Feature Assembly Framework: Towards Scalable and Reusable Feature Models

Lamia Abo Zaid
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
+32 2 629 3754
Lamia.Abo.Zaid@vub.ac.be

Frederic Kleinermann
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
+32 2 629 5713
Frederic.Kleinermann@vub.ac.be

Olga De Troyer
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels
+32 2 629 3504
Olga.DeTroyer@vub.ac.be

ABSTRACT

Feature models have been commonly used to model the variability in software product lines. In this paper we present the Feature Assembly framework which is a new approach for creating feature models through feature composition and feature assembly. Furthermore, it promotes feature reusability by storing features in a so-called feature pool, which acts as a feature repository. The Feature Assembly Framework is based on the Feature Assembly feature modeling method, which will be briefly introduced. It is a multi-perspective approach for modeling variability, to deal with the complexity of large systems. The feature assembly modeling method also provides a simpler and easier to use modeling language, which separates the variability specifications from the feature specifications to allow reusing features in different contexts.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software – *Domain Engineering, Reuse Models*

General Terms

Management, Design, Economics.

Keywords

Feature Analysis, Variability, Reuse, Feature Assembly, Feature Model, Feature Diagram, Perspective, Viewpoint, Feature Composition.

1. INTRODUCTION

A software product line (SPL) [1] is a collection of software products that has a common architecture that supports variant set of features (also called Software Product Families [2]). Feature is a lightweight term that refers to a capability of the system or artifact. For this reason, the notion of feature is very convenient for modeling Software Product Lines. Some features are required by all members of the product family and are thus common to all members of the product line, while each member of the product line has a set of distinguishing features (i.e. variable features). Different combinations of features allow constructing different products from a single product line.

To be able to profit maximally from the benefits of software product lines, but also to keep the development of such software under control, feature-oriented domain analysis should be adopted early in the development process to effectively identify and characterize the product line features (i.e. capabilities and functionalities). In feature oriented domain analysis (FODA), features are abstractions that different stakeholders can understand [3]. Feature models [3] [4] are used to represent the commonalities and differences in the features provided by the product line.

A software product line often undergoes adjustments to meet the continuous changes in customer and market requirements. Keeping this under control at an affordable cost is still a major problem. Increasing the scope and diversity of the products that the product line delivers, results in several serious problems both at the domain analysis level[1] (i.e. modeling level) (see e.g., [5] [6]) and at the architecture level (see e.g., [7] [8]). As the product line matures, its scope may significantly widen due to the introduction of new features. This causes on the one hand a decrease in the complete commonality (i.e. the features that are common to all members of the product line) of its products, and on the other hand, an increase in the partial commonality (i.e. the features that are common to a subset of the members of the product line) of its products [8]. It could even be the case that a company that started with one product line evolves to having multiple related product lines. However, current feature models are not flexible enough to cope with continuous changing requirements, furthermore they do not scale well (we will discuss this in more detail in section 2).

In this paper, we present a new approach for creating feature models. The presented approach is based on creating feature

[1] In this paper we restrict ourselves to the effect of change at a domain analysis stage.

models by composing and reusing features. Initially, as a product line is defined, a feature model(s) representing it is specified. These features are then stored in a *feature pool* for later reuse. The pool of features allows for creating different product lines by reusing already existing features. In addition, whenever an existing product line undergoes a change in its scope or requirements or a new product line is needed, new features can be introduced and added to the pool for later reuse. We call this approach *Feature Assembly Framework*. The new approach is based on a new revised feature modeling method (*Feature Assembly Modeling Technique*) that addresses the above-mentioned needs.

This paper is organized as follows: in section 2 we discuss the limitations of current feature modeling techniques. In section 3, we present our Feature Assembly Framework, and in section 4, we briefly explain the associated feature modeling method. Next, section 5 provides an example that illustrates the approach and its benefits. In section 6 we discuss related work. Section 7 provides the conclusion and discusses future work.

2. LIMITATIONS OF CURRENT FEATURE MODELING

In feature-oriented analysis, the product line capabilities and functionalities are identified and characterized. Feature models [3] [4] are used to model the commonality and variability in the features provided by the product line. They do so by means of a hierarchical (tree-based) representation of the features that make up the product line. This poses several problems, firstly in top down approaches the problem domain should be *fully* understood to be able to decompose the problem into smaller problems. Secondly, in feature models, the top down decomposition of features is implicitly based on both functional decomposition and variability decomposition. Not having a clear distinction between two fundamentally different types of relations, i.e. functional decomposition and variability decomposition, makes the modeling process difficult and is a source of errors [9] [10]. Thirdly, the hierarchical top down decomposition structure adopted in these feature models makes maintenance more difficult (due to a significant amount of backtracking), and reduce and hinder reusability. It has been shown in [11] that top down modeling approaches are not appropriate for reuse. For more efficient modeling of product lines a balance is required between top down (decomposition) and bottom up (compositional) approaches for supporting reusability with variability, as has been argued in [12].

Additionally, with the large number of features in today's software, feature models have shown to suffer from a scalability problem [13]. Creating a feature model with thousands of features and managing their dependencies is not an easy task. Not only is the creation of large feature models difficult, but also their modification and maintenance is a complex task [5] [14]. In addition, the added value of a graphical representation is lost as the trees easily become very large and don't provide an easy overview anymore.

In current feature models, a feature is given a *type* that indicates how the feature contributes to the variability of the system. This limits the possibility to reuse the same feature in a different context with different variability requirements, as the type may need to be different. For example, a *bank transfer payment* feature may be mandatory in one setting while optional in another (e.g., depending on the target market or country). As the type (here mandatory or optional) is inextricably associated with the feature, it will not be possible to reuse the feature as it is. In addition, change is also an issue. It is quite difficult to add new features or change an existing feature (e.g., change its variability type). For example, a *Payment* feature may have two alternatives *Bank Transfer* and *PayPal* (*Alternative Features*), when targeting new markets this feature may need to be extended with other payment methods (e.g., *Visa, Mastercard*, and *Bancontact/Mister Cash)*. Furthermore, suppose that the *Bank Transfer* feature needs to become mandatory to suit all markets, while there is a need to select one or more of the other payment features (*OR Features*). Such a change requires deleting the old Alternative Feature group, creating a new OR group, and changing the type of the *Bank Transfer* feature to mandatory. Note that adding and removing branches in the feature model tree may not always be a straightforward task in current feature modeling tools (e.g. it may need backtracking and reconstruction of more than one branch or even level).

The above-mentioned observations have given rise to the development of a new approach for feature modeling. The presented approach is based on reusing and composing features. It supports creating feature models for product lines by assembling features from an existing and continuously growing pool of features. To achieve this, a rigorous hybrid methodology that combines both a top down and a bottom up approach is adopted and a revised feature modeling technique was defined. To overcome the problem of scalability, abstraction mechanisms for feature models are introduced. This feature assembly approach is mainly intended for the feature modeling of large systems that will evolve or grow over time.

The next section will explain the Feature Assembly Framework.

3. FEATURE ASSEMBLY MODELING FRAMEWORK

Feature Assembly aims at modeling variable software by assembling together new features as well as previously defined features. Feature Assembly Framework allows reusing features from a repository of features that we call *Feature Pool*. Feature Assembly is a feature-oriented modeling framework. In contrast with existing feature modeling techniques it defines features independent from how they (directly) contribute to the variability of a specific product line. Rather a feature is defined based on whether it represents a concrete capability provided by the product line or as a specification of some abstract capability (more details in section IV). This is essential to be able to reuse features in different contexts. The information of how a certain feature contributes to the variability of a particular product is only relevant when the feature is actually assembled with other features to model a specific product line. As an example, two product lines may share some features, but may have different restrictions on how the features could be selected/ deselected in certain products. Therefore, in Feature Assembly, how a feature contributes to the variability of a specific product line or product is not inextricably associated with the feature. This information is rather part of the feature assembly model that specifies a specific software product line. Therefore, how the features contribute to variability is only expressed in the feature assembly model, not in the feature pool.

Figure 1 illustrates the Feature Assembly Framework. Features in the feature pool do not refer directly to code but are rather abstract representations of software features. They are stored together with some metadata (annotations) to facilitate searching for appropriate features. The metadata holds information like: a description, a rationale, the product lines in which this feature is used, the date of creation, collaborators (i.e. the stakeholders involved with it), and owner(s) (i.e. the stakeholder(s) that defined it). In addition, relations and dependencies between features, which are independent of the context of use (but inherently connected to the features), are also stored. Such as composition relations of a feature, variants of a feature, and feature dependencies (such as includes and excludes).For example, in a quiz application (see section V for the full explanation), the dependency that a *Rich Editor* '*requires*' a *Basic Editor* is always true (i.e. for any application that will use the Rich Editor), therefore it is saved in the feature pool as a *property* of the *Rich Editor* feature.

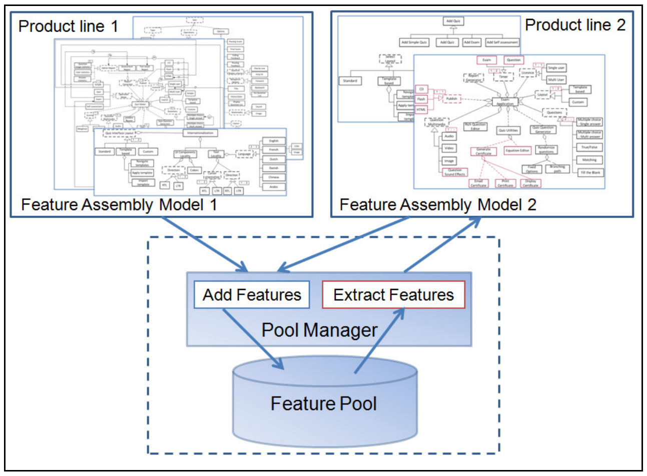

Figure 1. Feature Assembly Framework Overview

Initially the pool is empty, and gets populated with the features that define the first product line (or product). The mechanism of the Feature Assembly Framework is shown in figure 2. When a new product line (or even product) is required, the product line requirements are analyzed to determine the new product line variability and commonality. The new product line's features are then identified. It will be investigated whether the required features already exist in the feature pool or whether they are new and thus need to be defined. This is done via searching the feature pool using any of the metadata associated with the features. For example, by issuing a search query that contains a specific feature attribute, rationale, a specific word in the description, or annotation. Existing features are extracted from the feature pool (possibly with their appropriate descendents; this is entirely up to the modeler and differs from one case to another). Generally, for a feature that is composed of finer grained subfeatures, the subfeatures will also be extracted as long as they represent a mandatory part of their parent feature. Likewise, a feature that has many variant features associated with it, the modeler may extract a selection of variant features as not all may be relevant for his purpose.

New features are defined with the appropriate level of detail. A feature assembly model that represents the required new product line is created combining both the new features and the existing ones. In addition, the newly created features are added to the feature pool along with their metadata.

This process allows the continuous growth of the feature pool in addition to feature reuse. In the next section, we will briefly explain the Feature Assembly Modeling technique used for constructing the feature assembly models.

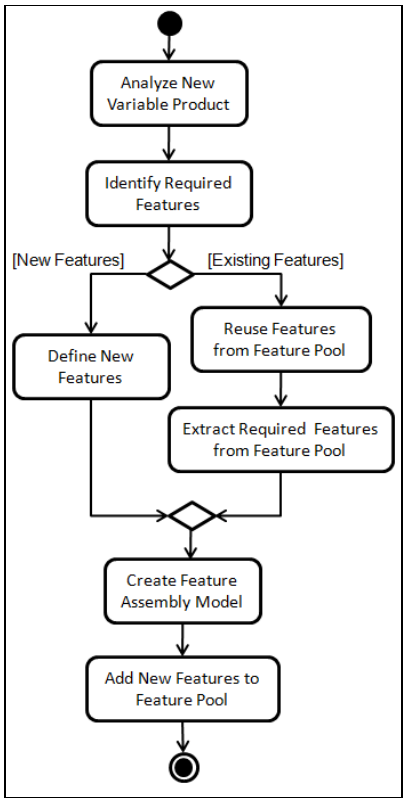
Analyze New Variable Product
Identify Required Features
[New Features]
[Existing Features]
Define New Features
Reuse Features from Feature Pool
Extract Required Features from Feature Pool
Create Feature Assembly Model
Add New Features to Feature Pool

Figure 2. Feature Assembly Process for Feature Model Assembly

4. FEATURE ASSEMBLY MODELING TECHNIQUE

The idea of creating feature models by assembling features, whether already existing or new, is made possible by using a flexible feature modeling technique, called Feature Assembly Modeling, that allows abstracting the features from how they relate to variability. In our previous work [15], we have defined this Feature Assembly Modeling technique[2]. The modeling technique is intended to model large and complex variable software during analysis and/or design. To deal with the scalability issue, we provide the modeler with different abstraction

[2] We give a brief overview of the approach here to help the reader understand the example that demonstrates the feature assembly framework.

viewpoints or so-called perspectives. Modeling software can be considered from many different viewpoints, e.g., from the viewpoint of the user, from the viewpoint of the functionality, from the viewpoint of the hardware, etc. Trying to deal with all the viewpoints at the same time is very difficult and will usually result in badly structured designs. A better approach is to identify the different viewpoints needed and model the required capabilities of the software with respect to one viewpoint at the time. Therefore, we have introduced the concept of *perspective*. We allow modeling the variability from a (variable and extensible) set of perspectives. Each perspective describes the software's variability from a certain point of view, and together they describe the variability of the required software. Note that the set of perspectives to be considered is variable. This means that the modeler can decide which perspectives are useful for the system and which not. He even can stick to one single perspective (e.g., the system perspective) and if he thinks a certain perspective(s) is missing, it can be added (extensibility).

For all perspectives the same modeling technique is used. This modeling technique is a revised version of the traditional feature modeling techniques. It was necessary to introduce such a revised technique to overcome the limitations of feature modeling mentioned earlier. It is based on a few simple modeling concepts that allow modeling features, variability relations and feature dependencies. For more details and evaluation of the approach we refer the reader to [15].

4.1 Multi-Perspective Approach

A perspective is used to model the variability of the software from a certain point of view. The perspectives used for the modeling can be freely chosen depending on the application under consideration. To help the analysis, a set of possible perspectives have been identified. Possible perspectives include: *System* perspective, *Users* perspective, *Functional* perspective, *Non-functional* perspective, *User Interface* perspective, and *Localization* perspective. As already mentioned, it is not required to consider all these perspectives. For instance, the *Localization* perspective is only useful for software that needs to be localized for different markets. This set of perspectives can be further extended based on the needs of the application under consideration. For example, a *Hardware* perspective may be considered for embedded applications; or a *Task* perspective could be used for modeling task-based applications.

The exact definition of the concept of feature depends on the perspective taken. In general, a feature can be considered as *a physical or logical unit that acts as a building block for meeting the specifications of the perspective it belongs to*. A feature belonging to one perspective may relate to other features (*via dependencies*), also to features in other perspectives.

4.2 Modeling Primitives

The concept of *Feature* is the basic building block in our feature assembly modeling technique. We distinguish two types of features: *Concrete Feature* and *Abstract Feature*. The first type, *Concrete Feature* represents a unit of system capability. A concrete feature may be further decomposed to finer grained features to increase the level of detail. The second type, *Abstract Feature* is a source of variability; it represents a generalization of one or more specific features, called *Option Features*. An Option Feature represents an actual specification of its abstract feature. Concrete option features may also be decomposed to finer grained features. To illustrate the difference between the Abstract features and Concrete features, consider a Quiz Product Line application (see also section 5). In this application, *Operation Mode* is an abstract feature, while *Quiz* and *Exam* are examples of concrete operation modes and are therefore concrete features. Note that they are option features for the abstract feature *Operation Mode*.

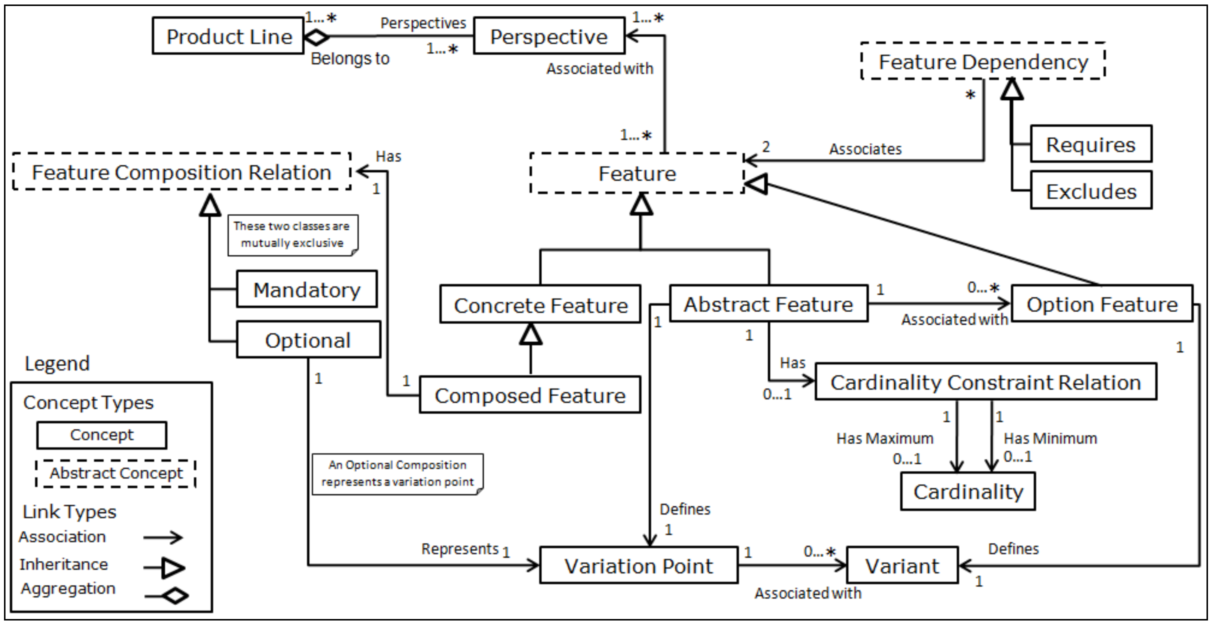

Figure 3. UML Meta Model for Feature Assembly Modeling Primitives

Within a feature assembly model, features are assembled using *feature relations*. Two types of feature relations are defined: *Composition Relation* and *Specification Relation*. A *Feature Composition Relation* represents a whole-part relation and is either *Mandatory* or *Optional*. A Mandatory Feature Composition defines a compulsory whole-part relation, while an Optional Feature Composition defines an elective whole-part relation. A Specification Relation can only be used for an Abstract Feature and allows associating the possible (concrete) Option Features with this Abstract Feature. In terms of variability, an Abstract Feature represents a *variation point* (i.e. a point at which several possibilities exist [16]). The Option Features associated with the Abstract Feature represent the *variants* (i.e. the specific possibilities of a certain feature [16]). The number of Option Features (variants) allowed to be selected in a certain product is expressed via a *Cardinality Constraint Relation*. The Cardinality Constraint Relation specifies the minimum and maximum number of features allowed to be selected. A dash (" -") is used to specify “any”.

In addition, features may be associated with *Feature Dependencies* [3], which specify how the selection of one feature may affect the selection of other feature(s). These dependencies specify how features interact with one another in a single perspective and between different perspectives. For example, a *Requires* dependency between two features belonging to different perspectives shows how they collaborate together to provide a specific required capability, while an *Excludes* dependency expresses incompatibility between features. We actually use the regular feature model dependencies. So, we will not elaborate on them.

Figure 3 shows the Meta model (UML class diagram) for the Feature Assembly Modeling technique.

All different types of features are stored in the feature pool. Feature Compositional Relations between features are also stored in the feature pool, yet they are not enforced while reusing the features, i.e. features can be reused from the pool with or without their full set of descendants. Furthermore, a feature can be reused without its ancestors. Similarly, also Specification Relations between Abstract Features and Option Features are stored in the pool. When reused, an Option Feature cannot be used without its Abstract parent, while an arbitrary number of Option Features can be reused for a single Abstract Feature. Also Feature Dependencies are stored in the feature pool, but they are not enforced when reusing features unless selected. The reason for storing relations and dependencies with features but not enforcing them when reusing the features is that it depends on the context of use where or not the relations and/or dependencies hold.

5. EXAMPLE

In this section we provide an example to illustrate the Feature Assembly and Feature Assembly Modeling technique. Consider a *Quiz Product Line application (QPL)* that is intended to meet many customers and markets. Therefore important perspectives for such as an application are: the System perspective that describes the fundamental features of the product line, the Users perspective that identifies possible target users, the Functional perspective that identifies and models the required functionalities, the User Interface perspective that identifies and models the elements of the required user interface (meeting all possible target users), and the Localization perspective that models different market requirements.

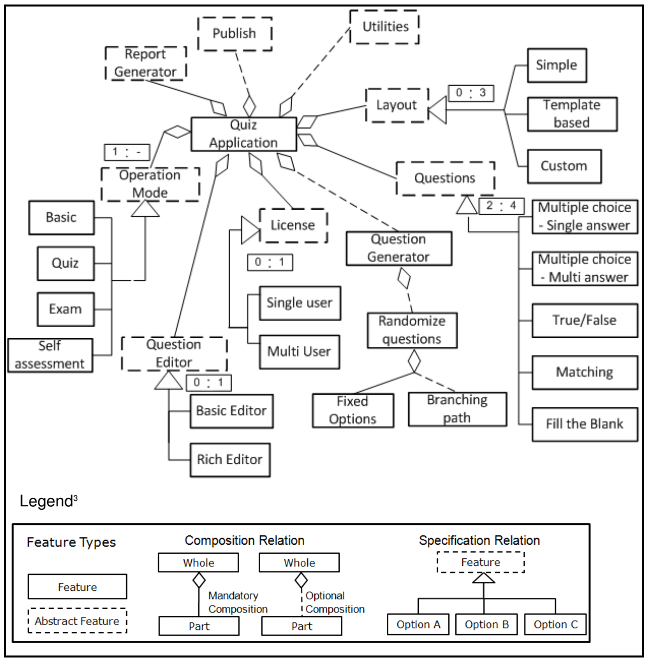

Figure 4. Quiz Product Line System Perspective

Figure 4 shows the Feature Assembly Model for the System perspective. As shown in figure 4, a Quiz application is mandatory composed of a set of features namely: *Questions*, *Layout*, *License*, *Report Generator*, *Operation Mode* and *Question Editor*. In addition, the following features are optional part of the quiz application: *Question Generator, Utilities*, and *Publish*. The *Questions* feature is an Abstract Feature (i.e. variation point), which has five concrete Option Features (i.e. variants). In any valid product at least two and at most four of these options should exist; this is specified by the cardinality *2:4*. On the other hand, the Abstract Feature *Operation Mode* has four Option Features; at least one has to be selected. No upper limit is defined, which means that maximum number of operation modes allowed in any valid application, is equal to the number of available Option Features. This is indicated by the dash in the variability cardinality *1:-*. The *Question Generator* feature is further decomposed into a *Randomize Questions* feature (responsible for randomizing the questions). The feature *Randomize Questions* on its turn is decomposed into a *Fixed Options* feature (which represents a normal random number generator) and an optional *Branching Path* feature (which allows creating paths for selecting the next question to display). Figure 4 also shows some other features that are part of the quiz application (*Utilities* and *Publish)*, but details about these features are not further specified (yet). This is an important aspect of the Feature Assembly approach; it allows identifying Abstract Features (variation points) while the concrete Option Features

[3] The same legend is used for all subsequent figures.

(variants) may not yet be known. This allows adopting an incremental design approach. When the concrete Option Features become available, they can be added to the model together with the associated Cardinality Constraints. Listing 1 shows the inter-perspective dependencies between the features of the quiz System perspective.

```
Matching *Excludes* Simple

Fill the Blank *Excludes* Simple

Self Assessment *Requires* Branching Path

Self Assessment *Requires* Report Generator

Exam *Requires* Report Generator
```

Listing 1. Sample Inter-Perspective Dependencies for the System Perspective

Figure 5 shows the Feature Assembly Model for the Users perspective, where features represent user categories. Figure 6 gives the Graphical User Interface perspective (due to space limitation only a subset of the features is shown). Furthermore, the three different perspectives shown in figures 4, 5, and 6 hold intra-perspective dependencies that specify how different features in different perspectives relate. Listing 2 shows a sample of the intra-perspective dependencies for the perspectives given for the Quiz application.

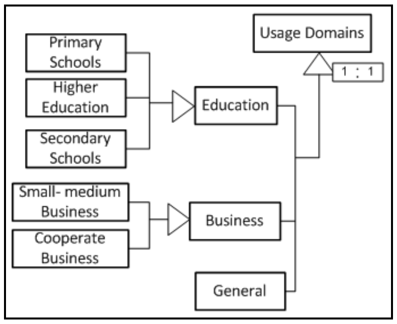

Figure 5. QPL Users Perspective

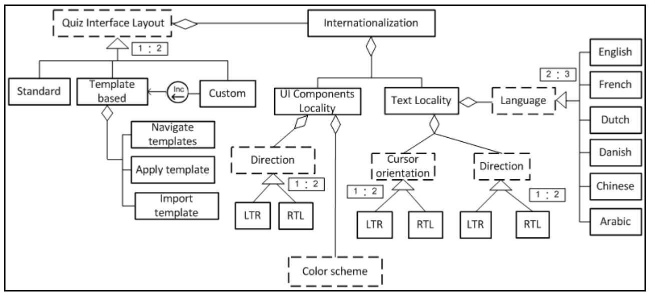

Figure 6. QPL Graphical User Interface Perspective

```
(Users.Higher_Education AND
User_Interface.Template_Based)   requires
System.Quiz_Question_Generator

(User_Interface.Dutch AND
User.Cooperate_Bussiness) Excludes
System.Standard

(User_Interface.Dutch OR User_Interface.French)
requires User_Interface.English

Users.Cooperate_Bussiness requires
User_Interface.English
```

Listing 2. Sample Intra-Perspective Dependencies

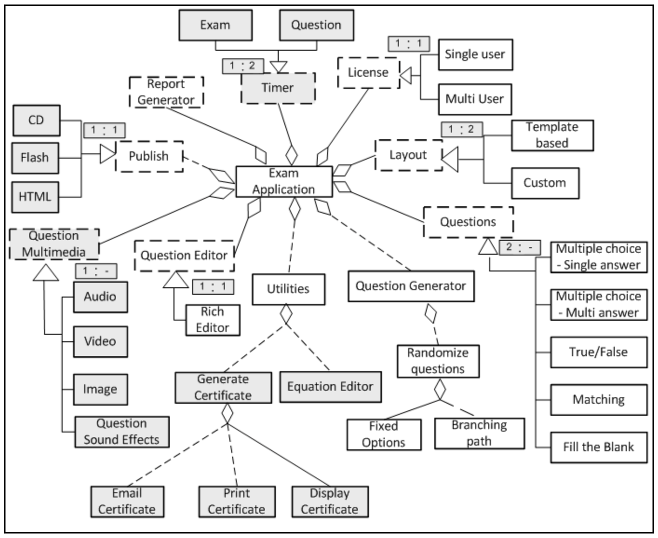

Figure 7. Exam Product Line Feature Assembly System Perspective

To illustrate how one can make use of *feature reuse* in Feature Assembly, consider the need for developing an *Exam* product line. Clearly, some features defined for the Quiz product line are also applicable in the Exam product line. Using our Feature Assembly approach, the reusable features are looked-up and extracted from the feature pool. Reusable features include: *Report Generator*, *License*, *Questions*, *Question Generator,* and *Layout.* Abstract Features such as *Layout* are extracted from the pool in addition to (some of) their associated Option Features, in this case, *Template Based and Custom*. Similarly, Concrete Features are extracted, because their decomposition (defined for the Quiz product line) is also applicable for the Exam product line, their descendents are also extracted. Figure 7 shows the feature assembly model for the System perspective of the Exam product line application. Note how some features (e.g., *Questions* and *Layout*) are reused but with different variability specifications (defined by the different cardinality*)*. When selecting features from the feature pool, the perspectives need to be respected, i.e. it is not allowed to reuse a feature in a perspective other than the one it was defined for. This is because the semantics of the concept *feature* is different in different perspectives. Therefore it is not allowed to mix features from different perspectives when reusing them. On the other hand a feature defined in more than one perspective can be reused in any of the perspectives it was defined in, as long as the perspective of reuse is the same as the perspective of definition. Actually a feature is identified by its name and its perspective.

In addition to the existing features, some new features were required for the Exam product line such as: *Timer*, *Generate Certificate*, *Question Multimedia* and *Equation Editor*. Moreover, features could be further elaborated when reused, such as the *Publish and Utilities* features, which are Abstract Features and had no Option Features associated with them yet in the Quiz product line. For example, the *Publish* feature has three Option Features associated with it in the Exam product line, namely: *CD*, *Flash* and *HTML*. A Feature Assembly model is created from the new features and the existing ones as shown in figure 7 (new features are shown with a grey shade).

6. RELATED WORK

Feature Assembly is based on the idea of assembling products from a set of existing features as well as new features. This idea has been applied successfully in manufacturing as a way to achieve mass production, such as cars, engines and many electronic appliances. In Feature Assembly we use the idea of assembling to compose feature models from a set of existing features. The principle of assembling a certain product from preexisting artifacts (components) has previously been proposed in software engineering. For example, in [17] the statement "One of the essential characteristics of engineering disciplines is to build a product by assembling pre-made, standard components" was made. Component Based Development (CBD) [17] [18] is based on developing software by composing pre existing components. Furthermore, there is a separation between the development of the components and the development of the software that will utilize these components [19]. This has called for creating self contained components that would then minimize the writing of code to only gluing code (code that glues the components together). Although the idea of CBD did not achieve its merits in software development in general, it has been a great success in some specific domains. For example web services based applications [20] [21] and e-learning applications [22] are often built using a CBD approach. For example, in web services, applications are assembled from a set of appropriate web services according to the functionality they provide. Web service discovery and identification plays an important role in the success of the web service composition approaches [20]. Web services are annotated with their usage, this description is then stored in a central web service registry. To find a certain web service, the registry is inspected [21].

The idea of using composition of components for achieving software reusability with variability was first introduced in [8], as a result of the widening of the product line scope due to new emerging requirements. The idea proposed in that paper was to create software based on composing existing and new components. Frameworks were introduced as a possible architectural support.

The need for applying reusability in combination with the product line technique was advised in [12]. The authors advice reusing previously made (variable) components in new software applications for the sake of rapid development in large variable software applications. They argue that the combination of reusability with variability will speed up the development process. However they do not give details about how such reusability can be achieved.

In [23] the idea of *product populations* was introduced to represent a portfolio or set of product lines, in which products of a single product line have many commonalities, and in addition many commonalities exist between the different sets of product lines [23]. The author points out that to achieve productibility in such a setting there is a need for reuse between the different product lines. In addition, their methodology relies on *composition* rather than *decomposition* of components. This is the same principle as we propose with feature assembly but rather on a feature level.

The above-mentioned work in CBD applied to product lines, tried to solve the problem of maintaining productiblity via reuse in addition to variability. They are situated on the component level rather than the on the modeling level. This makes them more code oriented. Furthermore, taking reusability into account at an early design stage is complementary to reuse at a component level and could enhance the reusability of the components. In [24] the authors promote reuse in product line development as means for rapid development. They argue that care must be taken to balance between reuse and product differentiation. They present an approach for balancing needs for differentiation and reuse in complex product lines based on industrial cases.

On the level of modeling variability, many works have extended the original FODA, for example FORM [25], FeatureRSEB [26], PLUSS [27], and CBFM [28] in order to introduce more modeling power and overcome some of its limitations, but none of them have explored the idea of feature assembly.

Software perspectives or viewpoints was first introduced to software development in [29] to show how adopting perspectives helps in efficient modeling of the software system. In [30] [31] [32] abstraction via viewpoints was introduced for software architecture modeling. In [33] a multi perspective approach for modeling variability was proposed, in which perspectives were defined based on stakeholders. Each stakeholder has his/her own perspective in defining variability. Therefore, stakeholders are able to maintain their own partial models about the domain and its variability. In [34], a different approach for separation of concerns was adopted, the approach depends on defining model fragments. A model fragment is a partial model with defined dependencies to other model fragments. The fragments need to be merged to have a global overview of the complete model, while doing so the consistency of the overall model is checked.

In addition, in the domain of feature modeling, some works have been proposed to support change (the type of change at the design stage is referred to as *offline* change [35]) in feature models. The need for refactoring emerges from the need for change in feature models, either to widen the scope of an existing product lines or to support product evolution (usually driven by customer needs) or to recover from existing errors. In [36], the authors developed a versioning system for feature models in which each feature is associated with two versions: logical version and container version. They mark the change in the features logical functionality and in the overall feature model functionality respectively. In [37], the authors define a list of possible patterns for refactoring feature models. The patterns identify the most common changes that could be required in a certain feature model and provide how to modify the feature model to adapt to that change. Although somehow related, the purpose of that kind of work is on capturing the evolution of a single model.

In [38] a decision oriented approach for modeling variability called DOPLER (Decision-Oriented Product Line Engineering for effective Reuse) was presented. DOPLER provides a generic meta model composed of Assets and Decisions for modeling variability. This meta model can then be extended to each specific domain to create domain specific meta models for variability modeling. Decisions represent a problem space view on the product line's variability, while assets represent an abstract view of the solution space in the degree of detail needed for subsequent product derivation, both are represented from the perspective of users. Assets are linked to decisions via inclusion conditions, which are rules that define which assets will be added during product derivation. The approach supports evolution by allowing propagation of the meta model changes to already existing variability models.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a new framework for modeling and reusing variability called Feature Assembly. Feature Assembly Framework uses the idea of assembling features in order to model variable software. Furthermore, it promotes feature reusability by storing the features in a so-called feature pool, which acts as a feature repository. Whenever a new feature is needed, it is also added to the pool of features therefore allowing the pool to continuously grow.

We believe that Feature Assembly is an important step towards better reusability in variable software. Although composing software from components has been around for over a decade now, it is by experience quite difficult to be fully achieved. This is mostly due to the fact that components are not usually built with reusability in mind. Therefore, in this paper we introduce composition and reusability as early as possible, i.e. during domain analysis and design. In this way, it promotes to take reusability into account early in the development cycle. Moreover, we also make use of reusability and composition during this phase as well. So it is design for reuse as well as design by reuse.

The concept of creating feature models by assembling features is made feasible via the Feature Assembly Modeling technique. We have briefly introduced the Feature Assembly Modeling technique which is a multi-perspective approach for realizing separation of concerns in order to deal with the variability modeling of large and complex software. Adopting a perspective-based approach for defining features helps abstracting from issues that are not relevant for a particular aspect or viewpoint. By expressing dependencies between features of different perspectives, the different perspectives are connected. This will also be useful when making configuration to eliminate invalid feature combinations. Through its revised feature modeling technique, it enables reusability of features. The specification of the information about the variability is separated from the definition of the features.

The next step in the research is to provide tool support for the Feature Assembly approach. This includes the realization of the feature pool and providing a powerful search mechanism for looking for suitable features. Scalability of the approach should be further investigated. Although current data stores hold billions of records with no problems further investigation on the query response time for the feature pool search should be investigated in large feature pools. Also an industrial evaluation of the approach is planned. Furthermore, linking features to implementation components is desirable to ensure traceability and to also realize reusability at the architecture level.

8. ACKNOWLEDGEMENT

This research is sponsored by IRSIB (Institute for the encouragement of Scientific Research and Innovation of Brussels) through the VariBru project (www.varibru.be). The authors would like to thank Wim Codenie, Nicolás González-Deleito, and Tom Tourwé from Sirris for their valuable discussions regarding the needs of the industry, which highly contributed to the elaboration of the ideas.

9. REFERENCES

[1] J. Bosch, "Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach", Addison-Wesley, Boston (MA), 2000

[2] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, "COVAMOF: A Framework for Modeling Variability in Software Product Families", In: Proceedings of the Third Software Product Line Conference (SPLC 2004), Springer Verlag Lecture Notes on Computer Science, vol. 3154 (LNCS 3154), , pp. 197–213, 2004

[3] K.C. Kang, J. Lee, P. Donohoe, "Feature-Oriented Product Line Engineering", IEEE Software, vol. 19, no. 4, pp. 58-65, 2002

[4] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study", Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, 1990

[5] M. Acher, P. Collet, P. Lahire, , R. France, "Composing Feature Models", In: 2nd International Conference on Software Language Engineering (SLE'09), 2009

[6] S. A. Ajila, A. B. Kaba, "Evolution support mechanisms for software product line process", In: J. Syst. Softw., vol. 81, no. 10, pp. 1784-1801, 2008

[7] J. Bosch, "Expanding the Scope of Software Product Families: Problems and Alternative Approaches", In: Proceedings of the 2nd International Conference on the Quality of Software Architectures (QoSA 2006), 2006

[8] R. Van Ommering, J. Bosch, "Widening the scope of software product lines — from variation to composition", pp. 31-52, 2002

[9] T. von der Massen, H. Lichter, "Deficiencies in feature models", In: T. Mannisto, J. Bosch (Eds.), Workshop on Software Variability Management for Product Derivation - Towards Tool Support , 2004

[10] M. Riebisch, "Towards a more precise definition of feature models", In: Modelling Variability for Object-Oriented Product Lines. pp. 64-76, 2003

[11] M. Pizka, A. Bauer, "A brief top-down and bottom-up philosophy on software evolution", In: Proc. of the Int. Workshop on Principles of Software Evolution (IWPSE), 2004

[12] J. Estublier, G. Vega, “Reuse and variability in large software applications”, In: ESEC/SIGSOFT FSE 2005, pp. 316-325, 2005

[13] J. Bosch Software Product Families in Nokia. In: 9th International Conference SPLC2005 (2005).

[14] A. Maccari, A. Heie, Managing infinite variability in mobile terminal software. Softw., Pract. Exper, 35(6), pp. 513-537 , 2005

[15] L. Abo Zaid, F. Kleinermann, O. De Troyer, “Feature Assembly: A New Feature Modeling Technique”, 29th International Conference on Conceptual Modeling, Lecture Notes in Computer Science, Vol. 6412/2010, pp. 233-246, 2010

[16] Svahnberg, M., van Gurp, J., Bosch,J., “A taxonomy of variability realization techniques”, In: Software Practice & Experience, vol 35(8), pp. 1-50, 2005

[17] J. A. Wang, “Towards component-based software engineering”, In: Proceedings of the seventh annual CCSC Midwestern conference on Small colleges. Consortium for Computing Sciences in Colleges, pp. 177-189, 2000

[18] G.T. Heineman, G.T. Councill, “Component-Based Software Engineering: Putting the Pieces Together”, Addison-Wesley Professional, ISBN 0-201-70485-4, 2001

[19] I. Crnkovic, M. Chaudron, S. Larsson, “Component-Based Development Process and Component Lifecycle”, International Conference on Software Engineering Advances (ICSEA'06), pp.44, 2006

[20] B. Srivastava, J. Koehler, “Web service composition - current solutions and open problems”, In: Proceedings of ICAPS 2003, 2003

[21] S. Dustdar, , W. Schreiner, “A survey on web services composition, International Journal of Web and Grid Services”, Vol. 1, No. 1, pp. 1-30, 2005.

[22] V. H. Menéndez, , M.E. Prieto, “A Learning Object Composition Model”, UNISCON 2008, 469-474, 2008

[23] van Ommering, R.: Software reuse in product populations. Software Engineering, IEEE Transactions on 31 (7), 537-550, 2005

[24] J. Savolainen, J. Kuusela, M. Mannion, T. Vehkomäki: “Combining Different Product Line Models to Balance Needs of Product Differentiation and Reuse”. ICSR 2008: 116-129, 2008

[25] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh: “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures”, In: J. Annals of Software Engineering. vol. 5, pp. 143-168, 1998

[26] M. Griss, J. Favaro, M. d’Alessandro, “Integrating Feature Modeling with the RSEB”, In: Fifth International Conference on Software Reuse, pages 76–85, 1998

[27] M. Eriksson, J. Börstler, K. Borg: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In: Obbink and Pohl (eds), pp. 33- 44, 2005

[28] K. Czarnecki, C. H. P. Kim, “Cardinality-Based Feature Modeling and Constraints: A Progress Report”, In: OOPSLA’05 International Workshop on Software Factories , 2005

[29] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke, “Viewpoints: A Framework for Integrating Multiple Perspectives in System Development”, Intl. J. of Software Engineering and Knowledge Engineering 2(1), pp. 31–57, 1992

[30] T.C. N. Graham, “Viewpoints Supporting the Development of Interactive Software”, In Proceedings of Viewpoints 96: International Workshop on Multiple Perspectives in Software Development, ACM Press, San Francisco, USA, pp. 263-267, 1996

[31] E. Woods, “Experiences Using Viewpoints for Information Systems Architecture: An Industrial Experience Report”, EWSA 2004: pp. 182-193, 2004

[32] B. Nuseibeh, J. Kramer, A. Finkelstein, “ViewPoints: Meaningful Relationships Are Difficult!“, In: Proceedings of International Conference on Software Engineering (ICSE'03), 2003

[33] M. Mannion, J. Savolainen, T. Asikainen, “Viewpoint-Oriented Variability Modeling”, In: International Computer Software and Applications Conference (COMPSAC’09), pp. 67–72, 2009

[34] D. Dhungana, P. Grünbacher, R. Rabiser, T. Neumayer, “Structuring the modeling space and supporting evolution in software product line engineering”, Journal of Systems and Software 83(7), pp.1108-1122 , 2010

[35] S. A. Ajila, A. B. Kaba: “Evolution support mechanisms for software product line process”, In: J. Syst. Softw., vol. 81, no. 10, pp. 1784-1801, 2008

[36] R. Mitschke, M. Eichberg, “Supporting the Evolution of Software Product Lines”, In: ECMDA Traceability Workshop, 2008

[37] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. J. P. de Lucena, “Refactoring product lines”, GPCE 2006: 201-210

[38] D. Dhungana, P. Grünbacher, R. Rabiser, “The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study”, Journal of Automated Software Engineering, DOI: 10.1007/s10515-010-0076-6 (in press)