

# Feature Assembly: A New Feature Modeling Technique

Lamia Abo Zaid<sup>1</sup>, Frederic Kleiner mann<sup>1</sup>, and Olga De Troyer<sup>1</sup>

<sup>1</sup> Vrije Universiteit Brussel (VUB)  
Pleinlaan 2, 1050 Brussel  
Belgium

{Lamia.Abo.Zaid, Frederic.Kleiner mann, Olga.DeTroyer}@vub.ac.be, <http://wise.vub.ac.be/>

**Abstract.** In this paper we present a new feature modeling technique. This work was motivated by the fact that although for over two decades feature modeling techniques are used in software research for domain analysis and modeling of Software Product Lines, it has not found its way to the industry. Feature Assembly modeling overcomes some of the limitations of the current feature modeling techniques. We use a multi-perspective approach to deal with the complexity of large systems, we provide a simpler and easier to use modeling language, and last but not least we separated the variability specifications from the feature specifications which allow reusing features in different contexts.

**Keywords:** Feature, Variability Modeling, Feature Models, Domain Analysis

## 1. Introduction

Over the last decades software development has evolved into a complex task due to the large number of features available in software, and secondly due to the many (often implicit) dependencies between these features. In addition, there is an increased demand to deliver similar software on different platforms and/or to different types of customers. This has led to the emergence of so-called Software Product Lines (SPL) [1] or more generally *variable software*. SPLs tend to *manufacture* the software development process. Instead of developing a single product the fundamental base is to develop multiple closely related but different products. These different products share some common features but each individual product has a distinguishable set of features that gives each product a unique flavor. To be able to profit maximally from the benefits of variability, but to keep the development of such software under control, feature-oriented analysis is used to effectively identify and characterize the SPL capabilities and functionalities. In feature-oriented analysis, features are abstractions that different stakeholders can understand. Stakeholders usually speak of product characteristics i.e. in terms of the features the product has or delivers [2].

Feature oriented domain analysis (FODA) [3] was first introduced in the 1990 for domain modeling, and since then it has become an appealing technique for modeling SPLs. It was applied in several case studies [2] and many extensions to the original technique have been defined. However, these feature modeling techniques have not

gained much popularity outside the research community. Several explanations can be given for this. Firstly, there are many different “dialects” of feature modeling techniques (e.g. [4] [5] [6]), each focusing on different issues; there is no commonly accepted model [7]. Secondly, feature models do not scale well, mainly because they lack abstraction mechanisms. This makes them difficult to use in projects with a large number of features [8]. Thirdly, little guidelines or methods exist on how to use the modeling technique. This often results in feature models with little added value or of discussable quality.

To overcome these limitations companies define their own notations and techniques to represent and implement variability. Examples are Bosch [9], Philips Medical Systems [10] and Nokia [11]. Yet the proposed notations are tailored to each company’s specific needs for modeling variability in their product line. In [9] and [10] a hierarchical structure of features, introducing new feature types was adopted. While feature interaction and scalability issues were more important for [11], therefore they adopted a separation of concern approach for devising higher level features. They used documentation to specify the systems evolution using its features and relations.

In this paper we present a new feature modeling technique that is based on using multiple perspectives (viewpoints) to model (variable) software in terms of its composing features. We call it Feature Assembly Modeling (FAM). The presented modeling technique is innovative from different perspectives. It separates the information on variability (i.e. how features are used to come to variability) from the features it serves. In FAM, how a feature contributes to the variability of a specific piece of software (or product line) is not inextricably associated with the feature. Rather this information is part of how the features are assembled together in the feature assembly model that models the software (or product line). This yields more flexibility and allows the reuse of these features in other contexts and even in other software. The model is also based on a multi-perspective abstraction mechanism. It is well known that focusing on one aspect at the time helps to deal with complexity (also known as the separation of concerns paradigm). FAM provides better abstraction mechanism by using perspectives to model large and complex software; and thus will also increase scalability of the modeling process. Furthermore, we have reduced the number of modeling primitives to simplify and ease the modeling process.

This paper is organized as follows, in section 2, we review existing feature modeling techniques. In section 3, we discuss the limitations of the mainstream feature modeling techniques. In section 4, we explain our Feature Assembly Modeling technique. Section 5 provides an example that illustrates the approach and its benefits. Next, in section 6 we discuss how FAM offers solutions for the limitations identified in section 3. Finally, section 7 provides a conclusion and future work.

## **2. Mainstream Feature Modeling techniques**

Over the past few years, several variability modeling techniques have been developed that aim supporting variability representation and modeling. Some of the techniques extend feature models (e.g. [4], [5], [6], and [12]), while others tend to add profiles for variability representation in UML (e.g. [13], [14], and [15]). In addition,

some work has been done on defining new modeling languages and frameworks to model variability information (e.g. [16] and [17]). For the purpose of this paper we restrict ourselves to the modeling methods extending Feature Oriented Domain Analysis (FODA), commonly called feature models [3] [4]. For a detailed study classifying the existing well known feature modeling techniques, methodologies and implementation frameworks, we refer the reader to [18].

A feature model is a hierarchical domain model with a tree-like structure for modeling features and their relations. It is a variability modeling (visual) language indicating how the features contribute to variability. Over the past decade several extensions to FODA (the first feature modeling language) have been defined to compensate for some of its ambiguity and to introduce new concepts and semantics to extend FODA's expressiveness. Yet, all keep the hierarchical structure originally used in FODA, accompanied with using some different notations.

Feature-Oriented Reuse Method (FORM) [4] extends FODA by adding a domain architecture level which enables identifying reusable components. It starts with an analysis of commonality among applications in a particular domain in terms of four different categories (also called layers): capabilities, operating environments, domain technologies, and implementation [2]. AND/OR nodes are used to build a hierarchical tree structured feature model for the features belonging to each of the previously mentioned categories. The *excludes* and *requires* feature dependencies originally defined in FODA are still used; a new *implemented by* dependency was defined.

FeatureRSEB [5] aims at integrating feature modeling with the Reuse-Driven Software Engineering Business (RSEB). It uses UML use case diagrams as a starting point for defining features and their variability requirements. FeatureRSEB classifies features to *optional*, *mandatory* (similar to FODA) and *variant*. Variant is used to indicate alternative features. FeatureRSEB adds the concept of *vp-features* which represents variation points. The *excludes* and *requires* dependencies originally defined in FODA are used to represent constraints between features.

PLUSS [12], which is the Product Line Use case modeling for Systems and Software engineering, introduced the notation of *multiple adaptor* to overcome the limitation of not being able to specify the *at-least-one-out-of-many* relation in FODA. PLUSS also renamed alternative features to *single adaptor* features following the same naming scheme. The modeling notation was also slightly changed in PLUSS to meet the needs of the modified model, yet it remained a hierarchical tree structure based on the notation of FODA. Similar to FeatureRSEB, the *excludes* and *requires* dependencies originally defined in FODA are used to represent feature dependencies.

Cardinality Based Feature Models (CBFS) [6] represent a hierarchy of features, where each feature has a *feature cardinality*. Two types of cardinality are defined: *clone cardinality* and *group cardinality*. A feature clone cardinality is an interval of the form [m..n]. Where m and n are integers that denote how many clones of the feature (with its entire subtree) can be included in a specified configuration. A group cardinality is an interval of the form [m..n], where m and n are integers that denote how many features of the group are allowed to be selected in a certain configuration. Features still had one of four feature types AND, OR, Alternative, and Optional. In addition, the notation of *feature attribute* was defined. A feature attribute indicates a property or parameter of that feature; it could be a numeric or string value. CBFS kept

the original FODA feature dependencies. In addition, there are rational constraints associated with the value of the feature attribute (i.e.  $>$ ,  $<$ ,  $=$ ,  $>=$ ,  $<=$ ).

### 3. Limitations of Mainstream Feature Modeling techniques

Feature models relate features by means of a AND/OR hierarchical structure, describing how features are broken up into more finer-grained ones. For small applications this works fine, as features are perceived quite easily and often represent the main system capabilities and components. Yet for practical cases there is usually great doubt in how to apply the feature modeling technique. First, because there are many alternatives to the original FODA, which all differ in their semantics as well as their notations (in [19] a comparative survey on feature-based notations was done to help companies decide which technique better suits their needs). Next, these techniques are not associated with a concrete methodology or guidelines that designers can use in order to create their feature models. Usually, it already starts with the definition of the features. Very often, there are no guidelines or definitions that can be used to decide what to consider as a feature and what not. This makes the modeling process a difficult task. A recent study [20] reveals that there are very few reports on the application of feature models in practice. Out of the available literature of software variability only 16 cases were relevant. The study shows that only two of the 16 cases claim success in applying feature models.

In addition, FODA and subsequent FODA based feature modeling techniques lack explicit abstraction mechanisms. Usually, high level features are decomposed into lower level features in the feature model, but it is not defined to which level of granularity features should be defined. The original FODA defined four categories to which features of the system belong [2] [3]: operating environments, capabilities, domain technology, and implementation techniques. The Capabilities category is further categorized into functional features, operational features, and presentation features. However, we see this categorization process as very fragile and impractical (more details in section 6). In reality, a feature may have many faces which make categorizing features a difficult task.

Being originally defined for domain modeling, feature modeling techniques miss linking their notations of features with the notations of *variation point* and *variant* which is preferred among stakeholders interested merely in variability [1]. UML based variability modeling (e.g. [13], [14], and [15]) tried to address this issue. Yet UML variability modeling techniques speak the language of *class* rather than *feature*. This makes them more appropriate for architecture and/or class design rather than domain analysis.

As already mentioned, not only do feature modeling techniques lack an associated modeling method, also the main modeling concept, being *feature*, is not rigorously defined. There are many different “definitions” that exist for defining “feature”. Actually each technique is using its own definition. We list some of these definitions:

1. A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [3]

2. A feature is a logical unit of behavior specified by a set of functional and non-functional requirements [1]
3. A feature is an increment in program functionality [21]
4. A feature is a functional requirement; a reusable product line requirement or characteristic [1].

It can be seen from these different definitions that features can be considered from different perspectives. While the first definition takes the user's perspective for defining what a feature is, the second and fourth definitions take the requirements perspective for defining what a feature is, and the third takes the functional perspective for defining what a feature is. This observation has led us to base our feature assembly approach (which will be introduced in section 4) on multi perspectives as an abstraction mechanism.

The observation that feature modeling is not used by companies (probably due to the limitations of feature modeling techniques (see above)) but confronted with the many challenges related to variable software that companies face<sup>1</sup>, has triggered the need to revise feature modeling. The following requirements were formulated:

- 1) A rigorous methodology for feature modeling is needed.
- 2) Abstraction mechanisms to better deal with complex and large systems are necessary.
- 3) Separate the feature from how it contributes to variability; it must be possible to use the same feature in different variability specifications.

The next section will explain our feature assembly modeling technique. Note that this technique is part of an overall Feature Assembly approach, which supports the reuse of features between different software.

## 4. Feature Assembly Modeling Technique

Feature Assembly Modeling (FAM) is a feature-oriented modeling technique intended to model the variability aspects of complex variable software. It does so by using different perspectives. Often software can be considered from many different viewpoints, called *perspectives*. Trying to deal with all the viewpoints at the same point is very difficult and will usually result in badly structured designs. A more scalable approach is to identify the different perspectives needed and model the required capabilities of the software with respect to one perspective at the time. Not only do perspectives help separation of concerns, but also provide an abstraction mechanism which allows focusing on only related features. Based on this, our feature assembly modeling technique allows specifying software based on a set of perspectives. Each perspective describes the variability from a certain point of view (e.g., the Users perspective, the Functional perspective), and together they describe the variability of the required software. Furthermore, within a single perspective, we represent how features are composed and related (assembled). The introduced model is based on a few simple modeling concepts that allow modeling features, variability

---

<sup>1</sup> This research is carried out in the context of a research project VariBru (<http://www.varibru.be>) in which the needs and challenges regarding variability of industrial companies in the Brussels Region are investigated.

relations, and feature dependencies. Next, we will discuss the approach in more detail.

#### 4.1. Multi-Perspective Approach

A perspective is used to model the variability of the software from a certain viewpoint. 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. The above-defined set of perspectives can be further extended based on the needs of the application under consideration. For example, a *Task* perspective could be used for modeling task-based applications or a *Hardware* perspective may be considered for embedded 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*).

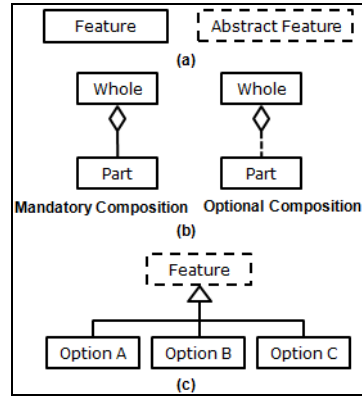
Note that the idea of using software perspectives or viewpoints is not new in software development; it was first introduced in [22] to show how adopting perspectives helps in efficient modeling of the software system. In [23] [24] [25] abstraction via viewpoints was introduced for software architecture modeling.

#### 4.2. Basic Modeling Primitives

To model the features of one perspective, we have revised the existing feature modeling techniques and came up with a new and simplified technique. In mainstream feature modeling, the *feature type* is used to express how a feature contributes to the variability. However, because a feature can contribute differently to variability in different situations, we have removed how the feature contributes to variability from its definition. Therefore, we only consider two types of features: *Feature* and *Abstract Feature*. A *Feature* represents a *concrete* logical or physical unit or characteristic of the system. A feature is represented by a solid line rectangle holding the feature's name. An *Abstract Feature* is a feature which is not concrete; rather it is a generalization of more specific features (concrete or abstract ones). An abstract feature is represented by a dotted line rectangle holding the abstract feature's name. Figure 1.a shows the two feature notations. To illustrate the difference between the two types of features, consider a Quiz Product Line application (see also section 5), *Operation Mode* is an abstract feature, while *Quiz* and *Exam* are examples of concrete features. In addition, *Operation Mode* is the generalization of the concrete features *Quiz* and *Exam*.

How the features are assembled together to model the system is specified via *feature relations*. We have defined two types of feature relations: *composition relation* and *generalization/specification relation*. The *composition relation* is used to

express the whole-part relation; i.e. a feature is composed of one or more fine-grained features. The composition can be *mandatory* or *optional*. A mandatory composition indicates a *compulsory whole-part* relation. An optional composition indicates an *elective whole-part* relation. Figure 1.b shows the notations used to represent the composition relation. The *generalization/specification relation* is used in combination with an abstract feature and allows specifying possible (concrete or abstract) *Option Features* of this abstract feature. Figure 1.c shows the notations used to represent the generalization/specification. Only abstract features are allowed to have generalization/specification relations. In terms of variability, an abstract feature represents a variation point. Its available option features represent variants. The number of option features allowed to be selected in a certain product is expressed via a *cardinality constraint*. The *cardinality constraint* specifies the minimum and maximum number of features allowed to be selected. A dash is used to specify “any”.



**Fig. 1.** Feature Assembly Model Notation (a) Feature types, (b) Composition relation, (c) Generalization/Specification relation.

### 4.3. Feature Dependencies

*Feature Dependencies* allow expressing dependencies between features. A Feature Dependency specifies how a feature may affect other feature(s). Dependencies can be expressed between features from a single perspective as well as between features from different perspectives. We will explain below the types of dependencies supported.

#### a) Feature dependencies within the same perspective (inter-perspective dependencies):

In our previous work we defined a set of (binary) feature dependencies [26]. The same set still holds for defining feature dependencies within the same perspective and corresponds with the dependencies usually considered in feature modeling (i.e. *requires* and *excludes*). Figure 2 shows the graphical representation and the associated semantics of the feature dependencies supported by the feature assembly model. It should be noted that some of these dependencies are symmetric (such as: *excludes*, *incompatible*, *same*) while others are asymmetric (such as: *extends*, *impacts*, *includes*, *requires*, *uses*), thus a direction (i.e. arrow) is associated with these dependency relations (Section 5 contains an example demonstrating their use).

#### b) Feature dependencies between different perspectives (intra-perspective dependencies)

It is often the case that two or more features constrain a feature belonging to a different perspective. Furthermore, a dependency may hold between features all belonging to different perspectives. Dependencies among features of different perspectives we call *intra-perspective dependencies*, the same dependencies shown in

figure 2 are valid, but now features from different perspectives can be combined with AND and OR. The form is:  $\langle virtual\_feature \rangle \langle dependency \rangle \langle virtual\_feature \rangle$ , where  $\langle virtual\_feature \rangle$  is one or more features connected with AND/OR, and  $\langle dependency \rangle$  is one of the keywords: excludes, incompatible, same, extends, impacts, includes, requires, uses. Here a feature must be identified by both the name of its perspective and its feature name. An example intra perspective dependency representing interdependencies in an e-shop application is: *user\_interface.checkout* AND *user\_interface.credit\_card* AND *users.customer* requires *user\_interface.discount*, which states that if the user interface contains a checkout feature and a credit card payment feature and there is a user category called customer then this requires that there is a discount feature in the user interface. Similarly *user\_interface.discount* uses *functional.discount\_rate*, states an operational dependency between the user-interface perspective and the functional perspective.









Dependency	Notation	Description
Extends		Feature A extends feature B if A adds to the functionality of B
Includes		Indicates that feature A has feature B inside of it.
Impacts		If feature A has an impact on feature B, it means that the existence of A affects the existence of B. This is typically used as a less rigid relation than the <i>requires</i> relation.
Incompatible		If feature A is incompatible with feature B, then A and B are mutual exclusive due to some conflict. From a configuration point of view, it is the same as the <i>excludes</i> constraint.
Same		Constraint used to indicate that two features are equivalent
Requires		Feature A requires feature B if A is functionally dependent on B.
Uses		Feature A uses feature B then there is a uses dependency relation
Excludes		Feature A excludes feature B indicates that A and B cannot occur together (exclusive OR)

Fig. 2. Feature Assembly Technique Feature Dependency Notations

## 5. Example

In this section we provide an example to demonstrate the FAM technique. Figure 3 shows the System perspective of a Quiz Product Line (QPL) application for making Quizzes, designed to meet the needs of multiple customers and markets. The QPL 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: *Quiz Question Generator*, *Quiz 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; as specified by the cardinality 2:4. Moreover,



the abstract feature *Operation Mode* has four option features; at least one has to be selected, the number of available option features is the upper limit as indicated by the cardinality 1:-. The *Quiz Question Generator* feature is further composed of a *Randomize Questions* feature that is responsible for making the questions random. The feature *Randomize Questions* is composed of a *Fixed Options* feature and an optional *Branching Path* feature. Figure 3 also shows some features part of the quiz application (*Quiz Utilities* and *Publish*) for which no details are specified (yet). This is an important aspect of FAM; it allows identifying abstract features or variation points while the concrete options (or variants) may not yet be known. This allows adopting an incremental design approach. When the concrete options are known, then they can be added to the model along with the associated cardinality constraints.

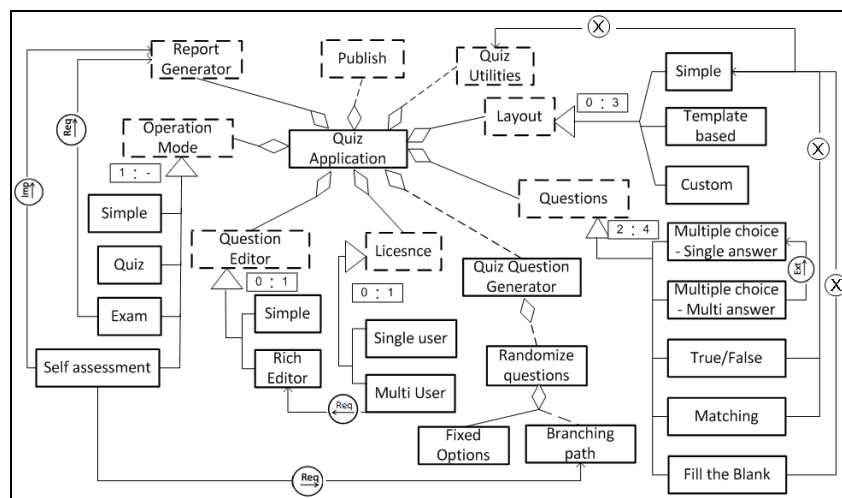


Fig. 3. Quiz product line system perspective.

Figure 3 also shows the inter-perspective dependencies, for example there is a *requires* dependency between *Exam* and *Report Generator*. Figure 4 shows features of the Users perspective and their dependencies. Figure 5 gives the User Interface perspective, showing the features that make up the user interface and their dependencies (due to space limitation only a subset of the features is shown). Furthermore, the three different perspectives shown in figures 3, 4, and 5 hold intra-perspective dependencies that specify how the different features relate. Listing 1 shows a sample of the intra-perspective dependencies for the perspectives given for the Quiz application.

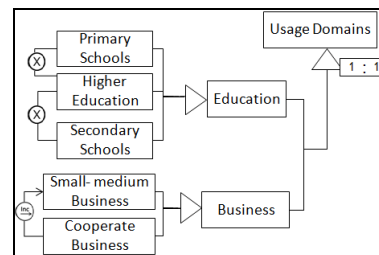


Fig. 4. QPL Users Perspective

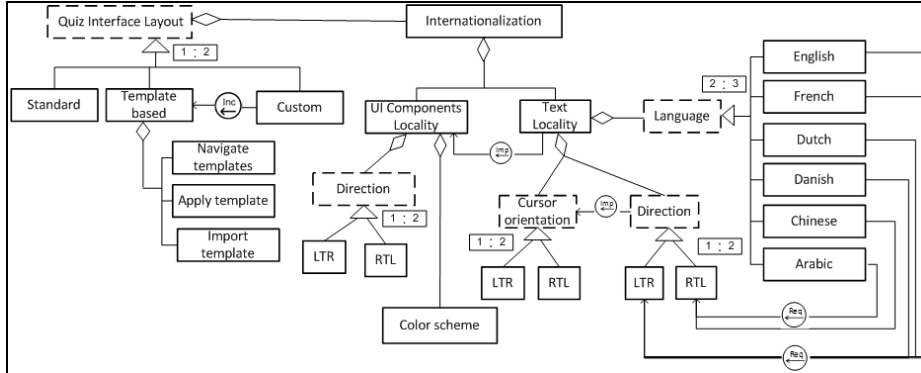


Fig. 5. Quiz product line user interface perspective.

```

Users.Higher_Education AND User_Interface.Template_Based requires
System.Publish
User_Interface.Dutch AND User.Cooperate_Bussiness requires System.Custom
(User_Interface.Dutch OR User_Interface.French) AND
Users.Cooperate_Bussiness requires User_Interface.English
Users.Cooperate_Bussiness requires (System.Custom AND User_Interface.English)

```

Listing 1. Sample Intra-perspective dependencies.

## 6. Discussion

In this section we demonstrate how FAM has solved some of the limitations of the mainstream feature modeling techniques (mentioned in section 3).

### a) Ambiguity in modeling concepts

Traditional feature models do not make an explicit distinction between a composition and a specialization. This may introduce problems, e.g. figure 6 shows the GPL problem introduced in [27], where two sets of alternative features are identified for the feature *Graph Type*, being *Directed/Undirected* and *Weighted/Unweighted*. This introduces two problems: Firstly, the model holds implicit information (by not naming the two concepts for which the two sets provide alternatives) leaving it to the intuition of the user to understand there are two concepts that makeup *graph type* (i.e. *direction* and *weight*). Secondly, *Graph Type* is a mandatory feature, while its succors are alternative features. Therefore, it is not clear whether at least one feature of one alternative group should be selected, or one feature of each group should be selected. Figure 7 shows another example of ambiguity by combining different types of variability: F is optionally composed of F1, and at the same time F1, F2, and F3 are alternative descendant features of F. Although this ambiguity can be resolved by normalizing the feature model [28] (i.e. allowing each feature to have only one type), the modeling method does not prevent such situations.

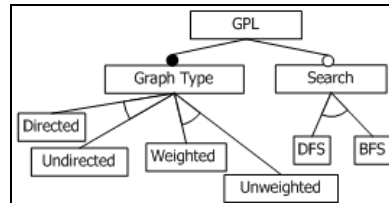
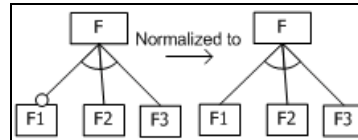


Fig. 6. Feature Model of GPL

The above-mentioned ambiguities are mainly due to the fact that FODA mixes the variability information of a feature with its composition information. This problem is solved in FAM by introducing abstract features that are intended for representing variation points (variability information), and by explicitly distinguishing between composition relations and specialization relations (where the last type can only be used for abstract concepts and thus for specifying variability information).



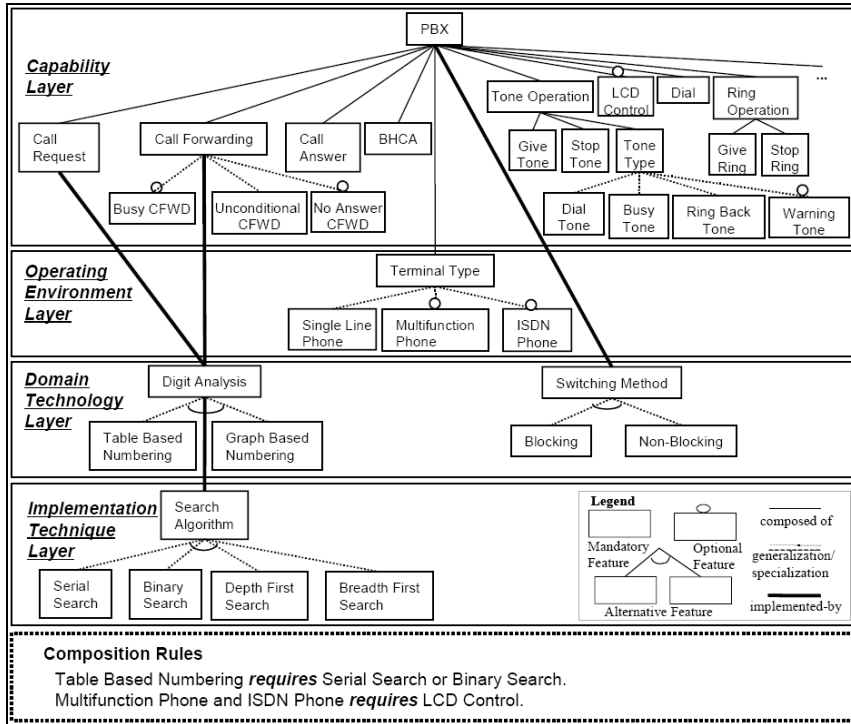
**Fig. 7.** Example showing the need for normalizing FMs, after [28]

### b) Missing Reuse Opportunities

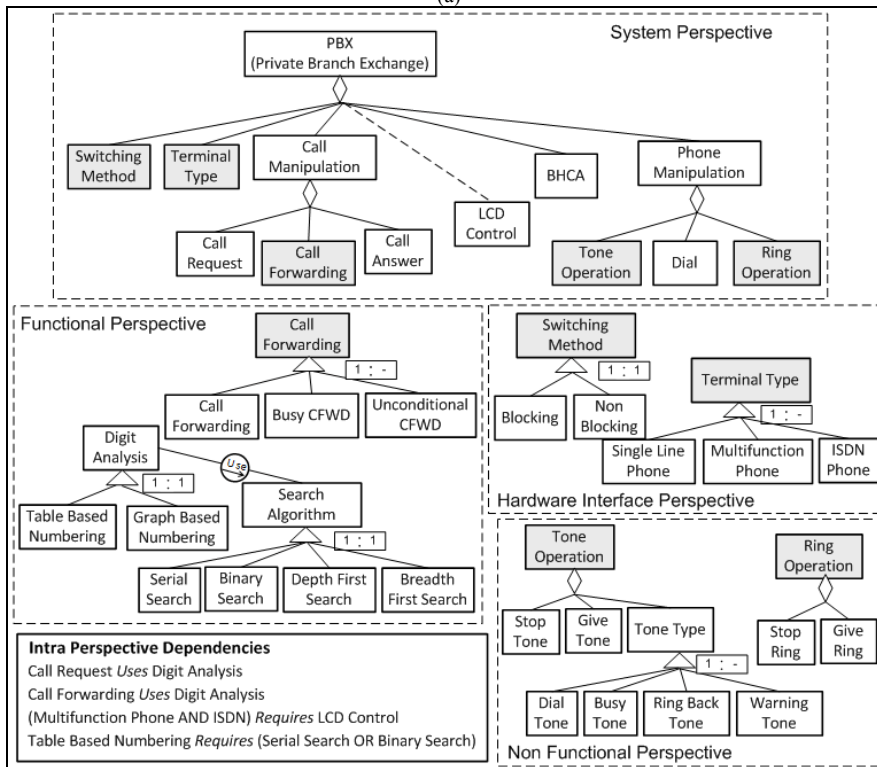
In current feature models, a feature is given a type that indicates how the feature contributes to the variability of the system. This limits the possibilities to reuse a feature in a different context. For example, a *bank transfer payment* feature may be mandatory in one setting while optional in another. As the type (e.g. mandatory) is inextricably associated with the feature, it is not possible to reuse the feature as it is. In addition, it is quite difficult to add new features or change (the variability type of) an existing feature. For example, a *Language* feature may have initially two alternative features *English* and *French*. When targeting new markets, this feature may need to be extended with other languages (e.g., *Dutch*, *Spanish*, and *German*). Furthermore, suppose that the *English* feature needs to become mandatory, while there is a need to select one or more of the other language features (OR features). In mainstream feature modeling, such a change requires deleting the old *Alternative* group, creating a new *OR* group, and assigning the *English* feature the type mandatory. Note that adding and removing branches may not always be a straightforward task. In FAM, this change is easily done by adding more option features to the abstract feature *Language* and assigning a *requires* dependency between the features *Language* and *English*.

### c) Lack of Abstraction Mechanisms

As previously mentioned, separation of concerns helps in designing complex and large systems. FAM uses a perspective-based approach to separate concerns and allow in this way to focus on one aspect at the time. Furthermore the intra dependencies between the different perspectives allow linking the different perspectives. In addition, the modeler may opt for an arbitrary number of perspectives. This is opposed to the technique of categorizing features adopted in FODA that groups features using predefined categories. First of all it is not always easy to decide on the category of a feature and secondly it is not an abstraction mechanism but rather a grouping mechanism. Figure 8 illustrates the difference between the two approaches using the Private Branch Exchange (PBX) system [2]. Using FODA (figure 8.a), one model is created to represent the overall system. Such a model can become very large and difficult to understand. Features are grouped together by means of a predefined set of categories: capabilities, operating environments, domain technology, and implementation techniques. Using FAM (figure 8.b), different models are used to model the system. Here we opted for a system, a hardware interface, a functional, and non-functional perspective. Each of those models is smaller, easier to understand, and easier to create as one only has to focus on one aspect of the system. Note that features common between two or more perspectives are shaded.



(a)



(b)

Fig. 8. (a) FODA model of PBX problem (b) FAM model of PBX problem

## 7. Conclusion and Future Work

In this paper we have presented a new multi-perspective feature-oriented technique for modeling variability, called Feature Assembly Modeling (FAM). FAM tried to address some of the limitations of mainstream feature modeling techniques such as lack of abstraction mechanisms, weak support (if any) for complex and large software, and the complexity of the technique for non-experience modelers. We have shown with some examples how FAM eliminates some of the limitations of FODA based feature modeling techniques. The modeling technique presented in this paper is part of the Feature Assembly approach, which also addressed some of the challenges that were not perceived by FODA such as the need for feature reusability.

FAM uses a multi-perspective approach for modeling the variability of a system. Perspectives act as abstraction mechanism enabling better separation of concerns when modeling software. Furthermore, by expressing the dependencies between features of the different perspectives, the different perspectives are interconnected, which provide a more complete picture of the system modeled. In addition, we have reduced the number of modeling primitives used and more importantly, the specification of the information about the variability is separated from the definition of the features, which should improve reusability. Adopting a perspective-based approach for defining features helps identifying the features that are relevant for a particular aspect or viewpoint, thus acting as an abstraction mechanism that helps dealing with complexity.

The next step in the research is to apply the technique to a large industrial case to validate its usability and expressivity. We are also working on a method to collect and document features in a so-called Feature Pool and provide mechanisms for feature reuse (the actual Feature Assembly approach, out of the scope of this paper).

## 8. Acknowledgement

This research is sponsored by the Institute for the encouragement of Scientific Research and Innovation of Brussels. The authors also like to thank Wim Codenie, Nicolás González-Deleito, and Tom Tourwé from Sirris for their valuable discussions regarding the industry needs, which highly contributed to the topic of this paper.

## 9. References

1. Bosch, J.: Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach. Addison-Wesley, Boston (2000)
2. Kang, K.C., Lee, J., Donohoe, P.: Feature-Oriented Product Line Engineering, IEEE Software, vol. 19, no. 4, pp. 58-65 (2002)
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI (1990)

4. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. In: *J. Annals of Software Engineering*. vol. 5, pp. 143-168 (1998)
5. Griss, M., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: *Fifth International Conference on Software Reuse*, pp. 76–85 (1998)
6. Czarnecki, K., Kim, C. H. P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: *OOPSLA'05 International Workshop on Software Factories* (2005)
7. Nestor, D., Thiel, S., Botterweck, G., Cawley, C., Healy, P.: Applying visualisation techniques in software product lines. In: *SOFTVIS 2008*, pp. 175-184 (2008)
8. Bosch J.: Software Product Families in Nokia. In: *SPLC 2005*, pp. 2-6 (2005).
9. MacGregor, J.: Bosch Experience Report, <http://www.conipf.org/download/BoschExperienceReport.pdf>
10. Jaring, M., Krikhaar, R. L., Bosch, J.: Representing variability in a family of MRI scanners, *Software—Practice & Experience*, Volume 34, Issue 1, P: 69 - 100, 2004
11. Maccari, A., Heie, A.: Managing infinite variability in mobile terminal software. *Softw., Pract. Exper.* 35(6): pp. 513-537 (2005)
12. Eriksson, M., Borstler, J., Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In *Obbink and Pohl [24]*, pages 33- 44
13. Clauss, M.: Generic Modeling using UML extensions for variability. In *Workshop on Workshop on Domain-Specific Visual Languages, OOPSLA 2001*, pp. 11-18 (2001)
14. Ziadi, T., Hérouët, L., Jézéquel, J.-M.: Towards a UML Profile for Software Product Lines", In *5th International Workshop Software Product-Family Engineering*, pp. 129-139, (2003)
15. Gomma, H.: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley (2005)
16. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A Domain Ontology for Modeling Variability in Software Product Families. *Advanced Engineering Informatics*, 21(1), pp. 23-40 (2007)
17. M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, COVAMOF: A Framework for Modeling Variability in Software Product Families, In: *SPLC 2004*, pp. 197-213 (2004)
18. Sinnema M., Deelstra, S.: Classifying Variability Modeling Techniques, *Elsevier Journal on Information and Software Technology*, Volume 49, Issue 7, pp. 717-739, July 2007
19. Djebbi, O., Salinesi, C.: Criteria for Comparing Requirements Variability Modeling Notations for Product Lines, In: *CERE '06*. pp: 20-35, (2006)
20. Hubaux, A., Classen, A., Mendonca, M., Heymans, P.: A Preliminary Review on the Application of Feature Diagrams in Practice. In: *VaMoS 2010*, pp. 53-59 (2010)
21. Batory, D.: Feature models, grammars, and propositional formulas. *SPLC 2005*: pp.7-20 (2005)
22. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *Intl. J. of Software Engineering and Knowledge Engineering* 2(1), pp. 31–57 (1992)
23. Nicholas Graham, T.C.: Viewpoints Supporting the Development of Interactive Software. In *Viewpoints 96*, ACM Press, pp. 263-267 (1996)
24. Woods, E.: Experiences Using Viewpoints for Information Systems Architecture: An Industrial Experience Report. *EWSA 2004*: pp. 182-193 (2004)
25. B. Nuseibeh, J. Kramer, and A. Finkelstein, ViewPoints: Meaningful Relationships Are Difficult! , *ICSE 2003*, pp. 676-683 (2003)
26. Abo Zaid, L., Kleinermann, F., De Troyer, O.: Applying Semantic Web Technology to Feature Modeling. In: *SAC 2009*, pp. 1252-1256 (2009)
27. Lopez-Herrejon, R.E., Batory, D.: A Standard Problem for Evaluating Product-Line Methodologies. In: Bosch, J. (ed.) *GCSE 2001*. LNCS, vol. 2186, pp. 9–13 (2001)
28. Czarnecki, K., Eisenecker, U.W.: *Generative Programming: Methods, Tools, and Applications*. Addison Wesley (2000)