Tanhaei M, Habibi J, Mirian-Hosseinabadi SH. A feature model based framework for refactoring software product line architecture. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(5): 951–986 Sept. 2016. DOI 10.1007/s11390-016-1674-y

A Feature Model Based Framework for Refactoring Software Product Line Architecture

Mohammad Tanhaei, Jafar Habibi *, and Seyed-Hassan Mirian-Hosseinabadi

Department of Computer Engineering, Sharif University of Technology, Tehran 11155-9517, Iran

E-mail: tanhaei@ce.sharif.edu; {jhabibi, hmirian}@sharif.edu

Received August 21, 2015; revised July 20, 2016.

Abstract Software product line (SPL) is an approach used to develop a range of software products with a high degree of similarity. In this approach, a feature model is usually used to keep track of similarities and differences. Over time, as modifications are made to the SPL, inconsistencies with the feature model could arise. The first approach to dealing with these inconsistencies is refactoring. Refactoring consists of small steps which, when accumulated, may lead to large-scale changes in the SPL, resulting in features being added to or eliminated from the SPL. In this paper, we propose a framework for refactoring SPLs, which helps keep SPLs consistent with the feature model. After some introductory remarks, we describe a formal model for representing the feature model. We express various refactoring patterns applicable to the feature model and the SPL formally, and then introduce an algorithm for finding them in the SPL. In the end, we use a real-world case study of an SPL to illustrate the applicability of the framework introduced in the paper.

Keywords software product line, feature model, refactoring, software architecture

1 Introduction

In today's competitive world, mass production has given way to mass customization. Mass customization means "producing goods and services to meet individual customer's needs with near mass production efficiency"[1]. For mass customization, the needs of a large number of customers must be considered. The products should be analyzed in such a way that mass production can be done. SPL (software product line) as an approach for developing a domain of products with a lot of similarities, is used in response to these growing needs[2]. Activities in SPL start with domain analysis. The result of domain analysis, which consists of a set of similarities and differences among the products in the domain, is displayed by means of a feature model[3]. As products are added to, or eliminated from the SPL, the domain of the SPL changes.

Changes in the SPL and its features are made constantly. These changes include adding a new feature to a product, eliminating a feature from a product, or customizing a feature which exists in core assets and in a product. Not only software products, but also the reference architecture (which represents the architecture of all products in the SPL), the core assets of the SPL, and the features in the feature model, as an illustrator of all the SPL features, experience frequent changes go through a process of evolving as features are added to them. Because of the continual changes, the reference architecture, the feature model and other artifacts of the SPL move away from their original form, and the level of quality attributes of the SPL such as maintainability and performance lowers[4]. The lack of appropriate change management in the SPL leads to the creation of some inconsistencies among the feature model, the reference architecture, and the core assets. These inconsistencies make it difficult to maintain the SPL and enhance it to support new features.

Refactoring is a process which increases the quality (such as maintainability) of the reference architecture, feature model, and core assets of the SPL, but keeps

Regular Paper
Special Section on Software Systems 2016
*Corresponding Author

the functionalities of the SPL products[5-6] intact. A customized feature in a product might be suitable to be used in other products of the SPL. On the other hand, a feature in the feature model might not be used in any product (dead feature), or those changes in the SPL might eliminate the dependency between two features. The lack of attention to the changes in the SPL leads to the inconsistency between the feature model and the SPL. As an example, the presence of features that are not used in any product causes the unnecessary enlargement of the domain, thereby increasing the complexity of the SPL and increasing maintenance costs.

The process of refactoring on the artifacts at the architecture level (such as the feature model) is usually done manually without the use of any tools and frameworks[7]. Refactoring of the feature model of an SPL with a small number of features and limited products can be done easily. As the number of products and features in the SPL increases, refactoring becomes difficult to be done manually. In this situation, the human analyst has to analyze many features, taking a large number of products into consideration. We study a small sample of an SPL with four products and about 30 features in Section 4. We see that even analyzing an SPL with such a (relatively) small number of features is not simple.

The complexity of finding refactoring opportunities and reviewing products and feature models based on existing patterns indicates the need for more automatic methods that are usable in practice as well as in SPLs with a large number of features and products. Ideally, these automatic methods should be able to identify latent refactoring opportunities, serving as decision-making aids for managers.

In this research, some refactoring patterns in the feature model are presented. Refactoring the feature model causes some changes in other artifacts of the SPL. This is because the changes to the feature model often need to be synchronized through the different artifacts of an SPL. These changes include changes in the reference architecture, changes in the feature mapping, and changes in the artifacts linked to the features in the feature models. Our framework helps in detecting refactoring opportunities on the feature model. We also suggest a way to detect change points in the reference architecture, the feature mapping, and other SPL artifacts such as code. However, our framework performs refactoring only on the feature model. Users should use other frameworks and tools to propagate changes through other artifacts such as architecture and code.

We use the upcoming products of the SPL inline with current products of the SPL to find refactoring opportunities in the SPL. The upcoming products include a product of a product line we perform some modifications on, a new version of a current product or a near future product. The upcoming products usually reveal the development trends in the SPL.

As the first motivating example of performing refactoring on the feature model and removing useless constraints in it, this scenario is considered in an SPL: developers modified a core asset of the SPL, and as a result of the modifications, the need for the inclusion of some other features is obviated. Based on the feature model, products need the required feature to work but the developer team in the development phase has withdrawn the required feature. How can the managers of the feature model find these types of problems in the feature model? Unless they have the configurations of the current and upcoming products, they cannot find the source of the problem or refactoring opportunities in this situation. Our framework helps developers in synchronizing the feature model with the actual core assets in the SPL.

Upcoming product configurations are appropriate indicators of the refactoring opportunities in the feature model. As another example of refactoring feature models and moving a feature from products to the feature model, a candidate feature (a feature that does not exist in the list of features of the feature model) considered is present in the majority of the upcoming products of the SPL but is not present in the feature model. What is a good decision in this situation? Can we add this feature to the feature model?

As the last example of refactoring feature models and removing a feature from the feature model, a situation where a feature of the feature model has not been used in any upcoming product of the SPL is considered. It is a good candidate to be removed from the feature model. As we mentioned before by removing a feature from the feature model, the artifacts related to that feature such as implemented classes, reference architecture, feature mapping, and related code should be changed in such a way that the consistency of the SPL remains intact.

The adoption strategy is one of the important properties of each SPL development approach. It indicates how the software is moving from single-system to an SPL. The adoption strategy can be proactive, reactive, or extractive[8]. While in the proactive strategy, the core assets are built before SPL products, in the reac-

tive strategy, one or more SPL products are built before establishing the core assets. In the extractive approach, the common parts of the existing (legacy) software are extracted and built into a single system. Our framework is applicable on the SPLs which were developed using the reactive or extractive adoption strategies. Although some refactoring patterns introduced in this framework are also applicable in an SPL with a proactive adoption strategy, some important patterns of our framework are not applicable in the SPL using this strategy.

The structure of the remainder of this article is as follows. The requisite background is surveyed in Section 2. The main idea of the paper is proposed in Section 3. In this section, a framework for performing refactoring in the SPL based on the feature model is proposed. Subsection 3.1 explains the idea of big refactoring. Subsection 3.2 presents a formal description of the feature model. In Subsection 3.3, refactoring patterns are defined with regard to the formal description of the feature model, and the set of actions for each pattern is discussed. In Subsection 3.4 we explain the rationale behind using the upcoming products for refactoring SPLs. A set of algorithms to find the refactoring patterns in a feature model is proposed in Subsection 3.5. The practical efficiency of the algorithms is demonstrated in Section 4 by showing their application to a real-world project. Threats to validity of the result of the proposed framework are assessed in Section 5. We discuss the proposed framework in Section 6. In Section 7 we survey the related literature. The paper concludes in Section 8 with a summary and a discussion of possible future work. The complexity of the algorithms proposed in this framework is analyzed in Appendix A.1. We proposed an approach to estimate SPL development cost in Appendix A.2. The experimental evaluation of this framework is explored in Appendix A.3.

2 Background

2.1 Software Product Line

Software product line is a method for developing a set of software products sharing commonalities in a specific domain[9]. Product development is done through the use of existing features in core assets. Specific needs of stakeholders in a particular product are supported through variation points and extensions[3]. To manage the similarities and differences among the products in a domain, a reference architecture along with a feature model[10] is typically used. While reference architecture has some variations in its structure that allows modeling the differences among products of the SPL, the feature model helps in specifying the constraints on the placement of the feature in these variation points. The architecture of the product is an instance of the reference architecture where some choices are done on its variant points[11].

SPL helps in reducing development cost and time to market and software quality improvement[3]. Reusing the core assets in a product leads to savings in the amount of code written. With the continued use of the assets, the assets gradually reach maturity and can exhibit a higher quality attributes level as compared with code written from scratch[9].

2.2 Refactoring

Changing over time is an inherent property of software artifacts in the real world[12]. As the artifacts change, their initial design and quality attributes level downgrade, and consequently, software maintenance becomes difficult[13]. Accordingly, a way is needed to reduce the complexity of the maintenance and the development of software artifacts. One of the common methods of improving the quality of software artifacts and reducing their complexity is refactoring[5].

According to the taxonomy of Opdyke, refactoring is defined as "the process of changing a software system in such a way that it does not alter the external behavior of code, yet improves its internal structure[5]". The refactoring process is not limited to the source code level. It can be considered at the higher levels of development such as design and software architecture[5]. However, refactoring at higher levels can cause large changes in the source code and may impose a large cost on the project.

The refactoring process generally comprises the following steps[5]:

- identifying the location of refactoring;
- identifying the type of refactoring that can be done;
- guaranteeing to maintain the external behavior of code;
- surveying the impact of refactoring on quality attributes;
- maintaining the compatibility between the refactored artifact and other artifacts.

Refactoring projects with a long lifetime is important. Refactoring SPLs, which is a developing methodology with a long lifetime, is of special importance.

Refactoring a product at code level can be done by conventional methods. Performing refactoring at higher levels is, however, usually trickier. One of the problems that might arise is that an artifact can be shared among several products. The user has to make sure that any modification on the software architecture is made only after fully considering the derived products. Hence doing refactoring at the levels higher than the code level in the SPL can be a challenging process.

2.2.1 Software Product Line Refactoring

The term SPL refactoring in this research means any changes made to the SPL artifacts whereby the internal structure of the SPL improves, while the functionality and the capabilities of the SPL are not affected[6,14-16]. The refactoring process may affect any such software artifacts as the architecture, the feature model, the design, and the source code of the SPL.

The changes on the SPL should be done in a way that the applicability of implementing products of the SPL is not affected by performing refactoring. These changes in SPL include removing a dead feature or functionality from the feature model and core assets[17], changes in the variability points[4], and adding a new feature or functionality to the feature model and core assets[6]. Changes on the artifacts such as the feature model should be done in a way that has no impact on the validity of the existing product configuration models[16].

Refactoring changes artifacts at lower levels of abstraction such as code in two ways.

• Changes in the feature model and core assets may necessitate some changes to the code behind them (to support new conditions).

• Removing or adding a feature to the feature model may include or exclude some pieces of code, especially when a feature is linked to some lines of code (LOC) through feature mapping.

2.3 Feature Model

The feature model represents a set of features for all products in an SPL. The similarities and differences of the SPL are captured in the form of a feature model[10]. The feature models represent not only a set of features but also how they are related to each other and can be combined/selected to define products in an SPL. This model was developed and used for the first time in the FODA[18] method.

The feature model is usually represented using a tree-like structure and usually displayed through a visual diagram called feature diagram (Fig.1). The feature diagram can display the mandatory, optional, alternative and OR-groups. The root of the tree stands for the context, and its children represent the features of the SPL.

Besides the items which can be displayed in a tree structure, some cross-tree constraints can be defined in the feature model that cannot be shown in a visual diagram. Two major types of cross-tree constraints are inclusion and exclusion. Inclusion means that with the presence of a particular feature in a product, some (specific) other features must be included in this product as well. Exclusion means that with the presence of a particular feature in a product, some other features must not be included in this product.

Fig.2 shows the feature model of graph product line (GPL). GPL is a standard problem to evaluate product-line development methodologies, developed by Lopez-herrejon and Batory[19]. Some examples of cross-tree constraints in the GPL are as follows:

{Strongly Connected} INCLUDES {DFS, Directed},

{Strongly Connected} EXCLUDES {UnDirected}.

2.4 Feature Model Mapping

For implementing a concrete product from the selected features in the feature model, one needs feature model mapping. Feature model mapping relates each

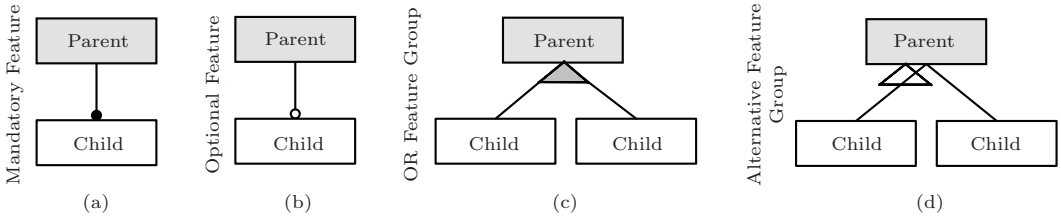

Fig.1. Graphical notation of the feature model[18].

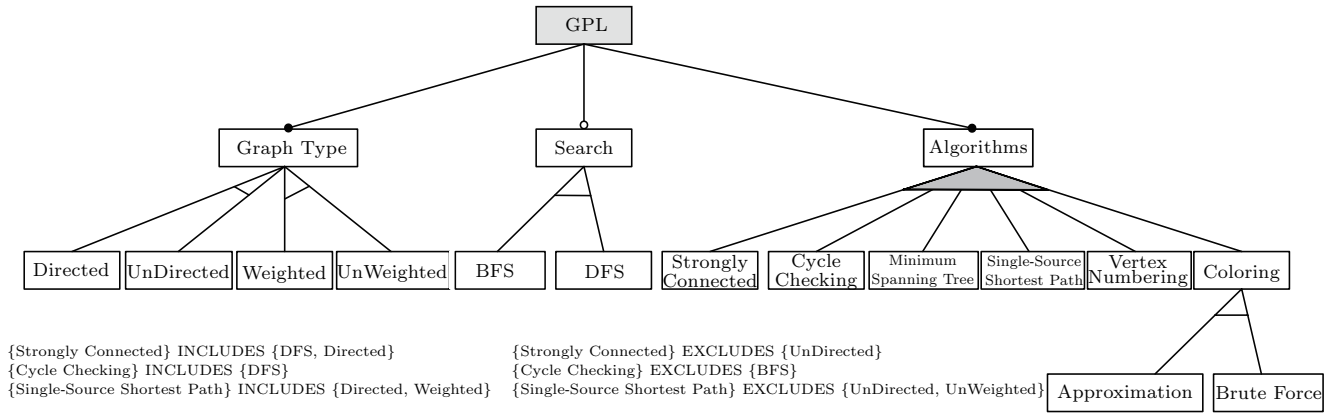

Fig.2. Feature model of graph product line.

feature in a feature model to some artifacts in the solution space[14]. Feature model mapping can be either explicit or implicit. In the case of implicit feature model mapping, the relation between features and software artifacts may be specified in the artifacts, comments, etc. Whereas in the case of explicit feature model mapping we have a particular artifact (e.g., table) to map every feature to some software artifacts. In this paper, we use the explicit form of the feature model mapping. Fig.3 shows a sample of feature model mapping from a feature model to designed classes. However, the level of granularity can vary from very fine to very coarse. Our example is a 1:1 feature mapping. It is possible that we have an $m : n$ feature mapping. The case of $m : n$ feature mapping can be seen regularly in annotative SPLs.

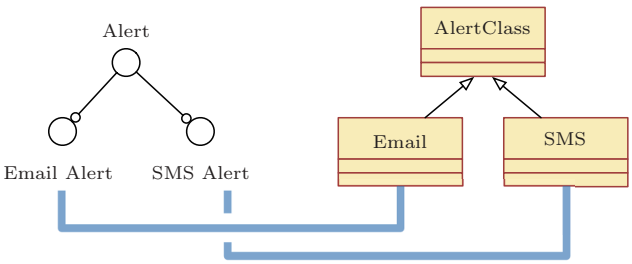

Fig.3. Sample of 1:1 feature model mapping (from features to classes).

2.5 Feature Model Analysis

The implementation of the new products in the feature-oriented product line begins with a selection of the feature from the feature model, and the inclusion of the artifacts related to in this product. Thus it is important to keep the constraint of the feature model consistent and the feature aligned to the existing artifacts in the SPL.

Over time, changes are inevitably made on the feature model. Because of this, the evaluation of the constraints of the feature model is a must. The human analyst can analyze models with a few features and a limited number of cross-tree constraints. With the enlargement of the model and the proliferation of the relationships among the features, performing analysis becomes complicated and cannot be done by a human analyst easily. That is why we need faster and more accurate methods for analyzing the feature model. One of the suitable methods for the analysis of the feature model is converting it to propositional formulas. In this way, each of the relations in the feature model is converted to a law in propositional formulas[10]. Table 1 shows the propositional logic equivalent of each feature model relation type. The notation of $C_1, ..., C_n$ is taken to denote the children of the representative feature P. The variables in the resultant SPL formula are then set to true or false based on the presence or absence of the corresponding feature from the product, and the statement is then evaluated. If the sentence evaluates to the true value, the current configuration satisfies the feature model constraints. Otherwise, the configuration violates at least one of the constraints.

Table 1. Propositional Formula[10]

Type	Propositional Formula
Mandatory	$P \Leftrightarrow C$
Optional	$C \Rightarrow P$
OR-group	$P \Leftrightarrow \bigvee_{1 \leqslant i \leqslant n} C_i$
Alternative group	$(P \Leftrightarrow \bigvee_{1 \leqslant i \leqslant n} C_i) \wedge \bigwedge_{j<k} (\overline{C_j} \vee \overline{C_k})$
A includes B	$A \Rightarrow B$
A excludes B	$\overline{A \wedge B}$

As an example, the propositional formula of GPL can be described as follows.

```
GPL ∧
//tree constraints:
(GPL ⇒ Graph_Type ∧ Algorithms) ∧ (Graph_Type
     ∨ Algorithms ⇒ GPL) ∧ (Search ⇒ GPL)
     ∧···∧
//cross-tree constraints:
Strongly_Connected ⇒ DFS ∧ Directed ∧ (
     Strongly_Connected ∧ UnDirected)   ∧···∧
```

3 Feature Model Based Refactoring

3.1 Big Refactoring

In general, refactoring can be done at every level of abstraction. The lowest refactoring level is the refactoring of the source code. At this level, such refactoring tasks as renaming, deleting, removing circular dependencies, breaking methods into several parts, and code reuse are considered. The refactoring process can be done on higher-level artifacts such as software design and software architecture. The scope of changes created by this type of refactoring is much broader than the scope of changes in code level refactoring. The cost and the time required to make changes at the architecture level are much higher than those for code-level changes.

Many identifiable refactoring opportunities at the architecture level are either not detectable or too costly to detect at the code level. That is because of the high impact of the software architecture on the software quality attributes. This research presents a systematic approach for identifying and carrying out this type of refactoring based on the feature model.

In Subsection 3.3, different types of refactoring patterns in the SPL are defined, and appropriate refactoring acts are described. The refactoring acts are the activities done if the conditions of the refactoring pattern are matched.

3.2 Feature Model

In this subsection, we present a formal definition to describe the feature model. We use this definition later to describe the refactoring patterns. Our definition is an extension to the work of Schobbens *et al.*[20] We present the notion of candidate feature and upcoming product based on Schobbens *et al.*'s definition[20]. We also convert the textual constraints in Schobbens *et al.*'s definition into two independent entries constituting inclusion and exclusion.

To define the feature model, we first explain the concepts used in our definition and then describe the feature model using these concepts.

• *FT.* Feature type specifies the type of the feature in the feature model. The feature type can be one of the types OR, Alternative, Optional, or Mandatory. The feature type can be defined on a set of nodes. In these cases, the type is determined by a cardinality number such as $Alternative_k$.

• *Abstract and Concrete Features.* Abstract features are the features not implemented in the SPL which do not have any impact on the implementation of the SPL[21]. The opposites of the abstract features are concrete features. Concrete features are the features which are used in the implementation of the SPL (features that link to an artifact in the implemented products).

• *Primitive and Compound Features.* Primitive features are the leafs of the feature model. Another type of features is compound features, which are used to categorize the primitive features in the feature model (for example, the root of the feature model is a compound feature). The type of the primitive and compound features can be either abstract or concrete. However, primitive features are the features that usually provide actual functionality in the SPL (they are usually concrete).

• *Candidate Feature.* A candidate feature is a feature which is implemented or scheduled to be implemented in some products of the SPL in the near future but does not exist in the core assets of the SPL. This type of features may be transferred to the SPL core assets in the evolution of the SPL. Because of this, we named these types of features as the candidate features. Candidate features can be either concrete or abstract. We have used this type of feature later in our reasoning algorithms.

• Op_k. Op_k in this feature model definition means an operation such as OR and Alternative on k input arguments.

Definition 1 (Feature Model). *An FM (feature model) is specified as* $d \in FM(FT) = (N, N', P, r, \lambda, \phi, \chi, DE)$, *where*:

• N *is the set of FM nodes.*

• N' *is the set of candidate features nodes.*

• $P : \wp N$ *is the set of FM primitive nodes.*

• $DE : \wp N \leftrightarrow N$ *is the list of FM edges,* $(n, m) \in DE$, *also shown as* $n \to m$.

• $\lambda : \wp N \leftrightarrow FT$ *labels each (group of) feature(s) with an operator from FT.*

• $\phi : \wp N \leftrightarrow N$ *is the list of inclusion relationships;* $(n, m) \in \phi$ *means if* n *is present in a product,* m *should be also there.*

• $\chi : \wp N \leftrightarrow N$ *is the list of exclusion relationships;*

$(n, m) \in \chi$ *means if* n *is present in a product,* m *cannot be present in it.*

• $r : N$ *is the root of a tree, which means* $\nexists n \in N \times n \to r$.

• *DE is acyclic*: $\nexists n_i, \cdots, n_j$ *where* $n_i \to \cdots \to n_j \to \cdots \to n_i$.

• *DE is a tree*: *there is a unique path from* r *to every other feature.*

• $N \cap N' = \varnothing$, N *and* N' *are disjoint.*

Definition 2 (Configuration). *A configuration is a subset of primitive features of the feature model.* $m \in \wp P$ *is a configuration.*

Definition 3 (Valid Configuration). *A configuration* $m \in M$ *in the feature model* $d \in FM$ *is valid and displayed as* $m \vdash d$ *, if the following conditions are met.*

1) $\forall \gamma \subseteq m, \nexists \lambda(\gamma) = Op_k \leftrightarrow Op_k(S_1 \in \gamma, \cdots, S_k \in \gamma) \equiv false$, *conforms to all feature placement restrictions.*

2) $(n_i \in m \wedge (n_i, n_j) \in \phi) \Rightarrow n_j \in m$, *inclusion constraints are not violated.*

3) $(n_i \in m \wedge (n_i, n_j) \in \chi) \Rightarrow n_j \notin m$,*exclusion constraints are not violated.*

Definition 4 (Product). *Product is defined as follows*:

Product: $[\![m]\!] = m \cap P \neq \varnothing \wedge [\![m]\!] \in \wp N$.

A configuration m *is a product if it is a non-empty subset of the primitive features. The product derived from configuration* m *is shown as* $[\![m]\!]$.

In this paper, we use upcoming products of the SPL as a source to find refactoring opportunities in the SPL. Upcoming products consist of two types of the products.

• *Future Products.* This type of products is scheduled to be implemented in the near future using SPL facilities. This type of products shows the strategy of the SPL evolution.

• *New Version of the Current Products.* The requirements of the products in the SPL are changed over time. SPL as an approach with high speed in response to the demands of the stakeholders should prepare itself for the new version of the current products. In this regard the current products may be changed to support new requirements of the stakeholders.

Definition 5 (Upcoming Product). *We display the candidate features and the upcoming products by adding an apostrophe* ($'$).

Upcoming Product: $[\![m']\!] = m' \cap \{P \cup N'\} \neq \varnothing \wedge [\![m']\!] \in \wp\{N \cup N'\}$.

An upcoming product can be formed using the existing primitive and candidate features of the feature model. While the configuration of the current product should be valid, there is no constraint on the validity of the upcoming product of the SPL. However, the upcoming product should be validated against the feature model when it wants to be a real product of the SPL.

Definition 6 (Product Line). *A software product line is defined as follows.*

ProductLine: $[\![d]\!] = \{[\![m]\!] \cup [\![m']\!] \mid m \vdash d\} \wedge [\![d]\!] \in \wp\wp N$. *The products resulting from* d *are shown as* $[\![d]\!]$. *The SPL* d *is a set of the models with regard to* $l \in FM$*, and is shown as* $d \vDash l$.

An SPL in our definition consists of two types of products, current and upcoming products. The current products of a product line are the valid products of the feature model.

3.3 Refactoring in SPL

In SPL refactoring, it is possible that an artifact related to each feature (via feature mapping) is changed in such a way that it might violate some feature model constraints. For example, consider the alert system in an SPL which is an alternative of SMS, email, or phone call. Each of these features is related to an artifact such as SMS class, email class, and call class via feature mapping. These classes are derived from a master class named alert class. Over the time, the alert class changes so much that the derived classes from it can be present in the same product. In other words, the derived classes from it are no longer alternatives of each other. As a result, one can find some products in the SPL that are not validated against the feature model of the SPL, but perform their functionalities without any problems. For example, a product containing the SMS and email features simultaneously is not validated based on the feature model with alternative constraint between them, but performs its duties uninterruptedly. In this case, the inconsistency between the feature model and the product configuration results from the desynchronization between features of the feature model and the actual features used in the SPL core assets. In this situation, the feature model and its constraints should be repaired and synchronized with the actual features of the SPL.

An evolution opportunity appears when one or more upcoming products' configurations are not valid, or some product configurations violate one or more constraints on the feature model. In these situations, the feature model and each product configuration are examined. Model invalidation may occur for two reasons.

• *Human Error* (*Product Configuration Invalidity*). The first cause for the invalidity of a product configura-

tion is human error. In such cases, the configuration of the actual product should be investigated. In case that the product does not properly fit in with the configuration, the inconsistencies between the actual product and the product configuration should be rectified in relation to the feature model and the constraints defined. In case that the product configuration is invalid, the feature model will remain intact. Batory *et al.* mentioned these types of problem in the SPL and surveyed the solutions to ratify them in [22].

• *SPL Evolution.* Here the human error does not cause the invalidity of a product configuration. For instance, in the SPL feature model, two features should be used simultaneously in a product to ensure desirable operation, but in the product under study, one of the features alone suffices to ensure proper operation. In this scenario, the invalidity of the product configuration stems from the inconsistencies in the feature model. In the above example, the inconsistency can be attributed to one of the features growing independent of the others during the development process. Detecting problem points in these situations becomes increasingly infeasible as the model grows bigger. The alert class we introduced at the beginning of Section 3 is another example of the inconsistencies in the feature model that the SPL evolution can cause. The evolution in the alert class breaks the alternative constraint in the feature model.

SPL is examined in regular time periods to detect and correct any errors related to the invalidity of the product configuration related to the feature model. Two causes are behind most violations of validity in the feature model: human error (invalidation of the product configurations) and the SPL evolution (inconsistency of the feature model). In case of human error, one investigates the wrong places in the product configurations and attempts to fix them.

It gets challenging when a suspected product performs its duties correctly. In this case, the models which are used to demonstrate the architecture and features of the SPL should be evaluated. In fact, it is possible that for some reasons, the changes in the products might not be mirrored in the feature model, which is supposed to reflect all product configurations. Refactoring methods introduced in this subsection provide ideas for finding the problems in the feature model and fixing them systematically. In the following we will define different refactoring patterns in the SPL by using the feature model.

3.3.1 *Absence of a Candidate Feature in the Feature Model*

It is possible that a feature which is present in the upcoming products $m'_{p_1} \sim m'_{p_k}$ is not considered as a feature of the feature model. In other words, a feature might be developed in a series of products, but be absent from the feature model. In this situation, the feature is a proper candidate to be added to the SPL artifacts such as the feature model, the reference architecture, and the core assets. By adding the feature to the core assets, the overall functionality of the SPL (the reference architecture, the feature model, and the SPL products) does not change. In this situation, some features from the SPL products are moved to the core assets, and no new functionality is added to the SPL.

The condition of this refactoring opportunity is as follows:

$$\exists f \notin N \wedge f \in N' \cdot f \in m'_{p_1} \cap m'_{p_2} \cap \cdots \cap m'_{p_k}.$$

Possible Refactorings in the SPL:

1) Adding a feature to the feature model, reference architecture, and core assets.

Transferring the features to the feature model, the reference architecture, and core assets should go through if the benefits of doing so (or the cost of not doing the transfer) outweigh its costs; otherwise it should not be done. For estimating the cost of adding a feature to the core assets and the feature model, as well as determining the cost of developing the SPL, methods such as COPLIMO[23] can be used. In this paper, we discuss the usage of the COPLIMO[23] approach in Appendix A.2. However, users can use other methods for estimating the benefits of adding or removing a feature from the feature model.

When a feature is used in some current and upcoming products of the SPL, the corresponding feature type in the feature model will be optional. Fig.4 shows an example of adding a feature to a feature model. FD means feature diagram; C1 and C2 mean component 1 and component 2 respectively; F1, F2, F3 mean feature 1, feature 2, feature 3 respectively; P′1, P′2, P′3 mean upcoming product 1, upcoming product 2, upcoming product 3 respectively. Adding a feature to a feature model is described formally as follows:

$$\begin{aligned}
&N = N \cup f,\\
&N' = N' \backslash f,\\
&DE = DE \cup (r, f),\\
&\lambda = \lambda \cup (f, \{optional\}),\\
&P = P \cup f.
\end{aligned}$$

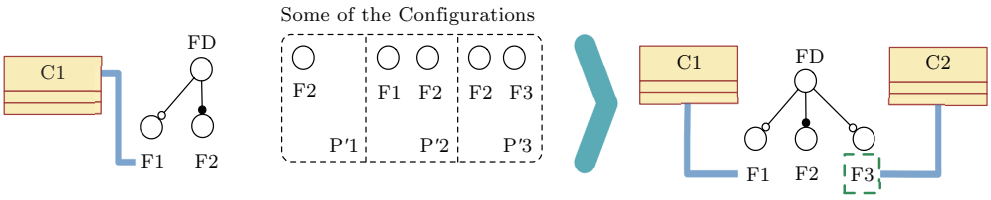

Fig.4. Adding a feature to the feature model.

Adding a new feature to the reference architecture may change the reference architecture. However, there is no automatic or semi-automatic way to find the location of the changes in the reference architecture. The task of adding a new feature to the reference architecture is not trivial. We provide an example of adding a new feature to a reference architecture in the case study (Section 4). By adding a new feature to the feature model, all artifacts related to this feature such as classes, code, should be investigated and added to the core assets. The feature mapping artifact should be changed in such a way that it can reflect the new relation between imported features and the core assets.

2) Avoiding to perform changes in the feature model. If the benefits of moving a feature to the feature model, reference architecture, and core assets are not favorable, the feature should not be transferred to them. We present some examples of this situation in Section 4.

3.3.2 *Absence of a Mandatory Feature in Some Upcoming Products*

It is possible that a feature which is defined as mandatory in the feature model does not appear in the SPL upcoming products $m'_{p_1} \sim m'_{p_k}$.

$$\exists\,(L, \{mandatory\}) \in \lambda \cdot L \subset P \wedge L \notin m'_{p_1} \cup \cdots \cup m'_{p_k}.$$

Possible Refactorings in the SPL: changing the type of the feature from mandatory into optional.

It is possible that a mandatory feature will not be the basic feature of the SPL anymore, because a new feature is taking its place (for instance, there is a viable alternative to the feature) or other features of the SPL do not need this feature any longer. In such a case, the type of feature can be converted to optional one (Fig.5).

Converting a mandatory feature to an optional feature in the feature model is represented formally as follows:

$$\lambda = [\lambda \backslash (L, \{mandatory\})] \cup (L, \{optional\}).$$

There is no need to change the reference architecture and the feature mapping in performing this refactoring on the feature model. In most cases, performing this refactoring does not need to change the core assets. However, this is not true in all cases. A piece of code cannot simply be made optional without modifying the other parts of the code base, because the rest of the code base could rely on the code of the feature.

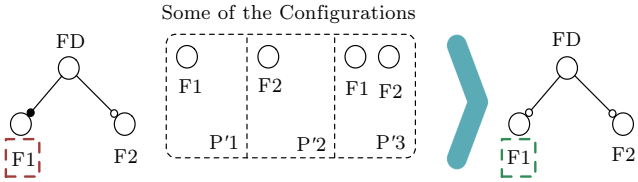

Fig.5. Converting mandatory feature to optional feature.

3.3.3 *Converting an Optional Feature into a mandatory Feature*

It is possible that an optional feature appears in all current products of the SPL from m_{p_1} to m_{p_n} and the upcoming products of the SPL from $m_{p'_1}$ to $m_{p'_{n'}}$.

$$\exists\,(L, \{optional\}) \in \lambda \wedge L \subset P \cdot L \subseteq m_{p_1} \cap \cdots \cap m_{p_n} \cap m_{p'_1} \cap \cdots \cap m_{p'_{n'}}.$$

Possible Refactorings in the SPL: changing the type of the feature from optional into mandatory.

The type of a feature that is present in all SPL product configurations can be moved from optional to mandatory. The modifications on a feature can convert the feature to a basic feature which should be present in all products of the SPL, or other features can be edited in such a way that they need this feature to perform their functionality correctly. In this case, the feature type can be converted to mandatory one (Fig.6).

We present the refactoring of changing an optional feature to a mandatory feature formally as follows:

$$\lambda = [\lambda \backslash (L, \{optional\})] \cup \{(L, \{mandatory\})\}.$$

This refactoring has no impact on the reference architecture, the core assets, and the feature mapping artifacts.

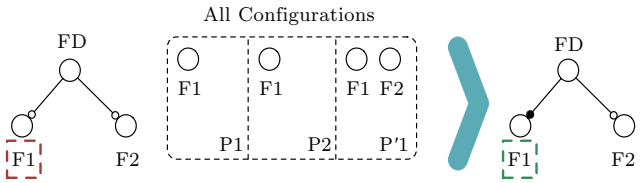

Fig.6. Converting optional feature to mandatory feature.

3.3.4 *Absence of a Member of an Alternative/OR Feature Group in All Products*

It is possible that a member of an alternative/OR feature group is not used in any of the current and upcoming products of the SPL:

$$\exists\,(L,\{alternative/or\}) \in \lambda \wedge f \in L \cdot f \in P \wedge f \notin m_{p_1} \cup \cdots \cup m_{p_n} \cup m_{p'_1} \cup \cdots \cup m_{p'_{n'}}.$$

Possible Refactorings in the SPL: removing a feature from an alternative/OR-group.

When an alternative/OR feature does not appear in all products of the SPL, it can be removed from the alternative/OR-group and become an independent optional feature. Fig.7 shows an example of converting an alternative feature to an optional one. Removing a feature from an alternative/OR-group is as follows:

$$\lambda = [\lambda \backslash (L,\{alternative/or\})] \cup (L \backslash f,\{alternative/or\}) \cup \{(f,\{optional\})\}.$$

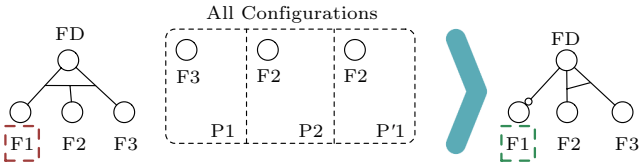

Fig.7. Converting type of a member of an alternative group to optional one.

By performing this refactoring, the reference architecture and the feature mapping artifacts do not change. Based on the alternative group implementation mechanism, we may need to adjust the code to accommodate this refactoring. For example, when an alternative group is implemented using a switch and *#ifdef* annotation inside, we need to extract the case of optional feature and write another *#ifdef* for implementing the optional feature condition in the code.

3.3.5 *Violation of Alternative Group Rule*

Two features of an alternative feature group might appear in one or more upcoming products at the same time:

$$\exists\,(L,\{alternative\}) \in \lambda \times \exists n_j \in L \wedge n_k \in L \wedge m'_{p_i} \cdot [n_j \cup n_k] \subset P \wedge (n_j \in m'_{p_i} \wedge n_k \in m'_{p_i}) \wedge (n_j \neq n_k).$$

Possible Refactorings in the SPL: transforming an alternative member feature to optional one.

In this case, some features can be removed from the alternative group. To do this, the causal relationships among the features need to be determined. Dependency analysis has to be done in this situation[24]. If the presence of a feature of an alternative group forces the configuration to include another feature of that group, it can be thrown out from the alternative group. Fig.8 shows an example of this situation. Removing an element of an alternative group is represented as follows (assume that n_j is the cause of violation of the alternative constraint in the group):

$$\lambda = [\lambda \backslash (L,\{alternative\})] \cup (L \backslash n_j,\{alternative\}) \cup \{(n_j,\{optional\})\}.$$

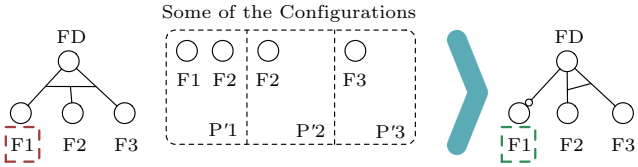

Fig.8. Transforming an alternative group member to an optional one.

The reference architecture and the feature mapping artifacts are not affected by performing this refactoring. As described in the previous subsection, based on the implementation mechanism we may need to adjust the code to accommodate this refactoring.

3.3.6 *Optional Feature Not Present in Any Product*

An optional feature might not be used in any product of the SPL:

$$\exists\,(L,\{optional\}) \in \lambda \wedge L \subset P \cdot L \notin m_{p_1} \cup \cdots \cup m_{p_n} \cup m_{p'_1} \cup \cdots \cup m_{p'_{n'}}.$$

Possible Refactorings in the SPL: removing the feature from the feature model, the reference architecture, the feature mapping, and the core assets.

When an optional feature is not used in all products of an SPL, the presence of this feature in the feature model adds complexity to the feature model and increases the cost of maintenance of the SPL. In this case, one can remove this unused feature from the SPL safely. Fig.9 shows an example of this refactoring. In removing a feature from a feature model, one should

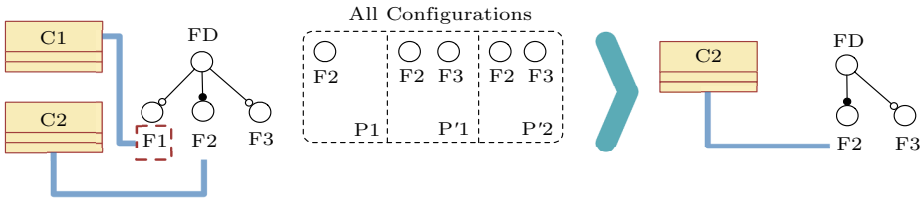

Fig.9. Removing an unused optional feature.

carefully consider the feature mapping artifacts. The artifacts that only relate (directly or indirectly) to the removed feature have to be dropped from the SPL core assets. The formal representation of removing an optional feature from a feature model is as follows: (in this definition $\triangleright$ means range restriction and $\triangleleft$ means domain restriction).

$$
\begin{aligned}
&N = N \backslash L,\\
&\lambda = \lambda \backslash (L, \{optional\}),\\
&P = P \backslash L,\\
&DE = DE \backslash [DE \triangleright L],\\
&\phi = \phi \backslash [\phi \triangleright L \cup \phi \triangleleft L],\\
&\chi = \chi \backslash [\chi \triangleright L \cup \chi \triangleleft L].
\end{aligned}
$$

To perform this refactoring, one should carefully investigate the SPL reference architecture. The feature mapping at the architecture level can be used in this regard. If some parts of the reference architecture are useless in the SPL (only related to the removed feature), they can be marked for further investigation and duly removed from it.

Other artifacts related to the removed feature such as classes and code also should be investigated. If the relations between the feature and these artifacts are $1:1$, the artifacts can be removed safely. In the case of an $m:1$ or $m:n$ relation between feature(s) and core asset artifacts, some investigations should be done. Finding dead parts in code can help in this regard. This work can be done by using the partial dead code removal approach described in several books and papers such as the book written by Knoop *et al.*[25]

In the end, after removing the feature from the feature model, the links between the deleted features and other artifacts should be removed from feature mapping artifacts.

3.3.7 *Transforming the OR Feature Group to the Alternative Feature Group*

If among the features of an OR-group, just one feature appears on each product, the type of these features can be changed to alternative one:

$$
\begin{aligned}
&\exists (L, \{or\}) \in \lambda \wedge n_1 \in L \wedge \cdots \wedge n_k \in L \cdots \nexists_{j<l} \wedge j \in \\
&\{1 \cdots k\} \wedge l \in \{1 \cdots k\} \wedge m_{p_i} \cdot (n_j \in m_{p_i} \wedge n_l \in \\
&m_{p_i}) \wedge \forall m_{p_i} \cdot \exists q \in \{1 \cdots k\} \cdot n_q \in m_{p_i}.
\end{aligned}
$$

Possible Refactorings in the SPL: transforming type of the OR feature group to the alternative one.

By performing this refactoring, the type of the OR-group is transformed to alternative one (Fig.10). This can be described formally as follows:

$$\lambda = [\lambda \backslash \{(L, \{or\})\}] \cup \{(L, \{alternative\})\}.$$

The reference architecture and the feature mapping artifacts are not changed by performing this refactoring. Based on the implementation mechanism of the OR-group, we may need to change the code to accommodate changes.

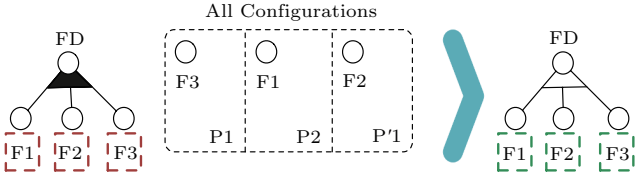

Fig.10. Example of converting type of OR-group to alternative.

3.3.8 *Adding Optional Node to Alternative Group*

Sometimes, an optional feature can be joined to its neighbor alternative group (the alternative group and the optional node share the same parent feature). The condition of this refactoring is as follows:

$$
\begin{aligned}
&\exists\, (l_1, \{alternative\}) \in \lambda \wedge (l_2, \{optional\}) \in \lambda \wedge f_1 \in \\
&l_1 \wedge f_2 \in l_2 \wedge h \rightarrow f_1 \wedge h \rightarrow f_2 \wedge (\forall f \in l_1 \exists m_{p_i} (f \notin \\
&m_{p_i} \wedge f_2 \in m_{p_i})) \wedge (\exists f \in l_1 \exists m_{p_i} (f \in m_{p_i} \wedge f_2 \notin m_{p_i})).
\end{aligned}
$$

Possible Refactorings in the SPL: adding the optional feature to the alternative group.

The optional feature can be added to the neighbor alternative group (Fig.11). In the following, we describe this refactoring:

$$\lambda = [\lambda \backslash \{(l_1, \{alternative\}), (l_2, \{optional\})\}] \cup \{(l_1 \cup l_2, \{alternative\})\}.$$

Performing this refactoring has no impact on the feature mapping. In adding an optional feature to an alternative group, the reference architecture may change. If the component of the optional feature differs from the components of the alternative group feature, we need to transform the functionality of the optional feature into the component of the alternative group. To perform this refactoring, we may need to adjust the code to include a new optional feature in the implementation of the alternative group in the code.

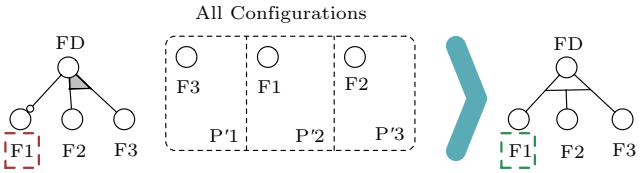

Fig.11. Adding optional feature to an alternative feature group.

3.3.9 *Violation of the Inclusion Constraint*

This refactoring is applicable if some inclusion constraints are violated in some SPL upcoming products. The constraint violation circumstance is as follows:

$$\exists\,(L, H) \in \phi \wedge m'_{p_k} \cdot L \in m'_{p_k} \wedge H \notin m'_{p_k}.$$

Possible Refactorings in the SPL: removing the constraint from the feature model inclusion constraints.

Inclusion constraints are generally imposed because one component depends on the services of another. If component A can do its required services without needing those of component B, any rule stating the dependency of A on B should be removed. The break-up of the dependence of a component on another may be caused by the modifications done on the SPL components or human errors. A sample of removing an inclusion constraint from a feature model is shown in Fig.12. Removing an inclusion constraint from a feature model is represented as follows:

$$\phi = \phi \backslash (L, H).$$

After removing an inclusion constraint from the feature model, there is no need to make changes to the reference architecture, the core assets, and the feature mapping artifacts to keep the consistency among the artifacts in the SPL.

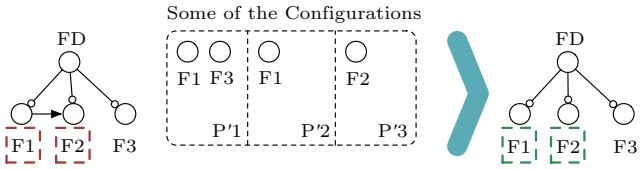

Fig.12. Removing inclusion constraint example.

3.3.10 *Violation of the Exclusion Constraint*

If an exclusion constraint is violated in some SPL upcoming products, this refactoring can be applied. The condition of the exclusion constraint violation is as follows:

$$\exists\,(L, H) \in \chi \wedge m'_{p_k} \cdot L \in m'_{p_k} \wedge H \in m'_{p_k}.$$

Possible Refactorings in the SPL: removing the constraint from the feature model exclusion constraints.

Exclusion restriction is imposed due to any specific inconsistency between two components. If the two mutually exclusive components can appear in a product of SPL simultaneously, the corresponding exclusion rule must be removed. Fig.13 shows an example of removing an exclusion constraint from a feature model. Removing an exclusion constraint from a feature model is represented formally as follows:

$$\chi = \chi \backslash (L, H).$$

Removing an exclusion constraint from a feature model has no impact on the reference architecture, the core assets, and the feature mapping artifacts.

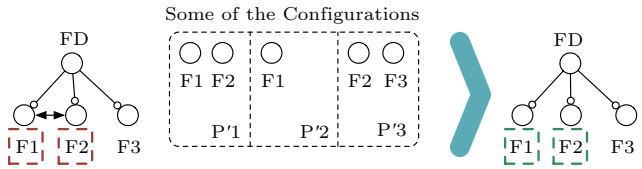

Fig.13. Removing exclusion constraint example.

3.4 Rationale Behind Using Upcoming Products for Refactoring

Software product line is a development approach with a long life cycle. This approach is used to develop a spectrum of similar products in a specific domain. Based on this goal, a reference architecture and a set of core assets are developed in the SPL. After adding new requirements by the stakeholders and the environment, SPL can take two ways:

1) perform domain analysis again and add support for the new requirements to the SPL (clean approach);

2) add support for the new requirements to the products which need them and postpone changing the SPL to the near future (dirty approach).

While the best way to manage SPL is to perform domain analysis again and add the requirements to the SPL, the second approach may nevertheless be considered as a quicker alternative in development teams (if you are interested, you can take look at the technical debt metaphor[26]). This is mainly done because of the time pressure and cost constraints in many situations. To save SPL from being a method of developing several disparate products, the team should consider synchronizing the product with the SPL architecture and core assets from time to time.

According to new research[27], cloning is a mechanism which is widely used in the industry to form a product line. Cloning decreases the cost of the development and increases the flexibility and the independence of SPL. It is known as an appropriate mechanism[27] to form an SPL. Cloning has its overhead for organizations: synchronizing products with SPL architecture and core assets in this mechanism is difficult, propagating changes between clones is difficult, and finally, repetitive tasks are common in this mechanism.

One of the approaches to deal with the problem of synchronizing products with SPL architecture is to use a refactoring framework similar to the one developed in this paper. Our approach facilitates the propagation of changes between the products and the SPL architecture by adding or removing features used or not used in the SPL products. Consider the following scenario for an SPL. Feature A requires feature B to perform its functionality. Over time, feature A changes in a way that it no longer depends on feature B for performing its duties. After synchronizing the product with the reference architecture of the SPL, the modified version of feature A is imported to the core assets. In this situation, we have some inconsistencies in the SPL feature model. The feature model forces the product to include feature B when feature A is added to its configuration, but feature A no longer uses the functionality offered by feature B. Our refactoring patterns can find these types of inconsistencies between the feature model and the current and upcoming products' configuration and make suggestions to the developer team to modify the feature model, the reference architecture, and the code accordingly.

3.5 Reasoning on Feature Model

Different types of refactoring on the SPL feature model were described in Subsection 3.3. These refactorings were defined by considering the current and upcoming products' configuration. Finding the refactoring opportunities in the feature model is difficult in practice. In the remainder of this subsection, we propose several algorithms for reasoning about refactoring in the feature model.

There already exist several tools for supporting analyses on the feature model (such as FAMA[28] and SPLOT[29]). However, these tools do not support analyses on the features like *Candidate* features which we used in our refactoring catalogs. As Thum *et al.*[21] suggested, an approach that uses propositional formulas can be used to do this type of analysis. In this subsection, we introduce our algorithms which use propositional formulas in order to find refactoring opportunities in a feature model.

3.5.1 Reasoning Algorithms

The first algorithm proposed in this subsection is the algorithm which makes propositional formula from a start feature r in the feature model tree (Algorithm 1). This algorithm uses the mechanism described in Subsection 2.5. In this algorithm, every feature model (which starts from feature r) is converted to a propositional string (lines 12~41). PA is a static array that saves the propositional formula of a feature model (line 46). For example, $PA['sample']['root']$ saves the propositional formula for a tree starting from *root* in *sample* feature model. This algorithm recurs on all child features of r (line 43) until it reaches the leaf of the feature model where the feature r equals null (lines 3~5). In order to decrease the complexity of the algorithm, we save the propositional formulas we computed previously and use them if we need them later in lines 6~8.

The next algorithm evaluates propositional strings based on a given feature model and a product configuration (Algorithm 2). The $ParentsOf(FM, m)$ method in line 3 returns the set of parents of the primitive features in m. In this algorithm, all features that are present in product configuration m and the parents of these features receive the true value (line 12). The value of the reminder of the features is set to false (line 14). The $BDD(P)$ method (line 21) evaluates the input binary string using binary decision diagrams (BDD). The input to the BDD method is a string in the form of true

Algorithm 1. Create Propositional Formula

```
1  Function: CreateP (FM, r)
   Data: FM is the feature model; r is the root of the feature model
   Result: returns propositional formula of feature model FM starting from r
2  begin
3      if (r = null) then
4          Return "true";
5      end
6      if (PA[FM.name][r.name] ≠ null) then
7          Return PA[FM.name][r.name];
8      end
9      P = "true";
10     foreach (r, f) in FM.DE do
11         if ((ε, FT) in FM.λ) and (f in ε) then
12             switch FT do
13                 case "Mandatory" do
14                     P = "P"+ "∧" + "(r.name ⇔ f.name)";
15                 end
16                 case "Optional" do
17                     P = "P"+ "∧" + "(f.name ⇒ r.name)";
18                 end
19                 case "Or" do
20                     Ors = " ";
21                     foreach (m in ε) do
22                         Ors = Ors + "∨" + m.name;
23                     end
24                     P = P + "∧" + "(r.name ⇔" + Ors +")";
25                 end
26                 case "Alternative" do
27                     Ors = " "; Ands = " ";
28                     foreach m in ε do
29                         Ors = Ors + "∨" + m.name;
30                     end
31                     foreach m in ε do
32                         foreach n in ε do
33                             if (m == n) then
34                                 Break;
35                             end
36                             Ands = Ands + "∧" + "¬m.name ∨ ¬n.name";
37                         end
38                     end
39                     P = P + "∧" + "(r.name ⇔"+Ors+") ∧ ("+Ands+")";
40                 end
41             end
42             foreach (f, c) in FM.DE do
43                 P = P + "∧" + CreateP (FM, c);
44             end
45         end
46         PA[FM.name][r.name] = P
47         Return PA[FM.name][r.name]
48     end
49 end
```

Algorithm 2. Evaluate Propositional Formula

```
1  Function: EvaluateP (P, FM, m)
   Data: P is propositional formula; FM is the feature model; m is a product configuration
   Result: true if propositional strings equal true regarding to members of m
2  begin
3      Parents = ParentsOf(FM, m);
4      pflag = false;
5      for (i=0; i<P.lenghth; i++) do
6          if (P[i] <> '∧' and P[i] <> '∨' and P[i] <> '⇔' and P[i] <> '⇒' and P[i] <> '(' and P[i] <> ')') then
7              p = p + P[i];
8              pflag = true;
9          else
10             if pflag then
11                 if (p ∈ m ∪ Parents) then
12                     P.Replace(p, true);
13                 else
14                     P.Replace(p, false);
15                 end
16                 p = '';
17                 pflag = false;
18             end
19         end
20     end
21     flag = JavaBDD(P)
22     Return flag
23 end
```

and false values with the operations between them. Libraries such as JavaBDD[1] can be used to implement Algorithm 2.

Algorithms 3~5 are used to check mandatory, optional, and alternative conditions in a tree respectively, starting from feature f. These algorithms use the propositional formula evaluation algorithm (Algorithm 2) and the propositional formula creation algorithm (Algorithm 1).

Algorithm 3 returns false if it finds a mandatory rule violation in the feature model starting from feature f regarding input models M (line 7). It returns true if the mandatory rule is not violated in the feature model starting from feature f with regard to input models M.

Algorithm 4 checks the optional rule on the feature model starting from feature f with regard to input models M. It returns true if the optional rule is true concerning some input models (line 7). It returns false if it cannot find any condition in which the optional rule is true.

Algorithm 5 checks alternative rules between the children of a given feature f. It iterates on all input models (line 4) and checks refactoring alternative rule on each input model (lines 6~11). The alternative rule is true with regard to an input model m, if only one of the children of feature f is present in that model. The algorithm counts the number of locations that the children of feature f are true (line 9). If this number is greater than one or equals zero (line 12), the alternative rule is violated, and the algorithm returns false.

Algorithm 3. Check Mandatory Propositional Formula

```
1  Function: CheckMandatoryP (FM, f, M)
   Data: FM is feature model, f is the root of the feature
         model, and M is a set of product configurations
   Result: true if feature f exists in all products and the
           tree having root f has no contradictions with
           tree-constraints
2  begin
3      P = CreateP(FM, f)
4      flag = true
5      foreach m ∈ M do
6          if EvaluateP(P, FM, m) = false then
7              flag = false; break;
8          end
9      end
10     Return flag;
11 end
```

Algorithm 4. Check Optional Propositional Formula

```
1  Function: CheckOptionalP(FM, f, M)
   Data: FM is the feature model; f is the root of the
         feature model; M is a set of product configurations
   Result: true if feature f exists in some products and the
           tree having root f
2  begin
3      P = CreateP(FM, f)
4      flag = false
5      foreach m ∈ M do
6          if EvaluateP(P, FM, m) = true then
7              flag = true; break;
8          end
9      end
10     Return flag;
11 end
```

[1] http://javabdd.sourceforge.net/, Aug. 2016.

Algorithm 5. Check Alternative Propositional Formula

```
1  Function: CheckAlternativeP(FM, f, M)
   Data: FM is the feature model; ε is an alternative
         group; and M is a set of product configurations
   Result: true if features in ε do not contradict the
           alternative rule in all products
2  begin
3      flag = true
4      foreach m ∈ M do
5          counter = 0
6          foreach ff in f do
7              P = CreateP(FM, ff);
8              if EvaluateP(P, FM, m) = true then
9                  counter = counter + 1;
10             end
11         end
12         if counter > 1 or counter = 0 then
13             flag = false; break;
14         end
15     end
16     Return flag;
17 end
```

Algorithms 6 and 7 are used to check the inclusion and exclusion conditions between two features with respect to a set of product configurations M. Algorithm 6 checks the inclusion constraint between two features f and f' with regard to all input models (line 4). If feature f is present in a product configuration and feature f' is not present in this configuration (line 6), the result of the algorithm is false. Algorithm 7 checks the exclusion constraint between two features f and f' concerning the given product configurations M. This algorithm checks the exclusion constraint between two features f and f' in all product configurations (line 4). If two features f and f' are present together in a product, the exclusion constraint is violated and the false value is returned as the result of the algorithm (line 5).

Algorithm 6. Check Inclusion

```
1  Function: CheckInclude (f, f', M)
   Data: f and f' are the features to check inclusion f → f'
         on them; M is a set of product configurations
   Result: true if f → f'
2  begin
3      flag = true
4      foreach m ∈ M do
5          if f ∈ m and f' ∉ m then
6              flag = false; break
7          end
8      end
9      Return flag;
10 end
```

Algorithm 7. Check Exclusion

```
1  Function: CheckExclude (f, f', M)
   Data: f and f' are the features to check exclusion f ↔ f'
         on them; M is a set of product configurations
   Result: true if f ↔!f'
2  begin
3      flag = true
4      foreach m ∈ M do
5          if f ∈ m and f' ∈ m then
6              flag = false; break
7          end
8      end
9      Return flag;
10 end
```

Algorithm 8 investigates a set of product configurations M for the features which do not exist in the feature model FM (lines 4~10). This algorithm recurs on all features (line 5) of every given model (line 4) and checks the presence of that feature in the feature model primitive set.

Algorithm 8. List of Feature Opportunity

```
1  Function: CheckOpportunity (FM, M)
   Data: FM is the feature model; M is a set of product
         configurations
   Result: FL list of feature opportunity
2  begin
3      FL = null
4      foreach m ∈ M do
5          foreach f ∈ m do
6              if f ∉ FM.P and f ∉ FL then
7                  Add f to FL
8              end
9          end
10     end
11     Return FL;
12 end
```

Algorithm 9 is given a feature model, a start feature, and has the upcoming products' configuration as input. It searches in a sub-tree of the feature model (starting from feature r) for refactoring patterns. The algorithm calls itself recursively for each of the sub-trees (children) related to the current tree. These calls continue until a leaf feature is reached. If the algorithm finds an appropriate refactoring opportunity, it adds refactoring opportunity to list L. In the end, L contains the output of the algorithm. The method *sibilingOf*() in line 4, returns the list of all siblings of a particular feature in the feature model. The method *child*() (line 4) returns the child of a specific feature in the feature model. The reasoning Algorithm 9 recurs on the λ set (see the outermost foreach loop (line 3) in Algorithm 9).

Algorithm 9. Check Feature Model

```
1  Function: Reasoning1 (FM, r, M, M′)
   Data: FM is the feature model; r is the root of the feature model; M is a set of current products configuration;
         M′ is a set of upcoming products configuration
   Result: L as a list of refactoring opportunities
2  begin
3      foreach ((ε, FT) in FM.λ) and (ε ∩ child(r) ≠ null) do
4          s = sibilingOf(ε); ff = ε ∩ child(r)
5          if FT = "Mandatory" then
6              if CheckMandatoryP(FM, ff, M′) = false then
7                  Refactoring of 3.3.2 for feature ff.name add to list L
8                  if ff not in FM.P then
9                      Reasoning1(FM, ff, M, M′)
10                 end
11             end
12         end
13         if FT = "Optional" then
14             if CheckOptionalP(FM, ff, M ∪ M′) = false then
15                 Refactoring of 3.3.6 for feature ff.name add to list L
16                 if ff not in FM.P then
17                     Reasoning1(FM, ff, M, M′)
18                 end
19             else
20                 if CheckMandatoryP(FM, ff, M ∪ M′) = true then
21                     Refactoring of 3.3.3 for feature ff.name add to list L
22                 end
23             end
24         end
25         if FT = "Alternative" then
26             foreach f in ε do
27                 if CheckOptionalP(FM, f, M ∪ M′) = false then
28                     Refactoring of 3.3.4 for feature f.name add to list L
29                     if f not in FM.P then
30                         Reasoning1(FM, f, M, M′)
31                     end
32                 end
33             end
34             if CheckAlternativeP(FM, ε, M′) = false then
35                 Refactoring of 3.3.5 for feature f add to list L
36             end
37             foreach si in s do
38                 if CheckAlternativeP(FM, ε ∪ si, M′) = CheckOptionalP(FM, si, M′) = true then
39                     Refactoring of 3.3.8 for feature si add to list L
40                 end
41             end
42         end
43         if (FT = "OR") then
44             foreach f in ε do
45                 if CheckOptionalP(FM, f, M ∪ M′) = false then
46                     if f in FM.P then
47                         Refactoring of 3.3.6 for feature f.name add to list L
48                     end
49                 end
50             end
51             if CheckAlternativeP(FM, ε, M ∪ M′) = true then
52                 Refactoring of 3.3.7 for ε features add to list L
53             end
54         end
55     end
56 end
```

In case of *Mandatory* feature (in Algorithm 9, lines 5~12), the algorithm checks mandatory condition on the child feature *ff*. If the mandatory condition is violated, refactoring 3.3.2 is matched. If the mandatory condition is not violated, the reasoning algorithm begins on the *ff* feature recursively. The behavior of the algorithm for the *Optional* feature is similar (in Algorithm 9, lines 13~24).

In case of *Alternative* feature (in Algorithm 9, lines 25~42), the algorithm checks the optional condition on all members of the alternative group. If checking some optional conditions results in a false value (on a primitive feature), it means that the investigated feature is not present in any product. In this situation, the refactoring of 3.3.4 is matched. If the alternative condition is violated (e.g., two members of the alternative group are present in a product at the same time), the refactoring of 3.3.5 is matched. Finally, if there is a sibling which can be added to the alternative group, the refactoring pattern 3.3.8 is matched.

In the case of the OR feature group (in Algorithm 9, lines 43~54), if a member is not present in any product, it can be removed from the OR-group. In this situation, the refactoring pattern 3.3.6 is matched. If the alternative condition is matched for an OR-group, it can be converted to alternative group and refactoring of 3.3.7 is matched.

Algorithm 10 checks the integrity constraints of the feature model. This algorithm is not recursive in nature and cannot be integrated with Algorithm 9, and as a result, it is proposed as a separate algorithm.

In this algorithm (Algorithm 10), the refactoring of 3.3.1 is matched (line 4) if a feature that does not exist in the feature model is present in some product configurations. If an inclusion rule is violated in some product configurations, refactoring 3.3.9 is matched (line 8). Finally, if there is an exclusion constraint that is violated in some product configurations, refactoring 3.3.10 is matched (line 13).

Algorithm 10. Check Cross-Tree Constraints

```
1  Function: Reasoning2 (FM, M′)
   Data: FM is the feature model; M′ is set of
         upcoming products' configuration
   Result: L as a list of refactoring opportunities
2  begin
3      FL = CheckOpportunity(FM, M′)
4      if (FL ≠ null) then
5          Refactoring of 3.3.1 for feature list FL add to
           L
6      end
7      foreach ((f, f′) ∈ FM.φ) do
8          if CheckInclude(f, f′, M′) = false then
9              Refactoring of 3.3.9 for feature f and f′
               add to L
10         end
11     end
12     foreach ((f, f′) ∈ FM.χ) do
13         if CheckExclude(f, f′, M′) = false then
14             Refactoring of 3.3.10 for feature f and f′
               add to L
15         end
16     end
17 end
```

4 Case Study

One of the common approaches for evaluating software engineering methods is using them in real projects[30-31]. In this section, our proposed framework is investigated in a practical example. We investigate the applicability of our framework in a real-world SPL.

4.1 Introducing the Case Study

Medio is a project in the domain of health information. The project aims to facilitate the communication between doctors and patients, and to manage appointments, medical consumable materials, finance, and customer relations in a better way. The Medio SPL is designed based on the diversity of expertise and needs in this area as well as the commonalities between applications in the domain of health information.

Medio has a reference architecture that is maintained on an ongoing basis. In addition to the reference architecture (Fig.14), which will be discussed in Subsection 4.2, Medio has a feature model for managing the features in the SPL (Fig.15). This SPL currently has three products and one upcoming product. We survey these products in Subsection 4.3.

In this case study, we assess the Medio SPL in three milestones called refactoring points. Each refactoring point succeeds after a new version of the feature model. The Medio SPL is investigated to find refactoring opportunities in it. We used two teams in evaluating our framework named Medio team and external team.

1) *Medio Team.* This team uses our framework to find refactoring opportunities and to perform refactorings on the SPL. The members of this team are the main developers of the Medio SPL. In our case study, this team looks for refactoring opportunities in the specific refactoring points mentioned in Subsection 4.5.

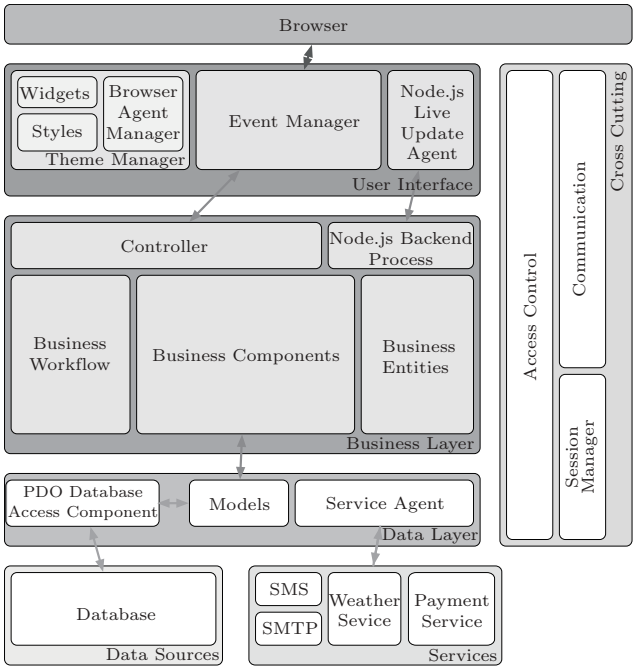

Fig.14. Medio SPL reference architecture version 1.

2) *External Team.* We formed a new external team in the Medio SPL, provided each of them with the artifacts we have in every refactoring point mentioned in Subsection 4.5, and asked them to suggest refactoring on the Medio SPL. This team does not use our framework to find and perform refactoring in the refactoring points. This team can survey every artifact of the Medio SPL including the feature model, current products' configuration, and reference architecture, as well as feature mapping and code. It is important to note that the external team only looks for refactoring opportunities in the specified refactoring points of Medio and does not maintain and develop the Medio SPL.

4.2 Medio Architecture

Fig.14 shows the general architecture of the Medio SPL version 1. Medio has a three-layer architecture consisting of user interface (UI), business layer, and data layer. An events manager is responsible for controlling the requests in the UI layer, while the controller component is responsible for managing the demands of the business layer. The business layer has a client-server style architecture. The controller component interprets the requests and sends them to the appropriate component with respect to the workflow. Communication with web-services and database resources is managed in the data layer. Besides the three layers, a cross-cutting layer is defined. This layer controls access and communication among the other three layers.

The feature mapping can map any feature in the feature model to several types of artifacts in the SPL. Fig.16 shows a mapping between the access control features in the Medio SPL and the reference architecture. Fig.17 indicates a mapping between features in the feature model and classes in the core assets of the SPL. In the end, Fig.18 shows a mapping between Medio access control feature and the code implementation of the SPL.

Every feature in the Medio feature model relates to some artifacts in the Medio core assets. The artifacts are spanned among the architecture level, the design level, and the code level. The relation between the artifacts in core assets and the features in the feature model can be direct, like the relation of the simple feature and the simple access class in Fig.17; or it can be indirect, like the relation of the simple feature and the access class in the access control component in Fig.17.

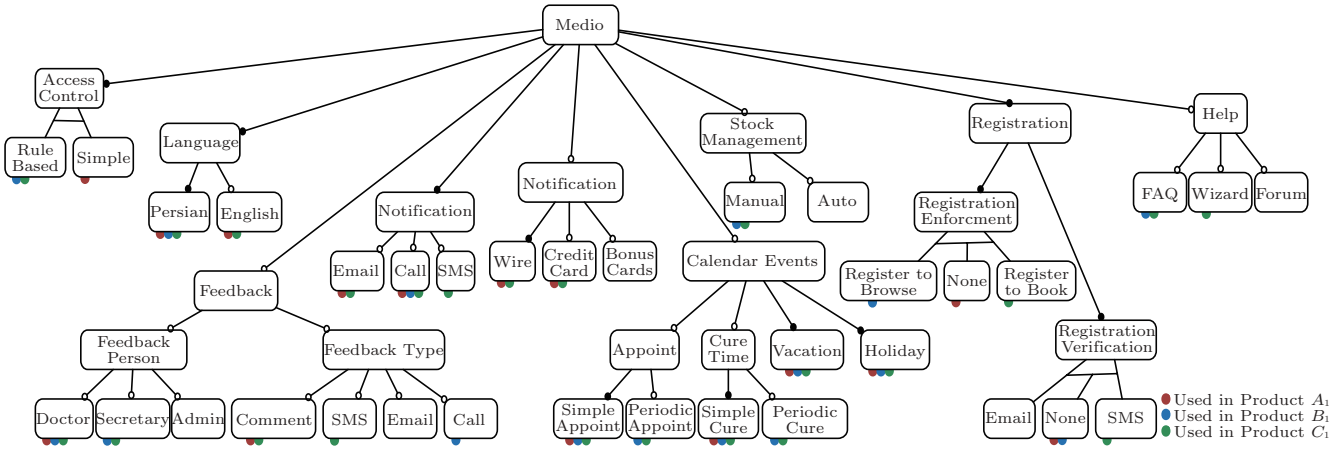

Fig.15. Medio feature model version 1. Admin: Administrator. Appoint: Appointment.

Fig.15 displays the Medio feature model which is investigated in this case study.

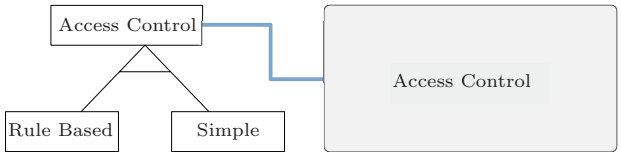

Fig.16. Example of 1:1 feature mapping in Medio SPL (from feature model to designed reference architecture).

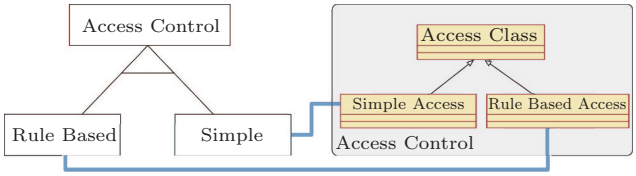

Fig.17. Another example of 1:1 feature mapping in Medio SPL (from feature model to designed classes).

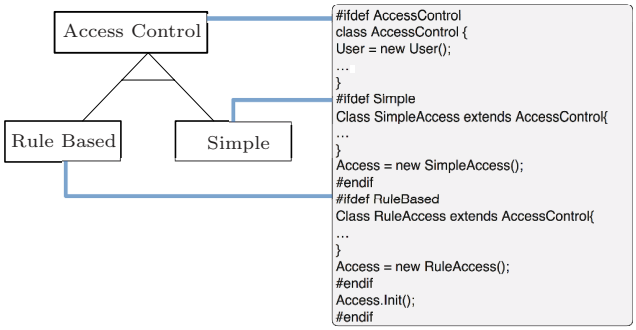

Fig.18. Example of $m:n$ feature mapping in Medio SPL (from feature model to code).

4.3 Products in the Medio SPL

Currently, four products are under development in the Medio SPL. We design each of them according to the needs of customers in this area. Medio clients can be placed into four categories.

1) *General Practitioner* (*GP*). The main use of Medio for this type of customer is managing appointments.

2) *Intranet Reserve System*. The customers of this product are the military and some clinics. They can manage appointments, pay the fees and manage the resources using this product. The doctors and the managers of the clinic can use this product offline. The other three products of the Medio SPL can be used through the Internet and on the customer servers, but this product can only be placed on the customer servers and can be used locally.

3) *Dentists*. The dentists' use of Medio is wider than that of GPs. They use Medio for setting simple and periodic appointments as well as managing the resources in the clinic.

4) *Clinics*. The most complete product of the Medio SPL is the clinic management product, which helps the user moderate a large number of activities done in a clinic.

To simplify the following discussion, we designate for every product in the Medio SPL an alphabet letter. The two products A and B are derived from the first version of the SPL. Product C is derived from the second version of the SPL, and finally, product D is an upcoming product scheduled for being developed in the near future.

4.3.1 *Product A: Product for GPs*

This product is a generic product suitable for most medical needs. Some of the features of the Medio SPL such as SMS and online appointment booking are not available for this product. Due to the assumption that patients are served on a spontaneous basis by GPs, online booking has not been implemented in this product. The configuration of product A_1 (the first version of product A) is as follows.

```
M_A1 = [SimpleAccessControl,
        EmailNotification, CallNotification,
        WirePayment, CreditCardPayment,
        CommentFeedback, DoctorFeedback,
        SimpleAppoint, SimpleCure,
        VacationEvent, HolidayEvent,
        NoRegisterEnforce,
        NoVerification, FaLang, EnLang].
```

4.3.2 *Product B: Intranet-Based Product*

This product controls the use of the Internet and data export, and has been developed for organizations requiring such services (such as the military and some government hospitals). All web-services and all features requiring the presence of web-services are disabled in this product. The users of this product use the product for financial purposes and for documenting information related to patients. The configuration of product B_1 is as follows.

```
M_B1 = [RuleBasedAccess, CallNotification,
        WirePayment, CallFeedback,
        DoctorFeedback, SecretaryFeedback,
        SimpleAppoint, PeriodicAppoint,
        SimpleCure, PeriodicCure,
        VacationEvent, HolidayEvent,
        ManualStock, RegisterToBrowse,
        NoVerification, FAQ, FaLang].
```

4.3.3 Product C: Product for Dentists

The main feature of this product is the doctor-patient interaction through the web and online booking. Some of the dentist's special needs such as periodic appointments are provided in this product. Due to the linguistic needs of patients, Persian and English languages are included in this product. Product C provides manual material management and purchase management. The configuration of product C_1 is as follows.

```
M_C1 = [RuleBasedAccess , EmailNotification ,
        CallNotification , SMSNotification ,
        WirePayment , CreditCardPayment ,
        CommentFeedback , SMSFeedback ,
        DoctorFeedback , SecretaryFeedback ,
        SimpleAppoint , PeriodicAppoint ,
        SimpleCure , PeriodicCure ,
        VacationEvent , HolidayEvent ,
        ManualStock , RegisterToBook ,
        SMSVerification , FAQ ,
        WizardHelp , FaLang , EnLang].
```

4.3.4 Product D: Clinic Management Product

This product is the most complete product of the Medio SPL. A large number of features of the feature model are present in this product. Product D includes all features of products A, B, and C. The aim of this product is to support activities of a clinic and help in its management. In addition to the conventional payment method in other products, the payment in product D can be done via bonus cards issued by organizations. The stock management in this product is automatic. It also uses email verification for user registration and has a forum to support users. The configuration of product D_1 is as follows.

```
M_D1 = [RuleBasedAccess , EmailNotification ,
        CallNotification , SMSNotification ,
        WirePayment , CreditCardPayment ,
        BonusPayment , CommentFeedback ,
        CallFeedback , SMSFeedback ,
        EmailFeedback , DoctorFeedback ,
        SecretaryFeedback , AdminFeedback ,
        SimpleAppoint , PeriodicAppoint ,
        SimpleCure , PeriodicCure ,
        VacationEvent , HolidayEvent ,
        ManualStock , AutoStock ,
        RegisterToBook , SMSVerification ,
        EmailVerification , FAQ ,
        WizardHelp , Forum , FaLang , EnLang].
```

4.4 Product Creation Policy

Medio SPL is a feature-oriented SPL designed to support the domain of health information. Each of the selected features is added to the product by putting the related artifacts to the selected features in this product. The related artifacts to each feature are specified by a feature mapping mechanism. For example, when a product uses the *RuleBasedAccess* feature in its configuration, the Medio team adds the related artifacts such as the *Rule Based Access* class, the *Access* class, and the code between *#ifdef* tags to the generated product. Diversification in the SPL is usually realized through such tactics as editing configuration files, inheritance, templates, and modification of the input parameters of the components. Because of this, the feature mapping rules may be in the form of editing a configuration file, cloning some files, executing some scripts on a database, or cloning-related classes in the product.

The feature mapping can be used to identify files and data tables related to each feature. When removing a feature from core assets and the feature model, the feature mapping can be used to analyze the core assets to find orphaned assets. If an artifact is not linked to a feature, the artifact can be marked as removable.

4.5 Development Timeline and Refactoring Points of the Medio SPL

Fig.19 shows the product development timeline of the Medio SPL. Products A_1 and B_1 are derived from the first version of the Medio SPL. At the refactoring point 1, two products have been developed using the Medio SPL and three products (C_1, $A_{1.1}$, and $B_{1.1}$) are scheduled to be released in the future. The refactoring at this point looks for refactoring opportunities based on these products.

The clinic management product is scheduled to be released using the Medio SPL (product D) after releasing $V_{1.1}$ of the Medio SPL.

Due to some problems at the refactoring point 2, product C_1 was not released. Two products have a new version at this point ($A_{1.1}$ and $B_{1.1}$). The configuration model of the C_1 and D_1 products was changed at this point.

Product C_1 was released before refactoring point 3. At the refactoring point 3, some changes were scheduled to be done on product C_1. At this point, the next versions of the products A_2 and $B_{1.3}$ are scheduled for release.

4.5.1 Medio SPL Version 1

The Medio reference architecture is as shown in Fig.14, and the feature model is as Fig.15. To save space, the mathematical model of the feature model is not mentioned here. *FM* denotes the feature model of Medio SPL.

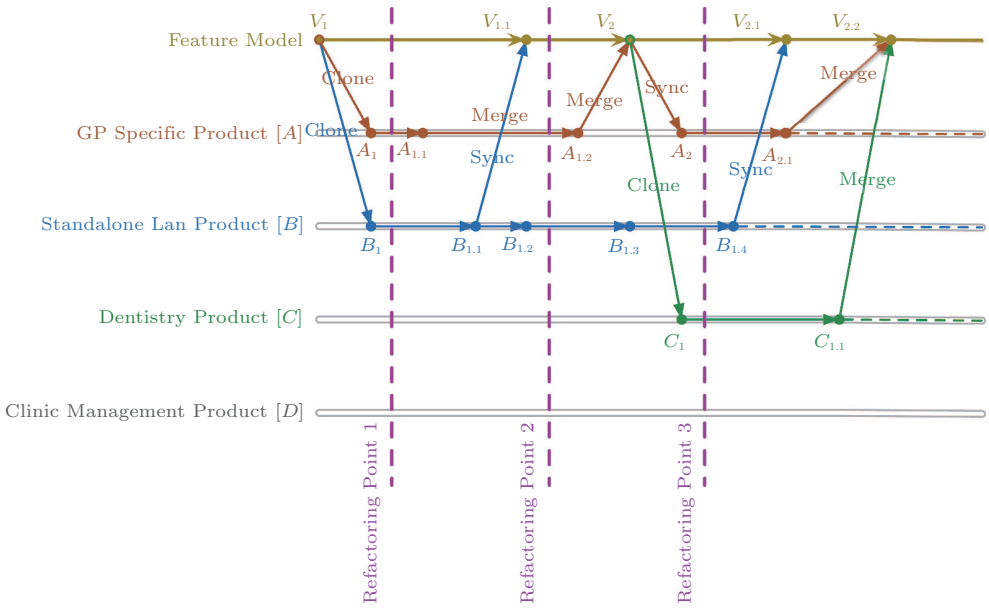

Fig.19. Development timeline of the products in Medio SPL.

4.5.2 *Refactoring Point 1*

The changes in the upcoming products' configuration in this refactoring point are as follows.

$$M_{A_{1.1}} = M_{A_1} \cup \{\texttt{RuleBasedAccess}, \texttt{PeriodicAppoint}\} \setminus \{\texttt{SimpleAccess}\},$$
$$M_{B_{1.1}} = M_{B_1} \cup \{\texttt{DBBackup}\}.$$

The complete list of the upcoming products is as follows.

$$M' = [M_{A_{1.1}}, M_{B_{1.1}}, M_{C_1}].$$

Because the *DBBackup* feature does not exist in the $FM.N$ list, it was added to the $FM.N'$ list. Fig.20 shows the Medio feature model in refactoring point 1. In this figure, we point out the refactoring opportunities by a different color (blue).

Medio Team Activities. The Medio team used our developed algorithm to find refactoring opportunities in the Medio SPL. The team started with executing the reasoning algorithm on the feature model and its product configurations.

The first execution of the reasoning algorithm on the feature model suggested the following refactoring opportunities:

1) deletion of the *SimpleAccess* feature from the alternative group;

2) conversion of the *PeriodicAppoint*, *CallNotification*, and *DoctorFeedback* features into Mandatory ones;

3) deletion of the *EmailFeedback*, *AdminFeedback*, *BonusPayment*, *AutoStock*, and *Forum* features;

4) deletion of the *EmailVerification* feature from the alternative group;

5) addition of the *DBBackup* feature to the feature model.

For each of the refactoring opportunities, the following decisions are considered.

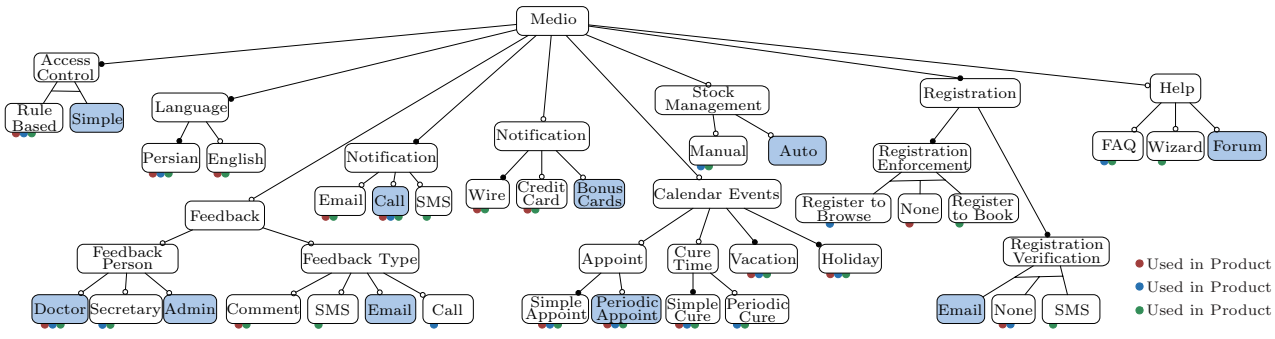

Fig.20. Medio feature model in refactoring point 1.

Deletion of the SimpleAccess Feature from the Alternative Group. The *SimpleAccess* feature is not used in any products of the Medio SPL. Our algorithm suggests converting the type of this feature from *Alternative* to *Optional*. The Medio team accepted the suggestion and converted the type of this feature. As a result of removing *SimpleAccess* from the alternative group, another member of that group which is named *RuleBasedAccess* becomes mandatory.

The feature model changes as follows.

```
FM.λ = FM.λ\{({SimpleAccess,RuleBasedAccess},
       Alternative)}∪{(RuleBasedAccess,
       Mandatory)}∪{(SimpleAccess, Optional)}.
```

Conversion of the PeriodicAppoint, CallNotification, and DoctorFeedback Features into Mandatory Ones. The Medio team accepted to convert the type of *PeriodicAppoint*, *CallNotification* and *DoctorFeedback* features to mandatory ones. This is because these features are used in all products of the Medio SPL. The feature model changes as follows.

```
FM.λ = FM.λ\{(PeriodicAppoint, Optional)}\{(
       CallNotification, Optional)}\{(
       DoctorFeedback, Optional)}∪{(
       PeriodicAppoint, Mandatory)}∪{(
       CallNotification, Mandatory)}∪{(
       DoctorFeedback, Mandatory)}.
```

Deletion of the EmailFeedback, AdminFeedback, BonusPayment, AutoStock, and Forum features. Our refactoring algorithm suggests removing *EmailFeedback*, *AdminFeedback*, *BonusPayment*, *AutoStock*, and *Forum* features from the feature model. This is because these features have not been used in any current and upcoming products. The Medio team declined to delete these features from the SPL. However, the Medio team marked these features as removable.

Deletion of the EmailVerification Feature from the Alternative Group. The Medio team declined to delete the *EmailVerification* feature from the alternative group and to change its type into optional one. The team marked this feature as removable for further investigation in the future.

Addition of the DBBackup Feature to the Feature Model. To perform the last refactoring opportunity, the business benefit of adding the feature to the core assets and the feature model should be computed. The COPLIMO[23] estimation approach (Appendix A.2) indicates the uneconomical nature of transferring the feature to the feature model.

The refactoring algorithm is run on the refactored version of the feature model until it cannot find any new refactoring opportunities in the feature model. This is because performing feature model refactoring can give rise to new refactoring opportunities. By executing the reasoning algorithm on the feature model, this new refactoring opportunity is found: deletion of the *SimpleAccess* feature from the feature model.

Deletion of the SimpleAccess Feature. The Medio team decided to remove the *SimpleAccess* feature from the feature model. This is because this feature is not used in any products of the Medio SPL and is not attractive for the Medio customer anymore. The artifacts related to *SimpleAccess* feature (such as the *Simple Access* class in Fig.17) are marked for the deletion from the SPL core assets.

Deletion of *SimpleAccess* has no impact on the reference architecture. This is because the *SimpleAccess* feature is not linked to any component of the reference architecture (Fig.16).

By deleting the *SimpleAccess* feature from the feature model, a dangling artifact in the class diagram appears (Fig.17). The simple access class is no longer related (directly or indirectly) to any features of the feature model. As a result, this class can be removed safely from core assets.

The deletion of the *SimpleAccess* feature affects the code behind the Medio SPL. Fig.18 shows the link between the *SimpleAccess* feature and the code to support this feature in the Medio SPL. After deleting the *SimpleAccess* feature, the code in the *#ifdef Simple* blocks should be removed.

Necessary changes in the *FM* feature model for carrying out this refactoring are as follows.

```
FM.N = FM.N\SimpleAccess,
FM.P = FM.P\SimpleAccess,
FM.DE = FM.DE\(AccessControl,SimpleAccess),
FM.λ = FM.λ\{(SimpleAccess, Optional)}.
```

Re-executing the refactoring algorithm on the feature model cannot locate any new refactoring opportunities on the feature model. As a result, the refactoring activities at this refactoring point are terminated.

External Team Activities. In refactoring point 1, the external team has access to all artifacts of the Medio SPL except the upcoming products' configurations. The current products' configurations of the Medio SPL in this refactoring point are M_{A_1} and M_{B_1}. In this refactoring point, the external team did not suggest any modifications on the Medio SPL.

4.5.3 Refactoring Point 2

The SPL version in this refactoring point is $V_{1.1}$. The SPL version $V_{1.1}$ is a small change on the Medio

SPL V_1 based on the changes in the products $A_{1.1}$ and $B_{1.1}$.

The *OnlineBooking* feature is initially present in all current products, without being present in the feature model. The addition of the *SMSBooking* feature means both *OnlineBooking* and *SMSBooking* were added to the feature model, sharing a "Booking" feature as their parent. The *Chat*, *DropIPRangeAccess*, *DiseaseDiagnosis* and *DrugConflictDiagnosis* features were added to N' list in this refactoring point.

The configuration model of SPL upcoming products is as follows.

$$\begin{aligned}
&M_{A_{1.2}} = M_{A_{1.1}} \cup \{\texttt{DiseaseDiagnosis}, \\
&\quad \texttt{OnlineBooking}, \texttt{DrugConflictDiagnosis}\}, \\
&M_{B_{1.2}} = M_{B_{1.1}} \cup \{\texttt{DropIPRangeAccess}, \\
&\quad \texttt{OnlineBooking}\}, \\
&M_{B_{1.3}} = M_{B_{1.2}} \cup \{\texttt{Chat}, \texttt{OnlineBooking}\}.
\end{aligned}$$

At this refactoring point, the configurations of the C_1 and D_1 products are changed, and some new features are added to their configurations.

$$\begin{aligned}
&M_{C_1} = M_{C_1} \cup \{\texttt{DBBackup}, \texttt{SMSBooking}, \\
&\quad \texttt{OnlineBooking}\}, \\
&M_{D_1} = M_{D_1} \cup \{\texttt{DBBackup}, \texttt{Map}, \texttt{OnlineBooking}\}, \\
&M' = [M_{A_{1.2}}, M_{B_{1.3}}, M_{C_1}, M_{D_1}].
\end{aligned}$$

Fig.21 shows the Medio feature model in the refactoring point 2. The effect of the refactorings which are performed in refactoring point 1 is shown in cyan in this figure.

Medio Team Activities. The Medio team ran the reasoning algorithms on the feature model and the configuration model of the current and upcoming products. The result of the execution of the algorithm is as follows:

1) addition of the *DBBackup* feature to the feature model;

2) addition of the *Chat*, *DropIPRangeAccess*, *Map*, *SMSBooking*, *DiseaseDiagnosis*, and *DrugConflictDiagnosis* features to the feature model.

Addition of the DBBackup Feature to the Feature Model: computing impact ratio (IR) for DBBackup.

To perform the first refactoring opportunity of the list, the impact ratio (IR) must be computed. The users can form different IR formulas based on the estimation approach they use. In this subsection, we use IR, which is formed based on the COPLIMO[23] estimation approach. Based on the formula in Appendix A.2.4, IR is computed as follows:

$$IR = \frac{Ruse \times Docu \times Rely + AA \times k'}{k'}.$$

Ruse is the cost of domain engineering and generalizing code for use in the Medio SPL. The *Ruse* value in the Medio project is 1.2. It means that for transferring a written code to the Medio SPL an extra 20% of writing should be done to make it reusable in the Medio SPL. This extra 20% includes the effort needed to change the SPL architecture and analyze the SPL domain to include the new feature in the SPL.

Docu is the cost of documenting the new feature in the Medio SPL. The *Docu* value equals 1.05 in the Medio project. It means that documenting a feature in the Medio SPL costs the team an extra 5% of the cost of developing this feature.

The magnitude of *Rely* shows the cost of tests and test scenarios, which have to be written and run in order to test new features in the SPL. The *Rely* value equals 1.2 in the Medio SPL. It means that writing new test scripts and test cases costs the Medio project 20% of the cost of developing the feature we want to write test script for.

The reuse policy in the Medio project is black-box. AA is 10% for this project, which means that reusing a feature costs 10% as much as writing this feature from scratch. Based on above values, the IR value for adding *DBBackup* to the feature model and core assets is:

$$IR = (1.2 \times 1.05 \times 1.2 + 0.1 \times 3)/3 = 0.604 < 1.$$

The IR value shows the economical nature of mov-

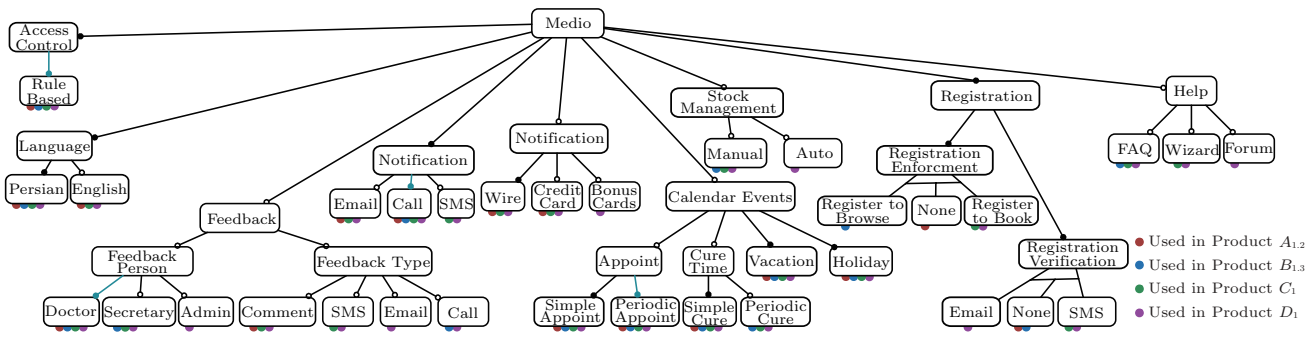

Fig.21. Medio feature model version 1.1 in refactoring point 2.

ing this feature to the feature model and the core assets. The Medio team decided to include the *DBBackup* feature in the feature model and the core assets (based on the business benefit of adding this feature to the core assets). For this reason, the feature mapping model should be modified to support the new feature and the relation of this feature to the artifacts in the core assets.

Changes Have Been Done on the Medio SPL to Add the DBBackup Feature to the Core Assets. Changes take place at several levels of abstractions. By adding *DBBackup* to the Medio SPL, the data layer of the reference architecture changes. A new component, named *Backup*, was added to the data layer. A link between the new feature and the *Backup* component is created. Fig.22 shows a partial view of the changes in the reference architecture and the feature model.

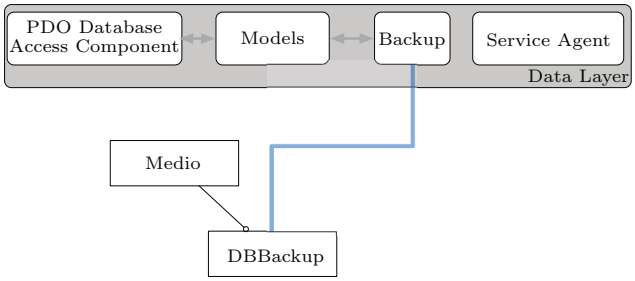

Fig.22. Changes in data layer of the Medio SPL reference architecture and the feature mapping artifact after adding *DBBackup* to the core assets.

The code written for use in a single product should be reviewed and rewritten so that it can be reused in the SPL products. This is a time-consuming and costly task. The result of this task is a *DBBackup* class which extends the *ActiveDBClass* in the Medio SPL. A link between this class and the *DBBackup* feature is established (Fig.23).

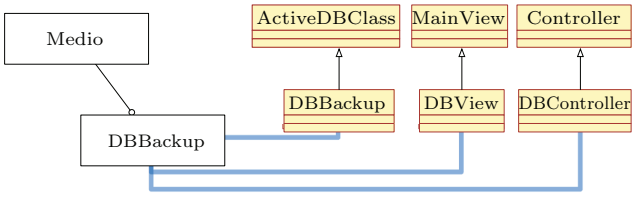

Fig.23. Changes in the class diagram and the feature mapping artifact of the Medio SPL after adding *DBBackup* to the core assets.

The refactoring process also affects the lowest level of the abstraction of the Medio SPL (code level). One of the effects is the addition of new code (including new classes) to the core assets (Fig.24). Another effect is changing the scripts which are used for creating code in the Medio SPL. Some changes should be done on the initial string of *RuleBasedAccess*. The *addmodule* function in *RuleBasedAccess* generates rules for accessing the *DBBackup* module. The menus of the Medio products are generated automatically from the list of modules.

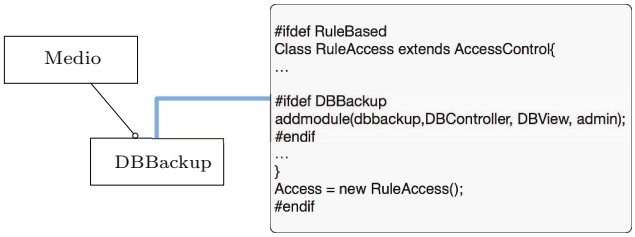

Fig.24. Changes in the code and the feature mapping artifact of the Medio SPL after adding *DBBackup* to the core assets.

The list of changes on the feature model after adding *DBBackup* to it is as follows:

$$
\begin{aligned}
&FM.N = FM.N \cup \{\texttt{DBBackup}\},\\
&FM.N' = FM.N' \setminus \{\texttt{DBBackup}\},\\
&FM.P = FM.P \cup \{\texttt{DBBackup}\},\\
&FM.DE = FM.DE \cup \{(\texttt{Medio}, \texttt{DBBackup})\},\\
&FM.\lambda = FM.\lambda \cup \{(\texttt{DBBackup}, \texttt{Optional})\}.
\end{aligned}
$$

Addition of the Chat, DropIPRangeAccess, SMSBooking, DiseaseDiagnosis, Map, and DrugConflictDiagnosis Features to the Feature Model. The *DropIPRangeAccess*, *DiseaseDiagnosis*, and *DrugConflictDiagnosis* features are presented in upcoming product $A_{1.2}$. The *Chat* feature is present in upcoming products $A_{1.2}$ and $B_{1.3}$. The *SMSBooking* feature is present in upcoming product C_1. And finally, the *Map* feature is present in upcoming product D_1.

However, the benefit of adding a feature to the feature model computed using the COPLIMO[23] approach shows the uneconomical nature of adding these features to the feature model and the core assets. These features should be rewritten from scratch in the upcoming products $A_{1.2}$ and $B_{1.3}$ by the Medio SPL developers. The Medio team decided not to move these features to the feature model at this refactoring point. The developer team used the code written for product $A_{1.1}$ with some changes in product $A_{1.2}$.

As mentioned before, we execute the reasoning algorithm again on the feature model until it fails to find new refactoring opportunities on the feature model. The reasoning algorithm cannot find new refactoring opportunities in the feature model in its new execution on the feature model. As a result, the refactoring activities in this refactoring point are over.

External Team Activities. In refactoring point 2, the external team has access to the Medio feature model version 1.1 (Fig.21), the reference architecture and the current products' configuration ($M_{A_{1.1}}$ and $M_{B_{1.2}}$). In this refactoring point, the external team suggested these modifications on the Medio SPL (based on analyzing the feature model, the reference architecture, and the current products' configuration): addition of the *DBBackup*, *SMSBooking*, and *Map* features to the feature model.

The external team suggested adding the *DBBackup*, *SMSBooking*, and *Map* features to the Medio SPL. The team suggested adding the *DBBackup* feature to the SPL core assets and the feature model because the team observes this feature in three products of the Medio SPL. The team also believes that the *SMSBooking* and *Map* features are attractive to the customers of the other products. As a result, the team suggested adding these features to the Medio SPL.

4.5.4 Refactoring Point 3

The second version of the SPL adds the *SMSBooking* and *Map* features to the feature model.

The feature model of the Medio SPL before performing the reasoning algorithm is as follows:

$FM.N = FM.N \cup \{\texttt{Map, Booking, SMSBooking, OnlineBooking}\},$
$FM.N' = FM.N' \cup \{\texttt{POSPrintService}\},$
$FM.P = FM.P \cup \{\texttt{Map, SMSBooking, OnlineBooking}\}$
$FM.DE = FM.DE \cup \{\texttt{(Medio, Map), (Medio, Booking), (Booking, SMSBooking), (Booking, OnlineBooking)}\},$
$FM.\lambda = FM.\lambda \cup \{\texttt{(Map,Optional), (Booking,Optional), (SMSBooking, Optional),(OnlineBooking, Mandatory)}\}.$

Upcoming products in this refactoring point are as follows:

$M_{A_2} = M_{A_{1.2}} \cup \{\texttt{DBBackup}\},$
$M_{A_{2.1}} = M_{A_2} \cup \{\texttt{PatientRating, PatientReview}\},$
$M_{B_{1.4}} = M_{B_{1.3}} \cup \{\texttt{POSPrintService}\} \setminus \{\texttt{OnlineBooking}\},$
$M_{C_{1.1}} = M_{C_1} \cup \{\texttt{ArLang}\},$
$M' = [M_{A_{2.1}}, M_{B_{1.4}}, M_{C_{1.1}}, M_{D_1}].$

Fig.25 shows the Medio feature model version 2 in this refactoring point with refactoring opportunities highlighted in it.

Medio Team Activities. The Medio team uses our algorithm to find the refactoring opportunities in this refactoring point. Our reasoning algorithm found these refactoring opportunities in the Medio feature model:

1) conversion of the *OnlineBooking* feature type to optional one;

2) conversion of the *DBBackup* feature type to mandatory one;

3) addition of the *POSPrintService*, *Chat*, *DropIPRangeAccess*, *DiseaseDiagnosis*, and *DrugConflictDiagnosis* features to the feature model.

Conversion of the OnlineBooking Feature Type to Optional one. The *OnlineBooking* feature is not used in all products of the Medio SPL. As a result, one can change the type of this feature into optional one.

$FM.\lambda = [FM.\lambda \setminus \{\texttt{(OnlineBooking, Mandatory)}\}] \cup \{\texttt{(OnlineBooking, Optional)}\}.$

Conversion of the DBBackup *Feature Type to Mandatory One.* The *DBBackup* feature is present in all current and upcoming products of the feature model. The Medio team decided to change the type of this feature based on its importance and needs of the Medio customers. As a result, the Medio feature model is changed as follows:

$FM.\lambda = [FM.\lambda \setminus \{\texttt{(DBBackup, Optional)}\}] \cup \{\texttt{(DBBackup, Mandatory)}\}.$

Addition of the POSPrintService, Chat, DropIPRangeAccess, DiseaseDiagnosis, and DrugConflictDiagnosis Features to the Feature Model. IR for *POSPrintService*, *Chat*, *DropIPRangeAccess*, *DiseaseDiagnosis*, and *DrugConflictDiagnosis* is computed similarly to the one done for the *DBBackup* feature.

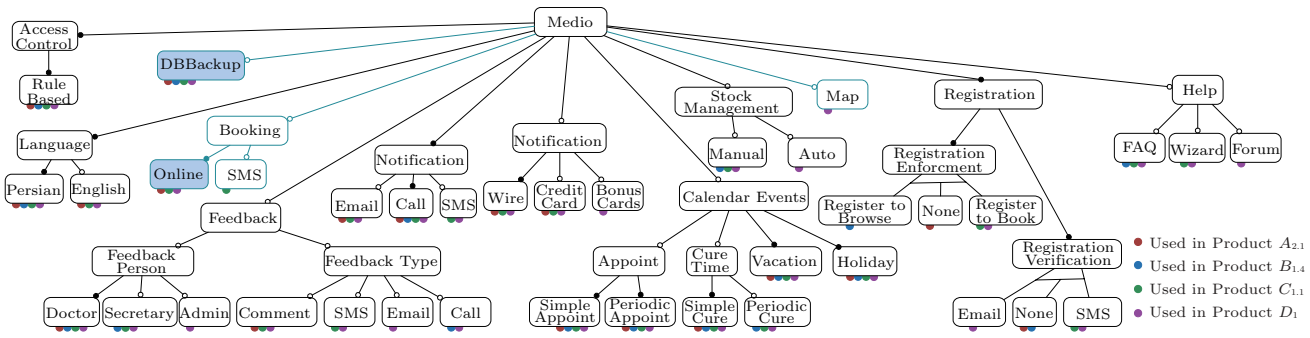

Fig.25. Medio feature model version 2 in refactoring point 3.

The value of IR for these features shows the uneconomical nature of transferring these features to the core assets and the feature model.

External Team Activities. In refactoring point 3, the external team has access to the Medio feature model version 2 (Fig.25), the reference architecture and the current products' configuration of Medio (M_{A_2}, $M_{B_{1.23}}$, and M_{C_1}).

Based on analyzing the Medio feature model and its related artifacts, the external team suggested this refactoring activity: addition of the *Chat* feature to the feature model.

The external team found the *Chat* feature attractive to the customers and as a result, suggested adding this feature to the feature model.

4.6 Comparing Performance of the Two Teams

We compare the performance of the two teams in finding and performing refactoring on the Medio SPL in Table 2. As one can see in Table 2, our framework can find refactoring opportunities in the feature model that the external team (the team that did not use it for performing refactoring) cannot find. In refactoring point 2, the external team suggests three refactorings on the Medio SPL. These three suggested refactorings are in the type of moving a feature to the feature model. The Medio team found these refactoring opportunities but refused to move them to the Medio SPL based on the computed IR for them. By comparing the performance of the two teams, we find following benefits in using our framework in refactoring the Medio SPL.

1) The team equipped with our framework can find small refactorings on the feature model that the human analysts cannot find. For example, the human analyst usually cannot find the refactoring of converting a mandatory feature into optional one in an SPL with a wide range of features.

2) The external team often finds the refactoring opportunities too late. This is because the human analyzer finds the refactoring opportunities based on the repetitions in the SPL. However, our framework can search for and suggest refactoring to the Medio team faster than an analyzer not powered by it.

3) While our framework considers the upcoming products of the SPL as well as the current products, the team which is not equipped with it usually looks at the current products and does not use the upcoming products to find refactoring opportunities. This is because the developers have no real sense of the products that are not implemented in the SPL.

5 Threats to Validity

Internal Validity. There are some internal threats to the validity of the refactoring performed on the feature model. It is possible that a new customer requires a feature, which has been removed recently from the SPL. This problem arises from a product line with poor foreseeing of the future or a product line including products with large domain differences. One of the other internal threats to validity in our framework is whether our framework is behavior preserving or not. Each refactoring has a guard (pre-condition) to check the correct condition in which the refactoring rule can be performed. The effects of the refactoring rule on the feature model can be anticipated. However, our framework does not have post-condition to check the actual correctness and behavior preservation of the feature model.

Construct Validity. There are some threats to the construct validity of the result of our framework. These threats can lead the user to an incorrect conclusion about refactoring on the feature model. One can essentially make two kinds of construct errors about feature model refactoring based on the estimation approach used to calculate the amount of benefit that comes from moving a feature to the feature model or removing a feature from the feature model.

• *False-Negative Conclusion.* It concludes that a feature should not be added to the SPL when, in fact, the feature should be included in the SPL. For example, we introduced the notion of IR for computing the economic benefit of moving a feature to the SPL. However, the IR parameters are computed experimentally. The lack of experiments in the SPL leads to supplying inaccurate parameters for computing IR. As a result, the value of IR may lead to an incorrect decision on not moving a feature to the feature model. In general, the

Table 2. Comparing Performance of the Two Teams in Finding Refactoring Opportunities

Team	Refactoring Point 1		Refactoring Point 2		Refactoring Point 3	
	Found	Applicable	Found	Applicable	Found	Applicable
Medio team	12	5	7	1	7	2
External team	0	0	3	3	1	1

false-negative conclusion can occur in every approach which uses experimental metrics.

• *False-Positive Conclusion.* It concludes that a feature should be added to the SPL when, in fact, the feature should not be included in the SPL. The inaccuracy of the estimation method may also lead to the incorrect moving of a feature to the feature model and the SPL core assets.

External Validity. The result of the framework can be validated by an external viewer of it. For example, in analyzing the case study in Section 4, an external observer can repeat the same results on the Medio SPL by using our framework. One of the other external threats to validity is the ability of the framework to go beyond the Medio SPL, which we analyzed. In this regard, we analyze the algorithms we use in our framework in Appendix A.1. The complexity of our framework is in the order of $O(n^6)$. We also survey the experimental evaluation of the framework in Appendix A.3. As one can see in Appendix A.3, our framework can be run on the SPL with a normal size feature model in a reasonable amount of time.

6 Discussion

In this paper, we proposed a feature model based framework for performing refactoring on the SPL. Our framework performs refactoring on the feature model and helps in synchronizing changes on it to the other artifacts of the SPL. In this way, other artifacts of the SPL can also sense the changes done on the feature model. For example, when we remove a feature from the feature model, we should assess other artifacts related to this feature and make appropriate changes on them to keep the consistency of the SPL.

We introduced algorithms needed to implement a tool for performing refactoring on feature models. Our reasoning algorithms are given the feature model along with the current and upcoming products' configuration to find refactoring opportunities in feature models.

In this paper, we used an extended version of the free feature model. The free feature model supports a majority of the methods used to develop SPLs[20]. As a result, we expect our framework to be usable in such methods with two considerations which we discuss in the following.

• The first consideration in using our framework is maintaining the upcoming products' configuration. Our framework uses the upcoming products to detect SPL refactoring opportunities. Upcoming products usually reflect the strategy of the SPL team in developing an SPL. As a consequence, by changing the SPL team strategy, the upcoming products' configuration may be modified. In this regard, the SPL developers should have a clear program to maintain upcoming products' configuration and keep it synchronized to the SPL strategies.

• Feature mapping is an appropriate approach to linking every feature to the related artifacts in the SPL. The second consideration in using our framework is to maintain a clear and explicit relation between features and other artifacts. We used feature mapping to find the artifacts which are affected by performing refactoring on the feature model at different levels of abstraction. Without a mechanism for feature mapping, the applications of the feature model become very limited.

Our approach is not usable in the SPLs without a clear development strategy or a mechanism for feature mapping.

Our aim in this framework is to modify the SPL artifacts in such a way that the overall functionality of the SPL (which appears in the SPL products) does not change. In our refactoring framework, the functionalities of the SPL are changed if the refactoring pattern turns the validity of a product configuration from valid into invalid. The conditions of our refactoring patterns ensure that none of the refactoring patterns can turn a valid product configuration into an invalid one.

One of the ways in which our framework may not preserve the functionalities of the SPL is in the situation where a refactoring pattern removes a feature from the feature model. By surveying the conditions of the pattern of removing a feature from a feature model in our framework, one can ensure that removing an unused feature from the feature model and the SPL does not change the functionalities of the SPL and its products.

We can use proof by contradiction to prove that our refactoring patterns do not alter the validity of the product configurations. For an example of using proof by contradiction, assume that converting an optional feature to mandatory one is not functionality preserving. As a result, at least one of the product configurations has become invalid by changing the type of a feature from optional to mandatory. The only way that a product configuration can be invalid by changing a feature type from optional to mandatory is when the product configuration does not contain the modified feature. This assumption contradicts the conditions of converting an optional feature to a mandatory one, because, for converting an optional feature to a mandatory one, the feature should be present in all current

and upcoming products of the SPL.

Many studies[32-37] use SAT solvers and binary decision diagram (BDD) solvers to analyze propositional formulae in the feature models. On the other hand, our work uses BDD solvers to analyze refactoring opportunities in the feature model. In this research, we performed feature model analysis by considering product configurations. Our framework first converts the feature model to a propositional formula and then solves the propositional formula using BDD solvers. Our framework generates strings consisting of Boolean values for analyzing the feature model. BDD solvers can check these Boolean strings in $O(n^2)$ time[22,38] whereas the SAT problems are NP-hard in this case[22]. Because of the simplicity of using propositional formulas and Boolean values in BDD solvers, we decided to use BDD solvers in our algorithms.

7 Related Work

One of the popular representations of the feature model is the free feature model[20]. Schobbens *et al.* provided a formal representation of the free feature model in their work[20]. They surveyed several SPL development methods such as FODA[18], FORM[39], FeatuRSEB[40], and PLUSS[41] and suggested a general model named "free feature model" to support them. Our feature model definition extends the free feature model by adding the definitions of the candidate features and the upcoming products. We used Schobbens *et al.*'s formalization[20] because their proposed feature model can handle the most important variability aspects of the SPL.

Feature model analysis is a tedious and error-prone task. In essence, manually analyzing feature models with more than a few dozen features is virtually impossible[22]. Several studies are done on analyzing feature models. Analysis performed on feature models can be any one of detecting empty models[18], product configuration verification[10,42-43], creating all possible models[44], core features analysis[42], enumerating all possible products, filtering[43], and anomaly detection[17], among others.

Anomaly detection is one of the analyses that can be performed on a feature model. The following is a list of three anomalies to hunt for:

• dead feature: a feature that cannot be present in any product[17];

• wrong optional feature: a feature that is present in all products[45];

• repetition: a feature model that contains several copies of a feature[45].

In our work, we provided support for exploring dead feature and wrong feature by investigating the products and feature models. Finding a repetition in a feature model in our work needs a sort of ontology-based analysis.

Some research related to the current work uses propositional formulas for analyzing feature models. Analysis consists of two phases in this group of studies:

1) transforming a feature model into propositional formulas;

2) analyzing the propositional formulas using off-the-shelf tools.

The term feature mapping comes from a tool written by Heidenreich *et al.*[46] Their tool (FeatureMapper) supports mapping features to the artifacts such as code and design model in SPL and facilitates creating concrete products from selected features. However, as documented by Seidl *et al.*[47], the granularity of feature mapping can vary considerably from "very fine" to "very coarse". We utilized the feature mapping idea in performing refactoring at three different levels of abstraction from architecture to design to code.

Seidl *et al.*[47] provided a mechanism for co-evolution design models and feature models. Their framework decreases the side-effects of the evolution of SPL in the feature mapping model. Their framework includes some ways to support moving, removing, copying, merging, and splitting the models and keeping the consistency between feature mapping and models. However, they do not consider changes that take place in the feature model in their work. We include this type of changes in our paper.

Alves *et al.*[6] discussed the inadequacy of traditional approaches for performing SPL refactoring. They suggested some refactorings which are applicable on the SPLs. They introduced some refactorings in feature models and assessed them through a case study[6]. The differences between our framework and the work of Alves *et al.*[6] lie in the way we find the refactoring opportunities. We used the product configurations along with the feature model to find the refactoring opportunities while they only used feature models to find them. The integrity constraints between the features of the feature model are not considered in the work of Alves *et al.*[6], while we used them to find refactoring opportunities in the feature model. The feature based refactoring is mentioned in [4]. We proposed a mechanism for variant-preserving refactoring, which preserves

the validity of the SPL.

Schulze *et al.*[4,48] proposed an approach to refactoring delta-oriented SPLs. They cataloged some refactorings applicable on delta-oriented SPLs. The delta-change done on the SPL is a basis for the definition of the refactoring catalogs in their work. While we used the product configurations and the changes performed on them to find the refactoring opportunities, they used the delta changes that come from the domain engineering activities. The problem with their refactoring framework is that their proposed framework cannot suggest some refactoring opportunities such as removing dead features or removing extra constraints in the feature model.

A formal representation for software product line and feature model was proposed by Borba *et al.*[49] Several types of refinement on SPL are proposed in their research. Some types of refactoring on the feature mapping model are introduced in their paper. The purpose of our framework differs from that of Borba *et al.*[49] While our main goal is to define and catalog several refactoring patterns on the feature models using a formal definition, the main purpose of Borba *et al.* is to provide a common notion for defining refactoring on the feature model. They did not provide a complete list of refactoring patterns and did not discuss how the refactoring patterns should be implemented. Fenske *et al.*[16] derived a taxonomy that distinguishes and relates reengineering activities on the SPL. They proposed definitions for the main branches of this taxonomy and finally classified a corpus of existing work.

Thum *et al.*[21] introduced several types of edits that can be done on a feature model. They categorized edits into four categories including 1) refactoring, 2) generalization, 3) specialization, and 4) arbitrary edit. They used propositional formulae to analyze the feature model and introduced an algorithm (which we also used in Appendix A.3) to evaluate the performance of the approach. Compared with our approach they did not catalog the refactoring patterns applicable to feature models and did not use product configurations for finding refactoring opportunities in the feature models.

Our work uses the feature model as a tool to find refactoring opportunities in the whole SPL. The advantage of our work over other refactoring approaches is that our model retains the consistency of the feature model with the feature mapping, product configurations, and reference architecture at different levels of abstractions. Our framework does the refactoring on a larger domain including the reference architecture, design and code, and uses feature mapping as a tool to track changes and find the locations of the changes.

Keeping the consistency of the feature model and other artifacts at different levels of abstraction is a challenge that the refactoring approaches face. Only Seidl *et al.*[47] and Borba *et al.*[49] considered this challenge in their refactoring definitions. We also tried to keep the consistency between the feature model and artifacts at different levels of abstraction by using feature mapping. For example, along with the deletion of a feature, our framework searches for the related artifacts at different levels of abstraction and suggests removing or keeping them to maintain consistency. However, our proposed framework does not ensure that consistency is maintained because the activities for keeping the consistency are not formally defined in it.

8 Conclusions and Future Work

Refactoring consists of small steps which, when accumulated, can effect large-scale changes in the SPL, and result in features being added to, eliminated from, or modified in the SPL. In this research, a feature model based framework for performing refactoring on the SPL artifacts, from code level to the reference architecture to the feature model, was developed. Upcoming products, which are indicative of SPL evolution strategies, were taken into account in performing refactoring, apart from current products.

While some refactorings, such as changing an optional feature into a mandatory one, only affect the feature model, some others, such as adding a feature to the feature model, have a wider effect on the SPL artifacts. As a result, SPL refactoring may cause inconsistencies between artifacts in SPL. Keeping the consistency among SPL artifacts at different levels of abstraction is complicated. One needs to know about the relation between features and other SPL artifacts to maintain the consistency among them. One of the appropriate ways to find the relationship between features and SPL artifacts is to use feature mapping.

Our framework provides a mechanism for finding refactoring opportunities in SPL and performing them on SPL. It uses the feature model to locate the refactoring opportunities in SPL. In this research, we provided several types of refactoring which are applicable on SPL. However, performing refactoring on SPL may give rise to some drawbacks for companies. It is possible that a feature that is removed from the feature model is required by some clients sometime later, or

the computed IR for a feature is false-positive, resulting in a feature being moved to the core assets in an uneconomical manner.

One interesting line of work is using artifacts in other levels of abstraction to find refactoring opportunities in the SPL. For example, at the code level, it is possible that one can extract new features from the code of the existing features' code, or find a dead block of code that causes some artifacts such as some features of the feature model and some components of the reference architecture to be removed from the SPL. However, refactoring is not limited to only one level and because of the tight relation of the artifacts in SPL, performing refactoring in one level can open new refactoring opportunities in other levels.

One of the approaches for performing refactoring on SPL and the feature model is to use the model-to-model (M2M) transformation languages. M2M transformation languages have been successfully used to perform refactoring on models. Wimmer *et al.*[50] introduced a catalog of refactorings for M2M transformations in their research. Using M2M transformations to perform refactoring is a new trend in this area. Utilizing M2M transformations to perform refactoring is another line of research, which currently is underway by the authors of this paper.

Finally, one future work is to identify opportunities for optimization of the approach and also evaluating scalability. Evaluating the framework in the context of another existing product line can be another future work of this paper.

References

[1] Tseng M M, Hu S J. Mass customization. In *CIRP Encyclopedia of Production Engineering*, Laperrière L, Reinhart G (eds.), Springer Berlin Heidelberg, 2014, pp.836-843.

[2] van der Linden F J, Schmid K, Rommes E. Software Product Lines in Action. Springer-Verlag Berlin Heidelberg, 2007.

[3] Clements P C, Northrop L. Software Product Lines: Practices and Patterns (3rd edition). Addison-Wesley Professional, 2001.

[4] Schulze S, Thüm T, Kuhlemann M, Saake G. Variant-preserving refactoring in feature-oriented software product lines. In *Proc. the 6th Int. Workshop on Variability Modeling of Software-Intensive Systems*, Jan. 2012, pp.73-81.

[5] Fowler M, Brant J, Opdyke W *et al.* Refactoring: Improving the Design of Existing Code. Pearson Education India, 2009.

[6] Alves V, Gheyi R, Massoni T, Kulesza U, Borba P, Lucena C. Refactoring product lines. In *Proc. the 5th Int. Conf. Generative Programming and Component Engineering*, Oct. 2006, pp.201-210.

[7] Zimmermann O. Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, 2015, 32(2): 26-29.

[8] Krueger C. Easing the transition to software mass customization. In *Lecture Notes in Computer Science 2290*, van der Linden F (ed.), Springer Berlin Heidelberg, 2002, pp.282-293.

[9] Pohl K, Böckle G, van der Linden F. Software Product Line Engineering. Springer-Verlag Berlin Heidelberg, 2005.

[10] Batory D. Feature models, grammars, and propositional formulas. In *Proc. the 9th SPLC*, Sept. 2005, pp.7-20.

[11] Pohl K, Böckle G, Linden F J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer Science & Business Media, 2005.

[12] Coleman D, Ash D, Lowther B, Oman P. Using metrics to evaluate software system maintainability. *Computer*, 1994, 27(8): 44-49.

[13] Guimaraes T. Managing application program maintenance expenditures. *Communications of the ACM*, 1983, 26(10): 739-746.

[14] Borba P. An introduction to software product line refactoring. In *Lecture Notes in Computer Science 6491*, Fernandes J M, Lammel R, Visser J, Saraiva J (eds.), Springer Berlin Heidelberg, 2011, pp.1-6.

[15] Gheyi R, Massoni T, Borba P. Automatically checking feature model refactorings. *J. UCS*, 2011, 17(5): 684-711.

[16] Fenske W, Thüm T, Saake G. A taxonomy of software product line reengineering. In *Proc. the 8th Int. Workshop on Variability Modelling of Software-Intensive Systems*, Jan. 2014, Article No. 4.

[17] Hemakumar A. Finding contradictions in feature models. In *Proc. the 12th Int. Software Product Line Conf.* (*SPLC*), Sept. 2008, pp.183-190.

[18] Kang K C, Cohen S G, Hess J A, Novak W E, Peterson A S. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[19] Lopez-Herrejon R E, Batory D. A standard problem for evaluating product-line methodologies. In *Proc. the Int. Conf. Generative and Component-Based Software Engineering*, Sept. 2001, pp.10-24.

[20] Schobbens P, Heymans P, Trigaux J C. Feature diagrams: A survey and a formal semantics. In *Proc. the 14th IEEE Int. Conf. Requirements Engineering*, Sept. 2006, pp.139-148.

[21] Thüm T, Kästner C, Erdweg S, Siegmund N. Abstract features in feature modeling. In *Proc. the 15th Int. Software Product Line Conf.* (*SPLC*), Aug. 2011, pp.191-200.

[22] Batory D, Benavides D, Ruiz-Cortes A. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 2006, 49(12): 45-47.

[23] In H P, Baik J, Kim S, Yang Y, Boehm B. A quality-based cost estimation model for the product line life cycle. *Communications of the ACM*, 2006, 49(12): 85-88.

[24] Mendonça M, Cowan D, Malyk W, Oliveira T. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *Journal of Software*, 2008, 3(2): 69-82.

[25] Knoop J, Rüthing O, Steffen B. Partial dead code elimination. *SIGPLAN Not.*, 1994, 29(6): 147-158.

[26] Kruchten P, Nord R L, Ozkaya I. Technical debt: From metaphor to theory and practice. *IEEE Software*, 2012, 29(6): 18-21.

[27] Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K. An exploratory study of cloning in industrial software product lines. In *Proc. the 17th European Conf. Software Maintenance and Reengineering* (*CSMR*), Mar. 2013, pp.25-34.

[28] Benavides D, Segura S, Trinidad P, Ruiz-Cortés A. FAMA: Tooling a framework for the automated analysis of feature models. In *Proc. the 1st Int. Workshop on Variability Modelling of Software Intensive Systems*, Jan. 2007.

[29] Mendonça M, Branco M, Cowan D. SPLOT: Software product lines online tools. In *Proc. the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications*, Oct. 2009, pp.761-762.

[30] Easterbrook S, Singer J, Storey M A, Damian D. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*, Shull F, Singer J, Siøberg D (eds.), Springer London, 2008, pp.285-311.

[31] Yin R. Case Study Research: Design and Methods. SAGE Publications, 2009.

[32] Batory D. A tutorial on feature oriented programming and the ahead tool suite. In *Lecture Notes in Computer Science 4143*, Lämmel R, Saraiva J, Visser J (eds.), Springer Berlin Heidelberg, 2006, pp.3-5.

[33] Benavides D, Segura S, Trinidad P, Ruiz-Cortès A. A first step towards a framework for the automated analysis of feature models. In *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms*, Aug. 2006, pp.39-47.

[34] Segura S. Automated analysis of feature models using atomic sets. In *Proc. the 12th Int. Software Product Line Conf.* (*SPLC*), Sept. 2008, pp.201-207.

[35] Thüm T, Batory D, Kastner C. Reasoning about edits to feature models. In *Proc. the 31st IEEE Int. Conf. Software Engineering* (*ICSE*), May 2009, pp.254-264.

[36] Mendonça M, Wasowski A, Czarnecki K. SATbased analysis of feature models is easy. In *Proc. the 13th Int. Software Product Line Conf.* (*SPLC*), Aug. 2009, pp.231-240.

[37] Czarnecki K, Wasowski A. Feature diagrams and logics: There and back again. In *Proc. the 11th Int. Conf. Software Product Line Conf.* (*SPLC*), Sept. 2007, pp.23-34.

[38] Fontoura M, Sadanandan S, Shanmugasundaram J, Vassilvitskii S, Vee E, Venkatesan S, Zien J. Efficiently evaluating complex Boolean expressions. In *Proc. the 2010 ACM SIGMOD Int. Conf. Management of Data*, Jun. 2010, pp.3-14.

[39] Kang K C, Kim S, Lee J, Kim K, Shin E, Huh M. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 1998, 5(1): 143-168.

[40] Griss M L, Favaro J, d'Alessandro M. Integrating feature modeling with the RSEB. In *Proc. the 5th Int. Conf. Software Reuse*, Jun. 1998, pp.76-85.

[41] Eriksson M, Börstler J, Borg K. The PLUSS approach-domain modeling with features, use cases and use case realizations. In *Lecture Notes in Computer Science 3714*, Obbink H, Pohl K (eds.), Springer Berlin Heidelberg, 2005, pp.33-44.

[42] Peña J, Hinchey M G, Ruiz-Cortés A, Trinidad P. Building the core architecture of a NASA multiagent system product line. In *Lecture Notes in Computer Science 4405*, Padgham L, Iambonell F (eds.), Springer Berlin Heidelberg, 2006, pp.208-224.

[43] Czarnecki K, Kim C H P. Cardinality-based feature modeling and constraints: A progress report. In *Proc. Int. Workshop on Software Factories*, Oct. 2005, pp.16-20.

[44] Trinidad P, Benavides D, Durán A, Ruiz-Cortés A, Toro M. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 2008, 81(6): 883-896.

[45] Von Der Maβen T, Lichter H. Deficiencies in feature models. In *Proc. the Workshop on Software Variability Management for Product Derivation — Towards Tool Support*, Aug. 30-Sept. 2, 2004, pp.59-62.

[46] Heidenreich F, Kopcsek J, Wende C. FeatureMapper: Mapping features to models. In *Proc. the 30th Int. Conf. Software Engineering*, May 2008, pp.943-944.

[47] Seidl C, Heidenreich F, Aβmann U. Co-evolution of models and feature mapping in software product lines. In *Proc. the 16th Int. Conf. Software Product Line Conf.* (*SPLC*), Sept. 2012, pp.76-85.

[48] Schulze S, Richers O, Schaefer I. Refactoring deltaoriented software product lines. In *Proc. the 12th Annual Int. Conf. Aspect-Oriented Software Development*, Mar. 2013, pp.73-84.

[49] Borba P, Teixeira L, Gheyi R. A theory of software product line refinement. Theoretical Computer Science, 2012, 455: 2-30.

[50] Wimmer M, Perez S M, Jouault F, Cabot J. A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 2012, 11(2): 2:1-2:40.

[51] Aspvall B, Plass M F, Tarjan R E. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 1979, 8(3): 121-123.

Mohammad Tanhaei is a Ph.D. candidate at Sharif University of Technology, Tehran. He received his B.S. degree in computer engineering from Isfahan University, Isfahan, in 2008, and then received his M.S. degree in computer engineering from Sharif University of Technology, Tehran, in 2010. He has worked in Software Engineering Lab under supervision of an associate professor, Jafar Habibi since 2008. Mohammad's research interests are mainly in the areas of software architecture, software product line, and software engineering.

Jafar Habibi received his B.S. degree in computer engineering from the Supreme School of Computer in 1980, his M.S. degree in industrial engineering from Tarbiat Modares University, Tehran, in 1988 and his Ph.D. degree in computer engineering from Manchester University, Manchester, in 1998. Currently he is an associate professor in the Department of Computer Engineering of Sharif University of Technology. He is the supervisor of Sharif's RoboCup Simulation Group. His research interests are mainly in the areas of computer engineering, simulation systems, MIS, DSS and evaluation of computer systems performance.

Seyed-Hassan Mirian-Hosseinabadi received his B.S. degree in computer engineering from Shahid Beheshti University, Tehran, in 1984, his M.S. degree in computer engineering from Sharif University of Technology, Tehran, in 1987 and his Ph.D. degree in computer engineering from The University of Essex, Colchester, in 1997. At present, he is an associate professor in the Department of Computer Engineering of Sharif University of Technology, Tehran. His research interests are mainly software engineering, in particular, formal methods, constructive mathematics, theoretical aspect of computer science, database design and compiler construction.

Appendix

A.1 Computational Complexity of Algorithms

Time complexity is the primary measure to bear in mind when one engages in algorithm design. In most cases, an exponential running time means very limited applicability for an algorithm. When a large time complexity in the worst case seems inevitable for any algorithm for a particular problem, one has to relax the requirement for the worst-case optimality of the designed algorithm. Heuristic, approximation, and randomized algorithms could enter the frame in these scenarios (though their use is by no means limited to these cases).

We calculate the complexity of the reasoning algorithm and show that this algorithm is reasonably efficient. In the following, we shall use m to denote the maximum tree degree, n to indicate the number of features in the feature model, k to show the number of current and upcoming products, and p to denote the number of primitive features.

A.1.1 Complexity of CreateP Algorithm

In this algorithm, the result of the execution on the sub-trees is kept in memory and used later for computing the propositional formulas of the parent. The worst case of the algorithm occurs in the case of the alternative group. In the alternative case, the algorithm makes all mutual combinations of the children. The number of mutual combinations of children is $O(m^2)$. The number of iterations of the algorithm is $n-p$ (the number of recursions in the algorithm). Consequently, the complexity of this situation is $O((n-p)m^2) = O(nm^2)$.

A.1.2 Complexity of EvaluateP Algorithm

The computational complexity of this algorithm is the time it takes to find the presence or the absence of a feature in a configuration model ($\log n$) multiplied by the number of propositions in the P string, which is $O(n^2)$ in the worse case. The JavaBDD method on true and false values can be implemented by an algorithm having linear complexity (with regard to the size of the string)[51]. In this case, the complexity of JavaBDD equals the maximum size of P, which is $O(n^2)$. In general, the complexity of the algorithm is $O(n^2 \log n + n^2) = O(n^2 \log n)$.

A.1.3 Complexity of CheckMandatoryP and CheckOptionalP Algorithms

In these algorithms, P is computed just once. The complexity of the foreach block is the multiplication of k by *EvaluateP* complexity, or $O(kn^2 \log n)$. The complexity in general (assuming a constant number of products) is $O((n-p)m^2 + kn^2 \log n) = O(nm^2 + n^2 \log n)$.

A.1.4 Complexity of CheckAlternativeP Algorithm

The worst case in this algorithm is when ε contains all features in the feature tree. In this case, the complexity of the algorithm is the complexity of *CreateP* algorithm plus the complexity of *EvaluateP* algorithm multiplied by the number of iterations in the two nested foreach loops. This makes the complexity of the algorithm to be $O(k \times m \times (nm^2 + n^2 \log n)) = O(nm^3 + mn^2 \log n)$.

A.1.5 Complexity of CheckExclude and CheckInclude Algorithms

The complexity of these algorithms equals the number of products in SPL, or k. Assuming a constant number of products, the complexity of these algorithms is $O(1)$.

A.1.6 Complexity of CheckOpportunity and NotExisitNode Algorithms

The complexity of these algorithms is the number of products in SPL or k, multiplied by the number of features in the feature model or n. Thus, the complexity of these algorithms is $O(kn) = O(n)$.

A.1.7 Complexity of Reasoning1 Algorithm

The foreach loop in this algorithm is executed n times (recursively). In each iteration, one of the if conditions is true. The complexity of the algorithm is the number of features multiplied by the maximum complexity in the if conditions.

The complexity of checking the mandatory (line 5) and the optional (line 13) feature types in the worst case is $O(nm^2 + n^2 \log n)$.

The complexity of checking alternative group (line 25) is the number of iterations in the foreach loop (line 26) which is m in the worst case. The complexity of *CheckOptionalP* is $O(nm^2 + n^2 \log n)$. Thus, the complexity of the foreach loop in line 26 is $O(nm^3 + mn^2 \log n)$. The complexity of if in line 34 is $O(nm^3 + mn^2 \log n)$. And finally the complexity of the foreach in line 37 is the number of siblings of a feature or m multiplied by the complexity of the *CheckAlternativeP* and *CheckOptionalP* methods, which is $O(m((nm^3 + mn^2 \log n) + (nm^2 + n^2 \log n))) = O(nm^4 + m^2 n^2 \log n)$. The overall complexity of checking the alternative-group and OR-group (line 25) is $O(nm^4 + m^2 n^2 \log n)$.

The complexity of checking the OR-group (line 43) is the addition of the complexities of the foreach in line 44 and the complexity of if in line 51. Their complexity is the number of iterations of the foreach loop which is m multiplied by the complexity of the CheckOptionalP method which is $O(nm^2 + n^2 \log n)$. The overall complexity of these foreach loops is $O(nm^3 + mn^2 \log n)$. The complexity of if (line 51) is $O(nm^3 + mn^2 \log n)$. Thus the overall complexity of the OR-group checking is $O(nm^3 + mn^2 \log n)$.

It can be readily checked that the worst-case running time of the *Reasoning*1 algorithm is $O(n(nm^4 + m^2 n^2 \log n)) = O(m^4 n^2 + m^2 n^3 \log n)$. If m is a constant value, the complexity of the algorithm is $O(n^3 \log n)$. If m is in the order of $\sqrt{n}$, the overall complexity is $O(n^4 \log n)$. Finally, if $m = n - 1$, the complexity amounts to $O(n^6)$.

A.1.8 Complexity of Reasoning2 Algorithm

The maximum number of cross-tree constraints in the feature model is $O(n^2)$. As a result, the complexity of the *NotExisitNode* and *CheckOpportunity* methods is $O(n^2)$. It follows that the complexity of this algorithm is $O(n^2)$.

A.1.9 Total Computational Complexity of Algorithms

The total complexity of the reasoning algorithm is the complexity of the *Reasoning*1 algorithm plus the complexity of the *Reasoning*2 algorithm. Thus, the total computational complexity of the algorithms in the worst case is $O(n^6)$.

A.2 SPL Development Cost Estimation Using COPLIMO Method

COPLIMO is one of the approaches used to estimate the cost of developing an SPL[23]. The three primary factors considered in this method to determine the cost of producing an SPL are as follows[23]:

1) $RCWR$ (relative cost of write for reuse): the cost of writing for reuse. It can be stated as the product of three factors:

$RCWR = Ruse \times Docu \times Rely$.

a) *Ruse*: the cost of domain engineering plus the cost of designing the SPL architecture and generalizing the code for reuse.

b) *Docu*: the cost of documenting the code written for reuse.

c) *Rely*: the cost of testing and evaluating changed code.

2) AA (assessment and assimilation): the cost of the black-box reuse of a component in the reference architecture. In the black-box reuse, there is no need to modify the internal code of the components. They can be reused by changing their input parameters.

3) AAM (adaptation adjustment modifier): the cost of the white-box reuse of a component in the SPL. In white-box reuse, the internal code of the components may be changed in the final products. To calculate AAM, the following parameters must be calculated[23]:

a) DM: percentage of change in designing for reuse;

b) CM: percentage of change in writing code for reuse;

c) IM: amount of effort used to integrate the components into the final product;

d) SU: developers component understandability level;

e) *UNFM*: developers component unfamiliarity level.

$$AAM = \begin{cases} \frac{AA+AAF(1+(0.02\times SU\times UNFM)}{100}, & \text{if } AAF \leqslant 50, \\ \frac{AA+AAF+(SU\times UNFM)}{100}, & \text{otherwise.} \end{cases}$$

$$AAF = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM).$$

A.2.1 Estimating the Cost of Transferring a Feature from Product to the Core Assets

In estimating the cost of transferring a feature from a product to the core assets (the feature model), one must plan methods and mechanisms of reuse. To determine the cost of carrying a feature from a product to the core assets (feature model), first and foremost the reuse methods and mechanisms should be taken into account. The written component for reuse can be reused as either white-box or black-box. A component may be designed to be reused as either white-box or black-box. In the black-box mechanism, efforts to identify all various configurations of the feature and their generalization is much bolder compared with the white-box approach. The black-box mechanism requires a much more pronounced effort to determine different configurations of the feature and their generalizations compared with the white-box mechanism. White-box reuse is more expensive at the time of reuse than the black-box reuse.

Suppose that we have a feature with the following conditions:

$$f \in m_{p'_1} \cap \cdots \cap m_{p'_k}.$$

That is, the feature is present in the upcoming products $1, ..., k$. We first calculate the cost of developing products with or without transferring the feature to the feature model, and then we develop a metric named IR assisting in deciding whether or not to transfer the feature to the feature model.

A.2.2 Costs of Not Transferring a Feature to the Feature Model

Assume that the size (in function points) of feature f is equal to *Size*, and the cost of development of each function point is equal to C. In this case, the cost of the development without transferring the feature to the feature model is equal to:

$$Cost\ of\ Development = Size \times C \times k'.$$

A.2.3 Costs of Transferring a Feature to the Feature Model

If the feature has been developed in the black-box approach, the development cost for using in the upcoming products of the SPL is as follows:

$$\begin{aligned} &Cost\ of\ Development \\ &= Size \times C \times (Ruse \times Docu \times Rely) + \\ &Size \times C \times AA \times k'. \end{aligned}$$

If the feature is developed with a white-box approach, AAM must be calculated and included in the above formula instead of AA.

A.2.4 IR Calculation

The SPL refactoring has a basic and important condition called "return on investment". If the refactoring cost is much greater than its profits, the refactoring is not favorable for the project managers and stakeholders. IR is defined as the development costs in moving the feature to the feature model divided by the cost of developing products without transferring the feature to the feature model. In case of black-box reuse, the amount of IR is equal to:

$$\begin{aligned} &IR \\ &= \frac{Size \times C \times (Ruse \times Docu \times Rely) + Size \times C \times AA \times k'}{Size \times C \times k'} \\ &= \frac{Ruse \times Docu \times Rely + AA \times k'}{k'}. \end{aligned}$$

IR value less than 1 shows economical nature of transferring a feature to the feature model.

A.3 Experimental Evaluation

A.3.1 Experimental Environment

For experimental evaluation of the algorithm, a MacBook Pro computer, with an Intel® Core i5 4288U 2.6 GHz CPU, 8 gigabytes of RAM running MacOS X El Capitan was used. The code was written in Java.

A.3.2 Random Feature Tree Creation Algorithm

We use the algorithm discussed in [35] to create the random feature model. The algorithm starts with a given feature r as the root of the tree. In every iteration of the algorithm, a child is added to an existing feature. The type of the added feature is determined randomly from alternative-group, OR-group, mandatory, and optional[35]. The random tree creation procedure is called recursively on child features of the created tree until the number of features in the feature tree reaches the required number P.

Some cross-tree constraints between primitive features are created in the feature model. The number of these cross-tree constraints is 5% of the number of features in the tree.

In the end, three valid current products with $\log p$ features, and two valid upcoming products with $\log p$ features are created.

A.3.3 Random Edit Algorithm

The random edit algorithm is given a feature model, a set of product configurations, and a number of desired changes on the feature model C as input, and produces C changes in the feature model. The edits are in the form of converting optional to mandatory, mandatory to optional, alternative to OR-group, and OR-group to alternative-group[35]. The probability of each type of edit is the same.

A.3.4 Experimental Result

Fig.A1 shows the execution time of the reasoning algorithm in the feature models with 10 up to 10 000 features and five random edits in these feature models. This figure indicates that with the enlargement of a feature model and a fixed number of edits in the feature model, the runtime growth rate decreases. This can be attributed to the fact that a subtree is increasingly more likely to be pruned as one moves down from the root (this is because the chance of finding an error in a more deeply rooted subtree is smaller, and consequently, the subtree is more likely to be pruned away).

We have done two simulations to find the run-time complexity of our algorithms. The first keeps the number of edits fixed and varies the size of the feature model (Fig.A1), and the second explores the effect of applying a variable number of edits to a feature model with a fixed number of features (Fig.A2).

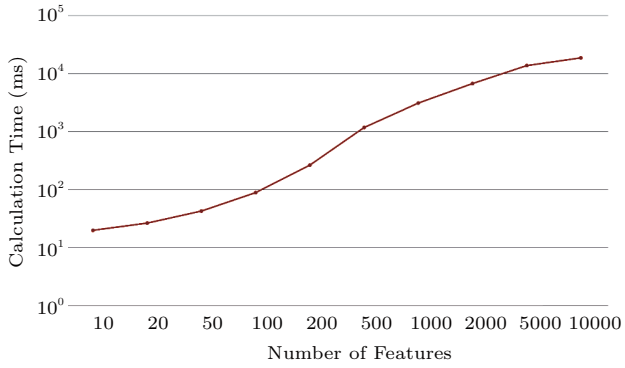

Fig.A1. Experimental results of running reasoning algorithm in order to find refactoring opportunities in a feature model with five random edits, based on the feature model size.

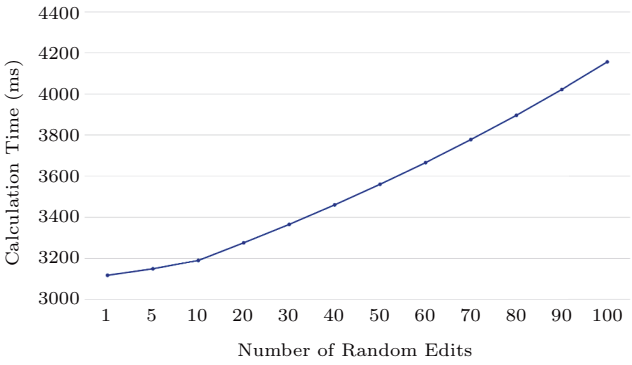

Fig.A2. Experimental results of running reasoning algorithm in order to find refactoring opportunities in a feature model of the size of 1 000, based on the number of edits made in the feature model.