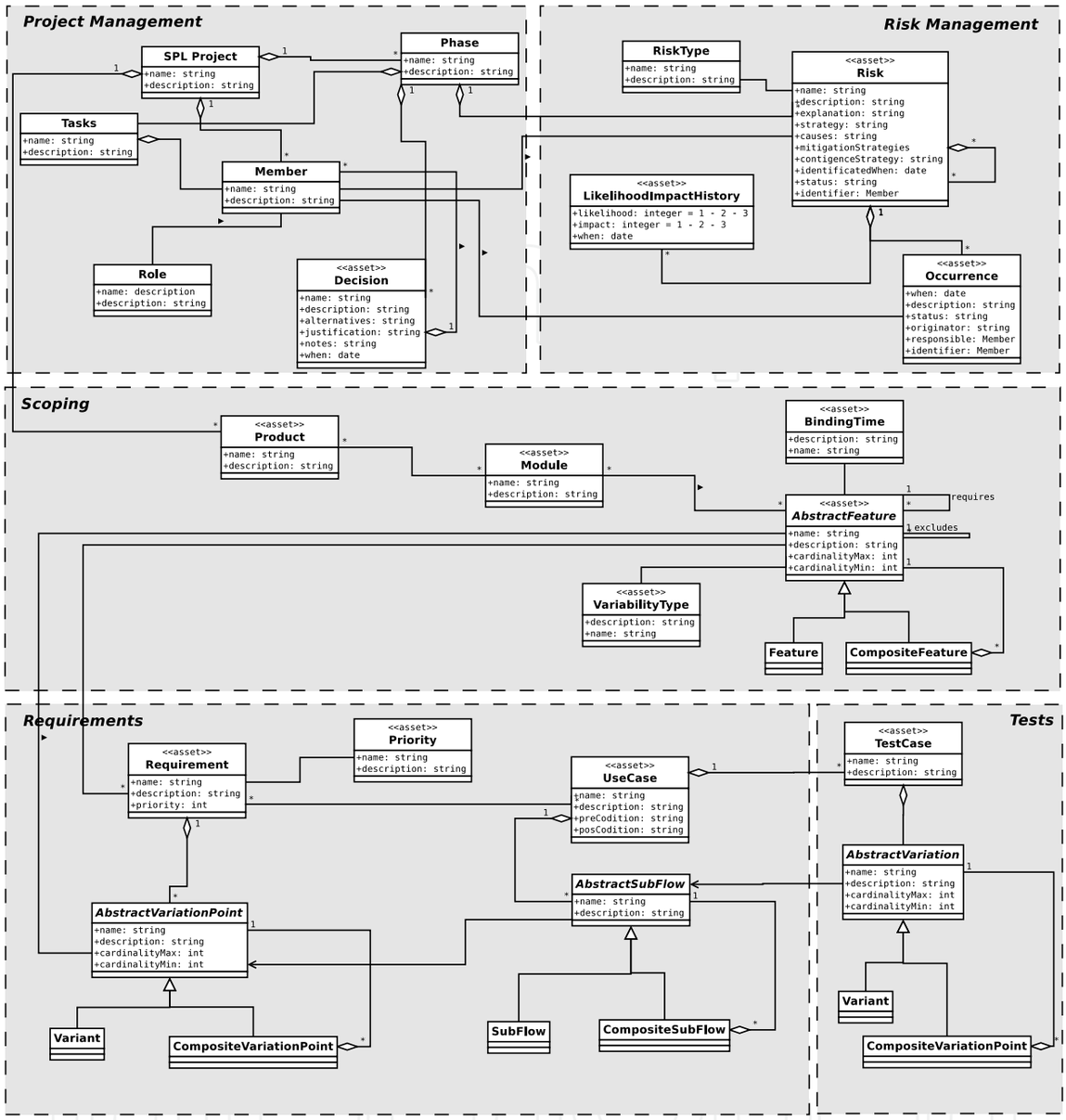

Fig. 4. Overview of the proposed SPL metamodel.

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

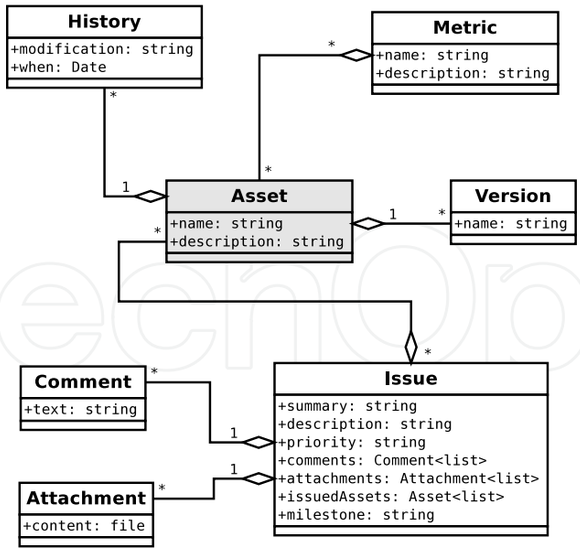

Fig. 5. The metamodel core UML profile.

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.

5.2 Scoping model

During the scoping phase (John & Eisenbarth, 2009), the feature model is assembled based on the method to identify those features (Kang et al., 1990). In some cases, a tree is built containing all feature types, such as mandatory, alternative and optional. The scoping model is shown in Figure 6.

We used the Composite design pattern (Gamma et al., 1995) to represent the feature model, since it is a good representation using UML diagrams for the feature model proposed by Kang et al. (1990). 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.

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 Kang et al. (1990), 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.

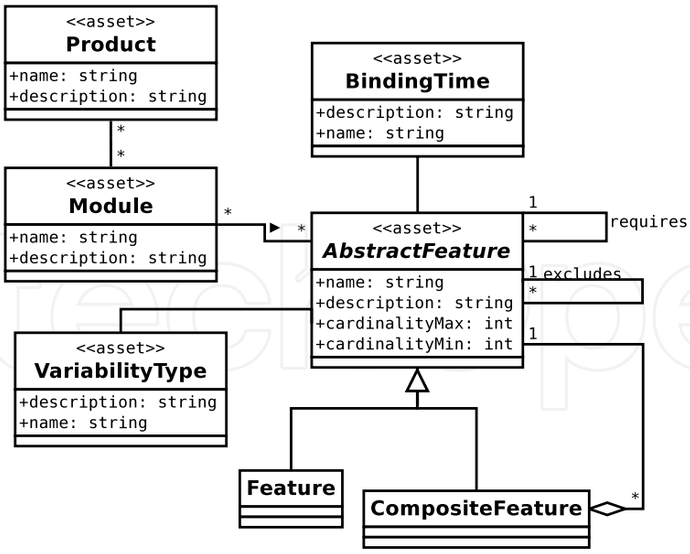

Fig. 6. The model for scoping phase.

5.3 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 7 presents the model regarding the requirements phase.

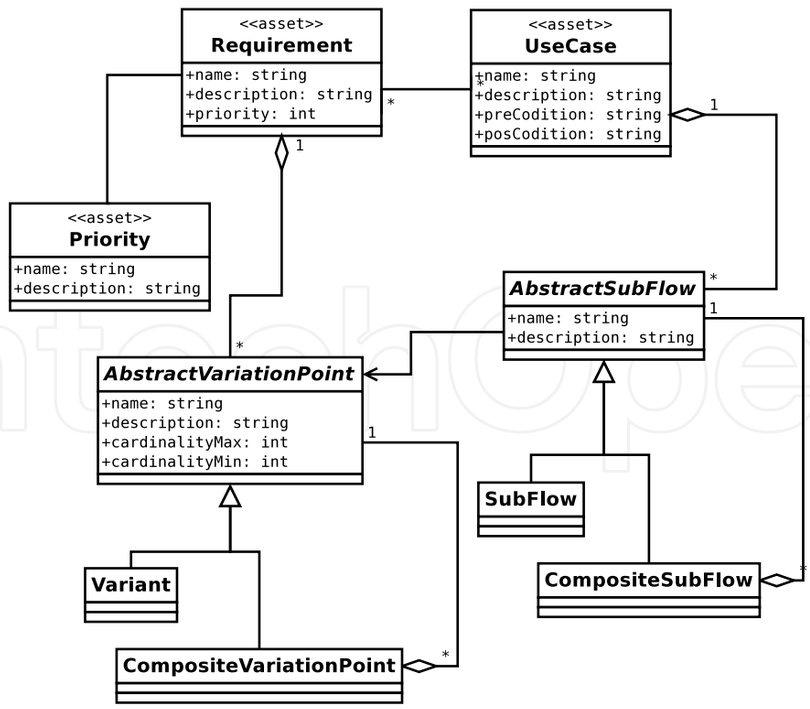

Fig. 7. The metamodel for requirements phase.

The *Requirement* object is composed by *name*, *description*, *bindingTime*, *priority*. During its elicitation, it should envisage the variations over the foreseeable SPL life-cycle (Clements & Northrop, 2001a), 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 (Neiva et al., 2009).

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.

5.4 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 8. 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.

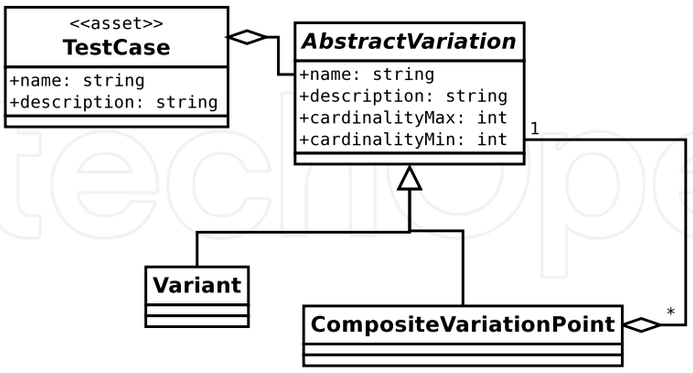

Fig. 8. The metamodel for tests.

Figure 9 illustrates the dependency between test objective and test case when variability is considered (Wübbeke, 2008). 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.

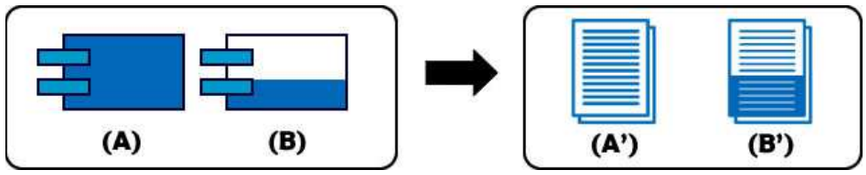

Fig. 9. Dependency between test objective and test cases considering variability.

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 9 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 7), 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 10. 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.

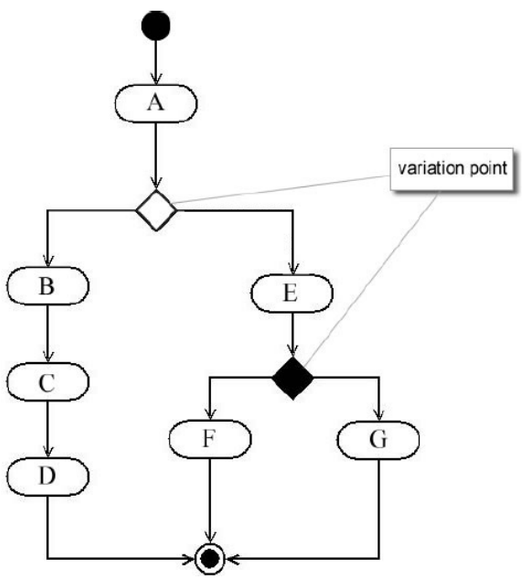

Fig. 10. Hypothetical diagram representing variability.

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 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.

5.5 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 11, 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.

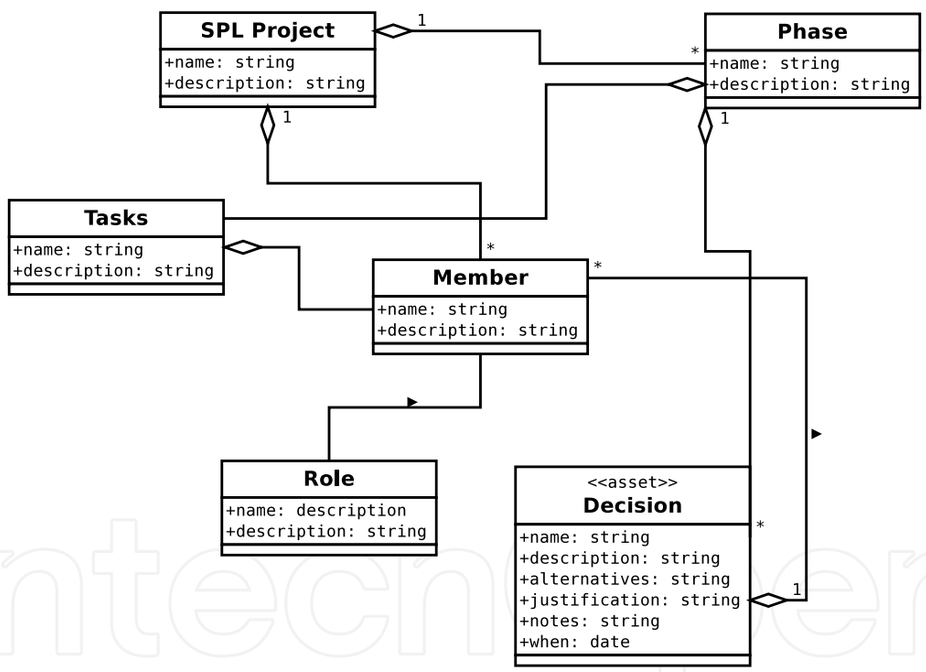

Fig. 11. The model for project management.

5.5.1 Risks model

According to Sommerville (2007), 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 Sommerville (2007), 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 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 12 shows the functionalities encompassed by the RM model.

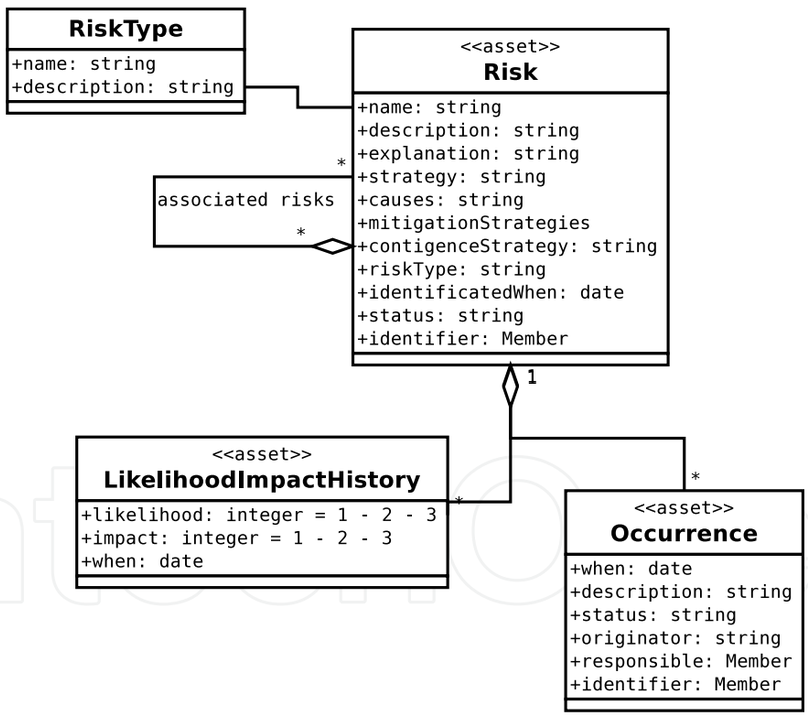

Fig. 12. The metamodel for risks management.

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. It is important to emphasize that our approach

to risk management should be performed in the essential activities of the SPL: Core Assets Development (CAD), Product Development (PD) and Management (M).

6. Implementing and validating the metamodel

In order to validate the 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[4] framework, which enabled the fast development of a functional prototype. With Django implementation, we mapped the metamodel entities and their relationship within Python[5] 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, 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.

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

7. Others metamodels, approaches and tools

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.

7.1 Metamodels

In (Streitferdt, 2001), it is proposed a metamodel to integrate requirements to the feature model, such as the one proposed in (Kang et al., 1990). 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 (Bachmann et al., 2003), 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 (Berg et al., 2005) 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 (Bühne et al., 2005), 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. Moon et al. (2007) introduced two metamodels representing domain requirements and domain architecture with variability, and the traceability between these artifacts are based upon the metamodel.

7.2 Approaches and tools

In (Dhungana et al., 2007), 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 et al. (2008) 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 & Zisman (2009) presented 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.

8. Conclusion and future work

In this chapter, it was 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 healthcare management systems domain, and have essentially 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 (Bühne et al., 2005) 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 engineering, in order to assess the metamodel effectiveness. Finally, the prototype initially created to support the metamodel should be evolved, as well as formally validated.

9. References

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

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

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

Bayer, J. & Widen, T. (2001). Introducing traceability to product lines, *PFE'01: Proc. of 3th International Workshop on Software Product-Family Engineering*, pp. 409–416.

Berg, K., Bishop, J. & Muthig, D. (2005). Tracing software product line variability: from problem to solution space, *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*, SAICSIT, Republic of South Africa, pp. 182–191.

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

Booch, G., Rumbaugh, J. E. & Jacobson, I. (2005). *The Unified Modeling Language User Guide*, second edn, Addison Wesley.

Bühne, S., Lauenroth, K. & Pohl, K. (2005). Modelling requirements variability across product lines, *RE'05: Proc. of the 13th International Conference on Requirements Engineering*, pp. 41–52.

Cavalcanti, Y. C., do Carmo Machado, I., da Mota Silveira Neto, P. A., Lobato, L. L., de Almeida, E. S. & de Lemos Meira, S. R. (2011). Towards metamodel support for variability and traceability in software product lines, *Proceedings of the Fifth International WOrkshop on Variability Modeling of Software-Intensive Systems (VaMoS'2011)*, Namur, Belgium, pp. 49–57.
URL: *http://yguarata.com/blog/wp-content/uploads/2011/01/vamos2011_submission_30.pdf*

Clements, P. & Northrop, L. (2001a). *Software product lines: practices and patterns*, Addison-Wesley, Boston, MA, USA.

Clements, P. & Northrop, L. (2001b). *Software Product Lines: Practices and Patterns*, Addison-Wesley, Boston, MA, USA.

de Oliveira, Junior, E. A., Gimenes, I. M. S., Huzita, E. H. M. & Maldonado, J. C. (2005). A variability management process for software product lines, *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pp. 225–241.

Dhungana, D., GrALnbacher, P. & Rabiser, R. (2010). The dopler meta-tool for decision-oriented variability modeling: a multiple case study, *Automated Software Engineering* pp. 1–38.

Dhungana, D., Rabiser, R., Grünbacher, P. & Neumayer, T. (2007). Integrated tool support for software product line engineering, *ASE'07: Proc. of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 533–534.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, Boston, MA, USA.

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

John, I. & Eisenbarth, M. (2009). A decade of scoping: a survey, *SPLC'09: Proc. of the 13th International Conference on Software Product Lines*, pp. 31–40.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. & Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study, *Technical report*, Carnegie-Mellon University Software Engineering Institute.

Linden, F. J. v. d., Schmid, K. & Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

McGregor, J., Sodhani, P. & Madhavapeddi, S. (2004). Testing Variability in a Software Product Line, *SPLIT '04: Proceedings of the International Workshop on Software Product Line Testing*, Boston, Massachusetts, USA, p. 45.

Mohan, K. & Ramesh, B. (2007). Tracing variations in software product families, *Communications of the ACM* 50: 68–73.

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

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

Northrop, L. M. (2002). Sei's software product line tenets, *IEEE Software* 19(4): 32–40.

Pohl, K., Böckle, G. & Linden, F. J. v. d. (2005a). *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Pohl, K., Böckle, G. & Linden, F. J. v. d. (2005b). *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag, Secaucus, NJ, USA.

Ramesh, B. & Jarke, M. (2001). Toward reference models for requirements traceability, *IEEE Trans. Softw. Eng.* 27: 58–93.
URL: *http://dl.acm.org/citation.cfm?id=359555.359578*

Rilling, J., Charland, P. & Witte, R. (2007). Traceability in Software Engineering – Past, Present and Future, *Technical Report TR-74-211*, IBM Technical Report, CASCON 2007 Workshop.

Sinnema, M., Deelstra, S., Nijhuis, J. & Bosch, J. (2004). Covamof: A framework for modeling variability in software product families, *SPLC'04: Proc. of the 9th International Software Product Line Conference*, pp. 197–213.

Sommerville, I. (2007). *Software Engineering*, 8 edn, Addison Wesley.

Streitferdt, D. (2001). Traceability for system families, *ICSE'01: Proc. of the 23rd International Conference on Software Engineering*, pp. 803–804.

Svahnberg, M., van Gurp, J. & Bosch, J. (2005). A taxonomy of variability realization techniques: Research articles, *Software Practice and Experience* 35(8): 705–754.

von Knethen, A. & Paech, B. (2002). A survey on tracing approaches in practice and research, *Technical report*, Fraunhofer Institute of Experimental Software Engineering.

Wübbeke, A. (2008). Towards an efficient reuse of test cases for software product lines, *SPLC'08: Proc. of the International Conference on Software Product Lines*, pp. 361–368.

Software Product Line - Advanced Topic

Edited by Dr Abdelrahman Elfaki

ISBN 978-953-51-0436-0
Hard cover, 122 pages
Publisher InTech
Published online 04, April, 2012
Published in print edition April, 2012

The Software Product Line (SPL) is an emerging methodology for developing software products. Currently, there are two hot issues in the SPL: modelling and the analysis of the SPL. Variability modelling techniques have been developed to assist engineers in dealing with the complications of variability management. The principal goal of modelling variability techniques is to configure a successful software product by managing variability in domain-engineering. In other words, a good method for modelling variability is a prerequisite for a successful SPL. On the other hand, analysis of the SPL aids the extraction of useful information from the SPL and provides a control and planning strategy mechanism for engineers or experts. In addition, the analysis of the SPL provides a clear view for users. Moreover, it ensures the accuracy of the SPL. This book presents new techniques for modelling and new methods for SPL analysis.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Yguarată Cerqueira Cavalcanti, Ivan do Carmo Machado, Paulo Anselmo da Mota Silveira Neto and Luanna Lopes Lobato (2012). Handling Variability and Traceability over SPL Disciplines, Software Product Line - Advanced Topic, Dr Abdelrahman Elfaki (Ed.), ISBN: 978-953-51-0436-0, InTech, Available from: http://www.intechopen.com/books/software-product-line-advanced-topic/handling-variability-and-traceability-over-spl-disciplines

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen