ISSN 1884-0760

GRACE TECHNICAL REPORTS

Supporting Feature Model Refinement with Updatable View

Bo WANG Zhenjiang HU Qiang SUN Haiyan ZHAO
Yingfei XIONG Hone MEI

GRACE-TR 2010–05 May 2010

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Supporting Feature Model Refinement with Updatable View

Bo WANG[1] Zhenjiang HU[2] Qiang SUN[3] Haiyan ZHAO[1]
Yingfei XIONG[4] Hong MEI[1]

[1]Key Laboratory of High Confidence Software Technologies,
Peking University, China
{wangbo07,zhhy, meih}@sei.pku.edu.cn

[2]GRACE Center, National Institute of Informatics, Japan
hu@nii.ac.jp

[3]Department of Computer Science
Shanghai Jiao Tong University, China
sun-qiang@sjtu.edu.cn

[4]Generative Software Development Lab
The University of Waterloo, Canada
yingfei@swen.uwaterloo.ca

Abstract

In the research of software reuse, feature models have been widely adopted to capture, organize and reuse the requirements of a set of similar applications in a software domain. However, the construction, especially the refinement, of feature models is a labor-intensive process, and there lacks an effective way to aid domain engineers in refining feature models. In this paper, we propose a new approach to supporting interactive refinement of feature models based on the view updating technique. The basic idea of our approach is to first extract features and relationships of interest from a possibly large and complicated feature model, then organize them into a comprehensible view, and finally refine the feature model through modifications on the view. The main characteristics of this approach are two folds: a set of powerful rules (as the slicing criterion) to slice the feature model into a view automatically, and a novel use of a bidirectional transformation language to make the view updatable. We have successfully developed a tool, and a nontrivial case study shows the feasibility of this approach.

1 Introduction

In domain engineering, feature models [1][2][3] have been widely used to capture, organize and reuse the requirements of applications in a software domain. An important step of constructing a feature model is to refine a big, abstract feature into small, concrete features. Different approaches have been proposed to guide the refinement of features. For example, in FODA [1], a set of guiding principles are proposed to refine feature models. In FORM [2], features are refined in four layers according to the feature hierarchy, i.e. capabilities, operating environments, domain technologies and implementation techniques.

Feature models grow large during refinement. Reports [4][5][6] show that real-would feature models often grow beyond thousands of features, and the largest one reported [7] has more than 5000 features. On the other hand, feature model refinement becomes more and more difficult when the feature model grows. For one thing, it becomes more difficult to find all features related to the current refinement task. For another, it is difficult to locate a specific feature in the large model.

One way to attack this problem is to organize features related to the current refinement task into a view. Then the domain engineer only modifies the view to accomplish the current refinement task. As the view is usually much smaller than the whole feature model, the refinement task becomes much easier. However, there are several challenges in applying this idea. First, it is difficult to locate all related features that should be contained in the view. Second, it is unknown how to organize these features into a view so that their relations and hierarchical information are preserved. Third, it is unclear how to reflect the modification on the view back into the feature model.

In this paper, we propose a new approach to supporting interactive refinement of feature models based on the above idea. In our approach, first, domain engineers choose features of interest (called initial features); second, a set of features and relationships that may help domain engineers refine the feature model are automatically extracted from the feature model, with the help of eight heuristic slicing rules; third, all the extracted features and relationships are automatically organized into an annotated feature model (called view); and finally, after domain engineers refine the view, we transform all view updates into updates on the feature model by using the bidirectional transformation technique [8]. The main contributions of our paper are summarized as follows:

- We have made the first attempt of applying the *slicing* technical to feature model refinement, by proposing a set of slicing rules to extract related features and relationships based on initial features, and giving an organizing algorithm to organize them into a view that is comprehensible enough for later refinement.

- We define a set of valid modifications on the view and apply the bidirectional transformation technique in building the *updatable view*, so that any refinement operations on the original feature model can be done through the valid modification on the view and any valid modification on the view can be correctly reflected back as a refinement in the original feature model.

- We implement a tool[1] and successfully apply it to refine the web store domain, which shows that our approach to feature model refinement via modification on sliced view is promising and potentially useful in practice.

The rest of this paper is organized as follows. Section 2 gives some preliminary knowledge on feature models. Section 3 introduces a running example that will be used through the paper. Sections 4, 5 and 6 amplify the whole process of our approach. Section 7 illustrates our approach with the web store system case study. Section 8 discusses the related work, and Section 9 concludes the paper and highlights the future work.

2 Preliminary: Feature Models

There currently exist several notations for describing feature model [1][3][9]. In our approach, we adopt, though not limited to, the notation proposed by Zhang et al.'s work [9]. The basic notations and their semantics are summarized in Table 1.

Feature groups with predicates are explained more in Table 2. In this table, $f_1...f_n$ denotes features. For a feature f, bind(f) is a predicate; it is true if f is bound, and false if unbound. And, complex constraints are shown in Table 3. In this table, p and q denote feature groups with predicates.

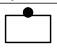
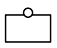
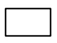
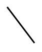
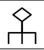
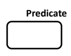
3 A Running Example

We use the web store domain as a running example to demonstrate our approach. A web store is an online shop that sells products or services. In the past few years, more and more consumers choose web stores to purchase products or services. Web store systems are composed of two parts, namely, the Products Purchase and the Business Management. In the Products Purchase, customers can register an account, browse and choose products, check out, pay the order and ask for help when encountering problems. In the Business Management, the shop operators can process orders, manage warehouse, arrange advertisements, start products promotions and manage staffs of the web store. Figure 1 shows part of the web store feature model. This example is constructed according to a real world example [10] which will later be used to evaluate our approach.

It is usually not easy for domain engineers to refine large feature models. For example, if a domain engineer wants to refine feature *Shipping Address*, he needs to find out the features related to it and analyze these features to find any hints for this refinement task. It is difficult for the domain engineer to do it manually in this large feature model. In this paper, we will demonstrate how our approach helps domain engineers collect and organize related parts in the feature model to help domain analysts refine the feature model.

[1]See http://sei.pku.edu.cn/~wangbo07/fmrvb.html for the detail.

Table 1: Symbols and Explanations for Feature Models

Symbol	Name	Explanation
	Mandatory feature	A *mandatory feature* must be bound in a configuration process, if its parent feature is bound .
	Optional feature	A *optional feature* can either be bound or be removed in a configuration process, if its parent feature is bound.
	Feature	In our paper, we use this symbol to denote a feature that can be replaced by either a mandatory feature or a optional feature.
	Refinement relationship	A *refinement* connects the parent (the feature connected to the up end) and the child (the feature connected to the down end). A feature must have one parent. The root feature has no parents. There are three kinds of refinement: decomposition, characterization and specialization.
	Decomposition Relationship	Refining a feature (up end) into its consistent features (down ends) is called *decomposition*.
	Characterization relationship	Refining a feature (up end) by identifying its attribute features (down ends) is called *characterization*.
	Specialization relationship	Refining a feature (up end) into further detailed features (down ends) is called *specialization*.
	Require constraint	A *require* constraint connects the *requirer* (the feature connected to the non-arrow end) and the *requiree* (the feature connected to the arrow end). This constraint means that if the *requirer* is bound in the configuration process, the *requiree* must be bound.
	Exclude constraint	A *excludes* constraint connects two features. This constraint means that these two features cannot be bound in the same configuration.
	Feature group with predicate	A feature group contains a set of features. We define three kinds of feature groups according to their predicates. See Table 2 for the formal definitions of feature groups with predicates.
	Complex require	A *complex require* constraint connects two feature groups. See Table 3 for its formal definition.
	Complex exclude	A *complex exclude* constraint connects two feature groups. See Table 3 for its formal definition.

4 Approach Overview

Our approach is composed of two parts, 1) build an updatable view with feature model slicing, and 2) refine the feature model with an updatable view, as shown in Figure 2. We will introduce the two parts in details in Section 5 and Section 6, respectively. In this section we first give an overview of the approach.

In the first part, the input is a feature model, and the output is an comprehensible view. The first part consists of three activities. In the first activity, domain engineers select a few features (called initial features) according to the refinement tasks. In the second activity, features and relationships that are related to the initial features are automatically extracted from the feature model according to a set of heuristic slicing rules and the initial features. In the third activity, the extracted features and relationships are automatically organized into a view which is a comprehensible annotated feature model.

Table 2: Feature Groups with Predicates

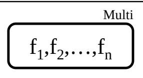
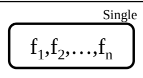
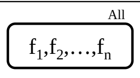
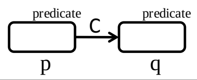
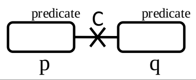
Feature group with predicate	Multi	Single	All
			
Formal definition	$\mathrm{bind}(f_1) \cup \mathrm{bind}(f_2) \cup \ldots \cup \mathrm{bind}(f_n)$	$\mathrm{bind}(f_1) \otimes \mathrm{bind}(f_2) \otimes \ldots \otimes \mathrm{bind}(f_n)$	$\mathrm{bind}(f_1) \cap \mathrm{bind}(f_2) \cap \ldots \cap \mathrm{bind}(f_n)$

Table 3: Complex Constraints

Complex constraint	Complex require (p, q)	Complex exclude (p, q)
		
Formal definition	$p \rightarrow q$	$p \rightarrow \neg q$

In the second part, domain engineers refine the original feature model by examining and modifying the view. We define a set of valid operations on the view and how these operations correspond to operations on the original feature model. With these valid operations, domain engineers can modify the view and all the modifications can be automatically reflected on the original feature model. We apply the bidirectional transformation technique to implement the reflection of view modifications.

As an example, let us see how our approach helps refine feature *Shipping Address* in Figure 1. First, the domain engineer selects initial features of interest. Since a shipping address is used to deliver an order, we suppose the domain engineer selects features *Shipping Address* and *Order Status Notification* as initial features. Then related features are automatically extracted to form a comprehensible view. In this case, the generated view contains *Shipping Address* and sub-trees of *Order Status Notification*. Then the domain engineer only inspects this view and founds that *Shipping Address* must has a child feature *Mobile Number* because there is feature *SMS*, which means that the system can notify the customer of the order status by sending messages to his mobile phone. So he added feature *Mobile Number* as a child feature of *Shipping Address* on the view, and the feature is also automatically added to the original feature model as a sub feature of *Shipping Address*.

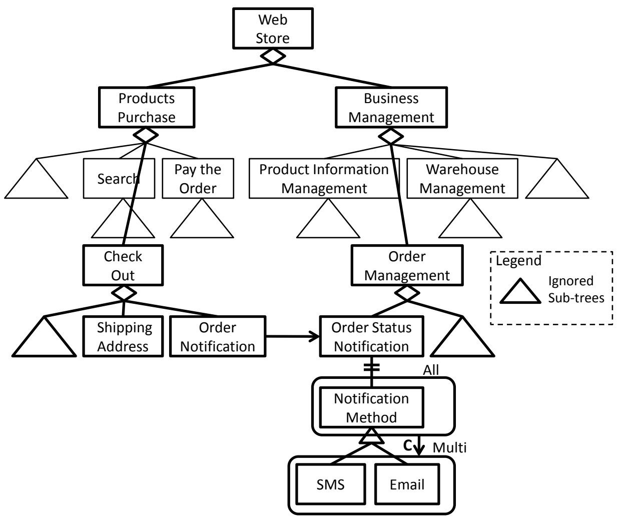


Figure 1: A Simple Web Store Feature Model

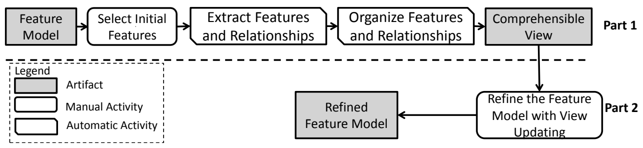


Figure 2: Overview of our Approach

5 Build a Comprehensible View with Feature Model Slicing

In this section, we introduce the first part of our approach, in which features and relationships are extracted from the feature model and organized into a comprehensible view to help domain engineers refine the feature model.

The idea of feature model slicing is inspired by the concept of program slicing. Program slicing [11] takes a program and a slicing criterion as inputs, and returns a sub set of the program that satisfy the criterion.

The feature model slicing takes the feature model and a slicing criterion as inputs. The slicing criterion is denoted as a pair $< F, R >$. F is the feature set that contains all the initial features. R is a rule set that contains all the slicing rules. In the feature model slicing, features and relationships related to the initial features are extracted from the feature model according to the slicing rules and organized into a view with the help of an organizing algorithm.

5.1 Preprocess: select initial features

To obtain the view, domain engineers are requested to select initial features that they want to focus on. The domain engineer may want to refine these initial features, or they may want to use these features to help refine other features. It is worthwhile to note that although it is a free selection process, the initial features have an influence on the generated view because the slicing rules use these initial features as starting points to extract related features and relationships. Therefore, whether the view can greatly benefit the process of refinement depends on the initial features.

5.2 Extract related features and relationships

Based on the initial features, some closely related features and relationships are extracted from the feature model. These features and relationships can help domain engineers refine the feature model.

Eight heuristic slicing rules are provided to identify features and relationships closely related to the initial features. These rules are categorized into two types, namely the Feature Identification Rules and the Relationship Identification Rules.

Feature Identification Rules

The goal of feature identification rules is to identify features closely-related to the initial features. We call these features *extended features*. The slicing rules for identifying extended features are listed as follows.

F1: *Identify extended features by finding offspring of the initial features.*

The offspring of a feature characterize their parent in detail and contain variable points related to the parent. If a domain engineer focuses on one feature, he should also consider the offspring of the feature when refining the feature model. For example, if feature *Warehouse Management* is selected as an initial feature, the domain engineer should also consider feature *Shipping*, *Products Return*, and *Procurement*, as shown in Figure 3(a).

F2: *Identify extended features by finding parents of the initial features.*

A parent feature describes its children in a higher abstraction level and the parent feature helps domain engineers understand the feature model. If a domain engineer focuses on all the children of a feature, he should also consider the parent of these children when refining the feature model. In Figure 3(b), feature *Shipping Method*, *Shipping Address*, *Payment Method* and *Order Notification* are selected as initial features. The parent feature *Check Out* of these initial features can help domain engineers understand these initial features and refine the feature model.

F3: *Identify extended features by finding characterization relationships.*

In a characterization relationship, the sibling features describe their parent from different perspectives. If a domain engineer focuses on one of the children in the

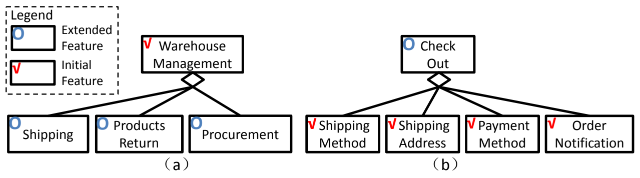


Figure 3: Examples of Rules F1 and F2

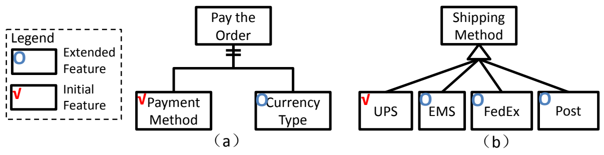


Figure 4: Examples of Rules F3 and F4

characterization relationship, the domain engineer should also consider all its characterization siblings when refining the feature model.

For example, the feature *Pay the Order* is characterized by two attribute features, *Payment Method* and *Currency Type*, as shown in Figure 4(a). If feature *Pay Method* is selected as an initial feature, the domain engineer should also consider *Currency Type*.

F4: *Identify extended features by finding specialization relationships.*

In a specialization relationship, siblings describe different ways of implementing their parent. If a domain engineer focuses on one of the children in the specialization relationship, the domain engineer should also consider all its specialization siblings when refining the feature model.

For example, feature *Shipping Method* is specialized into feature *UPS*, *EMS*, *FexEx*, and *Post*, as shown in Figure 4(b). If feature *UPS* is selected as an initial feature, the domain engineer should also consider the other three features.

Relationships Identification Rule

Generally speaking, all the relationships among the selected features (initial features and extended features) should be extracted, because these relationships help domain engineers understand and refine the feature model. It is worth noting that a constraint is a kind of relationship. A feature group is also a kind of relationship, because each feature group has a predicate on its members. In the following we explain each type of relationship in details.

R1:*Identify refinement relationships between the selected features.*

If all features in a refinement relationship are selected, the domain engineer should also consider the refinement relationships when refining the feature model.

For example, feature *Warehouse Management* is decomposed into three fea-

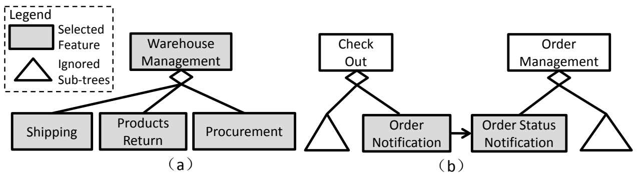


Figure 5: Examples of Rules R1 and R2

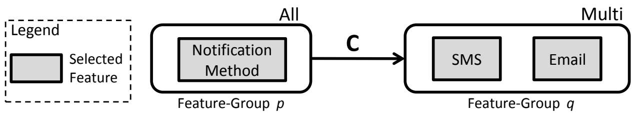


Figure 6: Examples of Rules R3 and R4

tures: *Shipping*, *Products Return*, and *Procurement*, as shown in Figure 5(a). Domain engineers should also consider this decomposition relationship.

R2: *Identify binary constraints between the selected features.*

If both features of a binary constraint (require constraint or exclude constraint) are selected, the domain engineer should also consider the binary constraint when refining the feature model.

For example, *Order Notification* and *Order Status Notification* are selected, and there is a require constraint between them. Domain engineers should also consider this require constraint.

R3: *Identify feature group predicates among the selected features.*

If all features of a feature group are selected, the domain engineer should also consider the feature group predicates when refining the feature model.

For example, *SMS* and *Email* are selected, as shown in Figure 6; domain engineers should also consider the feature group that has a Multi predicate on them.

R4: *Identify complex constraints among feature groups.*

If both feature groups of a complex constraint are included in the slicing, the domain engineer should also consider the complex constraints when refining the feature model.

For example, there are two feature groups *p* and *q* in Figure 6. Group *p* has one feature *Notification Method* and predicate *All*. Group *q* has feature *SMS*, *Email* and predicate *Multi*. When feature groups *p* and *q* are selected, domain engineers should also consider the complex requires between *p* and *q*.

5.3 Organize the features and relationships into a view

To make the results of the extraction more comprehensible, we organize these features and relationships into a view, which has additional annotations and maintains the original level relations among the selected features.

According to rules R1 and R2, the selected features are a set of sub-trees from the original feature model. Usually these sub trees are scattered in the model, which make the results of the extraction hard to understand. For example, there are 5 selected features in the feature model, as indicated by the grey rectangles in Figure 7. These 5 features are in three sub-trees locates in different parts of the feature model.

In our approach, we organize the results of the extraction by computing the lowest common ancestor (LCA) of the roots of the sub-trees. For any two features, their LCA is their shared ancestor that is located farthest from the root feature. With LCAs, artificial features, annotations and artificial relationships are created to organize the results of the extraction.

For each LCA, an artificial feature is created and attached with an annotation which indicates the source of the artificial feature. For example, the LCA of feature *F* and *G* is feature *D*. In the view, artificial feature *AF2* is created as the parent feature of *F* and *G*. This artificial feature is attached with an annotation which indicates that it can be traced to feature *D* in the original feature model, as shown in Figure 7. After we create an artificial feature, we create two artificial refinement relationships from the artificial feature to the roots of the two sub-trees, respectively. In this way we make the artificial feature the common parent of the two sub trees.

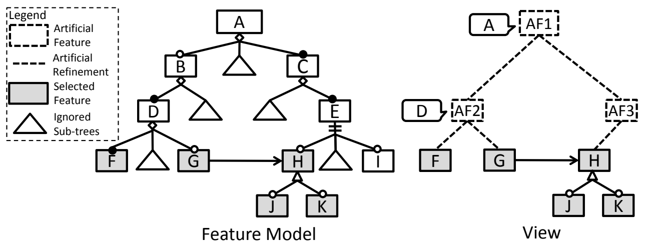


Figure 7: An Example of Constructing a View

Besides adding artificial features for LCAs, our approach also adds artificial features to keep the relative depth of each feature. For example, feature *F* and *H* have the same depth of 4 in the original feature model. After adding *AF2* and *AF1* as common features, feature *F* has a depth of 3 and feature *H* has a depth of 2, and they are not of the same depth. So we add another artificial feature, *AF3*, to make *F* and *H* have the same depth.

The view is built by an organizing algorithm that adopts a bottom up tree construction strategy, as illustrated in Figure 8. This algorithm takes the roots of the sub-trees as inputs and builds the view as output.

This algorithm 1) finds the LCA for each feature pair (f_i', f_j') in the input set *S*, 2) picks the LCA of these two features, 3) generates an annotated artificial feature for the picked LCA if it is are not in the set *S*, 4) adds the artificial refinement relationships from this LCA to the features f_i' and f_j', and 5) removes the features

```
Input: S = {$f_1,f_2,f_3,\ldots,f_n$}
Output: View

Initialization: For all $1 \le i \le n$, $f_i$.selected = true;
while(S has more than one element)
    Find ($f_i', f_j'$) with maximal depth(LCA($f_i', f_j'$)) in {($f_i, f_j$) | $i \neq j, f_i \in S, f_j \in S$} .
    Let $f$ = LCA($f_i', f_j'$) .
    if $f$.selected == false then $f$.addAnnotation();  end if
    if $f == f_i'$ then
        level = distance($f, f_j'$)-1;
        If $f$.selected == false then $f$.addChild($f_j'$,level);  end if
        S.remove($f_j'$);
    else if $f == f_j'$ then
        level = distance($f, f_i'$)-1;
        If $f$.selected == false then $f$.addChild($f_i'$,level);  end if
        S.remove($f_i'$);
    else
        if $f$ is not in S then $f$.selected = false;  S.add($f$); end if
        if $f$.selected == false then
            level = distance($f, f_i'$)-distance($f, f_j'$);
            if level>0 then
                $f$.addChild($f_i'$,level); $f$.addChild($f_j'$,0);
            else if level<0 then
                $f$.addChild($f_i'$,0); $f$.addChild($f_j'$,|level|);
            else
                $f$.addChild($f_i'$,0); $f$.addChild($f_j'$,0);
            end if
        end if
        S.remove($f_i'$); S.remove($f_j'$);
    end if
end while
View.root = S.getOneElement();
return View;
```

Figure 8: An Algorithm for Constructing a View

f_i' and f_j' from the set *S* and adds the LCA into the set *S*. The procedure is iterated until the whole view is built. The key variables and functions are illustrated as follows:

- Each feature has an attribute $selected$ that manifests whether it is in the input set;
- Function $LCA(f_i', f_j')$ computes the LCA of f_i' and f_j';
- Function $f.addChild(f_i', level)$ generates the artificial refinement relationships from f to f_i' by adding the number of level artificial features between them to keep their relative hierarchy. If the parameter level is equal to 0, f_i' becomes a child of f;
- Function $depth(f)$ returns the distance from the root feature to feature f;
- Function $distance(f, f_j')$ computes the length of the path from f to f_j'.

With this algorithm, we ensure that artificial features and relationships are correctly added and relative depths of features are preserved.

6 Refine Feature Models with Updatable Views

In this section, we first define the updatable view; then prove that the updatable view can be built by the GRoundTram system.

6.1 Refine feature models with view updating

The updatable view is defined by a view and a set of valid operations on it. Domain engineers can refine the feature model by using these valid operations to modify the view. These modifications on the view can be automatically reflected back to the original feature model.

In order to clarify the valid operations on view, we will introduce both the valid operations and the invalid operations. The valid operations are provided to facilitate the feature model refinement. And the invalid operations have little contribution to the feature model refinement.

We classify the operations on the view into three categories: *add*, *delete*, and *change*. Domain engineers can add or delete features and relationships. They can also change the attributes of features and relationships.

Invalid Operations

1) Add relationships: If the operation of adding relationship involves artificial features, it is invalid. Adding relationships that contains artificial features cannot help refine the feature model, because artificial features are created only for organizing. For example, adding a require constraint between artificial feature *AF1* and feature *G* is meaningless.

2) Delete features and relationships: If the deleting operation refers to artificial features and relationships, it is invalid. For example, in the updatable view shown in Figure 9, deleting artificial feature *AF3* and the artificial relationship between *AF3* and *H* will make the view hard to understand and cannot help refine the feature model.

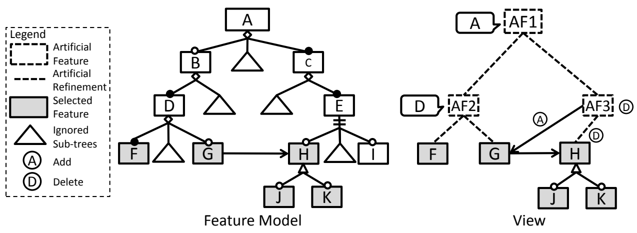


Figure 9: Examples of Invalid Operations

Valid Operations

1) Add features and relationships: If the operation of adding new feature with the selected parent in the view, it is valid. If the relationships that only contain the selected features, the operation of adding them is valid. Figure 10 shows how

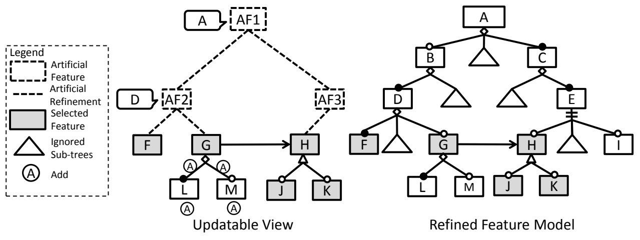


Figure 10: Example of Adding Features and Relationships

the added features and relationships are transformed back to the original feature model. For example, feature *L* and *M* are added as the children of feature *G* with a decomposition relationship in the view. All these modifications are reflected in the refined feature model.

2) Delete features and relationships: If the features are not artificial in the view, the operation of deleting them is valid. If a feature is deleted, all its children are deleted automatically. All the constraints and relationships on the deleted features are also deleted. If the relationships only contain the selected features, the operation of deleting them is valid.

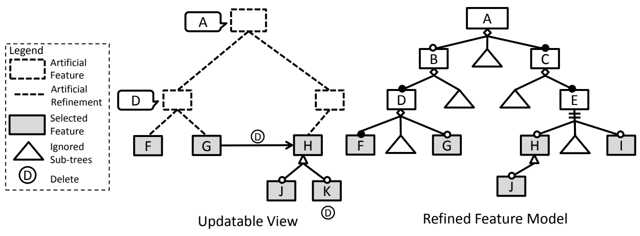


Figure 11: Examples of Deleting Features and Relationships

Figure 11 shows how the deletion of features and relationships are reflected back to the original feature model. For example, feature *L* and the require constraint between *G* and *H* are deleted in the view. All these modifications are reflected in the refined feature model.

3) Change attributes of features and relationships: If the attributes belong to the selected features and relationships, the operations of changing them is valid.

In general, domain engineers can modify any feature and relationship that exist in the original feature model. However, the artificial features and relationships which are introduced to organize and enrich the view cannot be modified, because modifying these features and relationship cannot help refine the feature model.

There is a special case we have to handle when modify the view. If the roots of the sub-trees are deleted in the view, the content and the structure of the artificial features may be changed, because the deletion of the roots may lead to the change of the LCA. In this case, the view needs to be built again after the deletion is reflected back to the original feature model.

For example, if feature *F* is deleted in the view, the feature *F* in the original feature model (see Figure 7 feature model) is also deleted. Then in the original feature model, feature *D* is no longer a LCA. So we have to regenerate the updatable view, as shown in Figure 12.

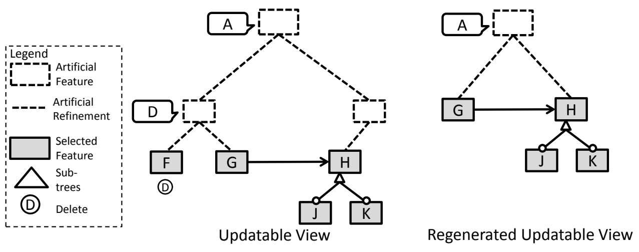


Figure 12: An Example of Deleting the Root of a Sub-tree

It is worth noting that any operation on the original source feature model corresponds to a valid operation on one of its sliced views.

6.2 Use bidirectional transformation to build updatable views

In the previous section we have seen the definition of updatable view. In this section we discuss how bidirectional transformation techniques can be used for easy implementation of an updatable view.

We use the system GRoundTram [12][13], which has been developed to support systematic development of bidirectional model transformation. GRoundTram adapts an existing well established graph querying language UnQL [14] for model transformation. It provides a powerful bidirectional transformation language UnQL+ that extends UnQL and performs an efficient bidirectional computation. In GRoundTram, graphs are edged-labeled in the sense that all information is stored as labels on edges rather than on nodes. The GRoundTram system gives a support to construct an updatable view, maintaining the traceability between the feature model and the updatable view.

We represent, in GRoundTram, a feature model by a source graph model[2], and an updatable view by a target graph model. A forward UnQL+ query is *automatically* created by analyzing the result of feature model slicing. Once a forward query is provided, the backward transformation comes for free by the GRoundTram system. In this way we only need to implement a forward query that extracts a view

[2]See http://sei.pku.edu.cn/~ wangbo07/fmrvb.html for the graph representation of feature models.

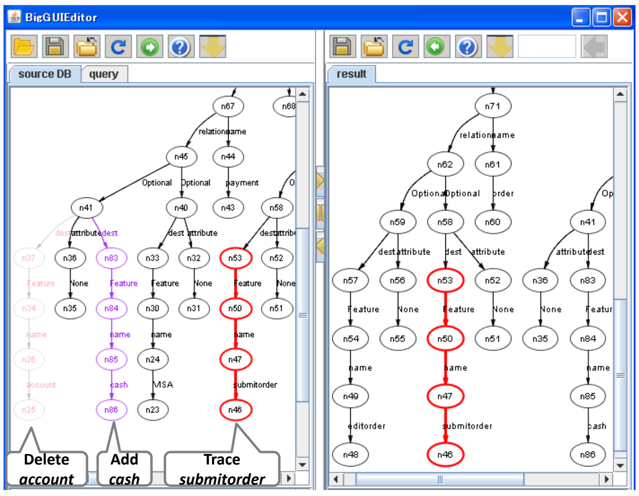


Figure 13: Snapshot of GRoundTram System

from the feature model, and do not have to write code to reflect the updates back into the source.

Besides the easy implementation of the updatable view, another two advantages of using GRoundTram in our approach is that: 1) GRoundTram maintains the history of the modifications caused by backward transformation, so that we can easily implement an undo functionality to cancel a refinement; 2) GRoundTram records the traceability links between nodes in the source graph and nodes in the target graph, so that we can easily support tracing back from features on the view back to the features in the original model. For example, Figure 13 shows a snapshot of GRoundTram system, the source graph model and target graph model are displayed in the left and right part, respectively. Suppose that in the target graph model, we first delete the feature *account* and perform the backward transformation. And then, we add the feature *cash* and perform the backward transformation again. History of these changes is reflected to the source graph model (left in Figure 13). In this modified source graph model, the feature deleted is represented by a set of dash nodes and edges and the feature added is colored with purple. In addition, when we select the feature *submitorder* on the target graph model, the selected feature can be traced back to the source graph model with red highlight.

6.3 Reflect the refinement on views to feature models

Now we show that GRoundTram is a powerful bidirectional transformation system which makes all the refinement on views be truly reflected to the original feature model. There are two facts about GRoundTram [12][13].

Fact 1. *The UnQL+ is as expressive as FO(TC) (first order logic with transitive closure).*

Fact 2. *All the basic graph operations on the projection view of a graph model can be transformed back. These basic operations are: 1) adding/deleting nodes, 2) adding/deleting edges and 3) relabeling edges.*

First, we explain that it is feasible to use UnQL+ provided by GRoundTram to implement the feature model slicing. We define some concepts to describe the view formally. F is a feature set and R is a set of relationships over F. Relationship $r\,(r \in R)$ can be represented as $\{(f_1, f_2)\,|\,(f_1, f_2) \in r, f_1 \in F, f_2 \in F\}$. The union of the relationships in R is defined by $U(R) = \bigcup_{r \in R} r$. Then, the transitive closure can be defined as follows:

- $Set^0(UR) = F_0$,
- $Set^1(UR) = \{f\,|\,(g, f) \in UR,\, g \in Set^1(UR)\}$,
- $TC(UR) = Set(UR) = \bigcup_{i \geq 0} Set^i(UR)$.

Any part of the feature model can be characterized by a FO(TC) formula. Since the extracting features and relationships in the view can be traced back to the feature model, they correspond to a part of the feature model. Therefore, they can be described by FO(TC) formula. The artificial features and relations of the view can be directly illustrated by FO formula. The whole view can be characterized by the FO(TC) formula. Fact 1 guarantees that the forward transformation (i.e., slicing) can be expressed in UnQL+ language.

Then, we show informally that all the valid operations on the view can be reflected backward to the feature model. The view and feature model in GRoundTram are represented in the form of the target graph model and source graph model, respectively. Target graph model contains two kinds of elements, namely traceable elements and artificial elements. Artificial elements are the artificial nodes and edges. Traceable elements are the nodes and edges that can be traced back to the source graph model. Any valid operation on the view is decomposed into some basic graph operations on the traceable elements in the target graph model. Fact 2 guarantees that the valid operations on the view can be backward transformed to the feature model.

7 An example

In this section, we use the web store feature model constructed from a published feature model [10] to illustrate our approach. The constructed feature model has 314 features. In this example, we refine the certain parts of the web store feature model that describe the shopping process in web stores. A typical shopping process in a web store is described as follows: a customer chooses products, fills out the order and pays the order. The store wraps the products and ships them to the customer.

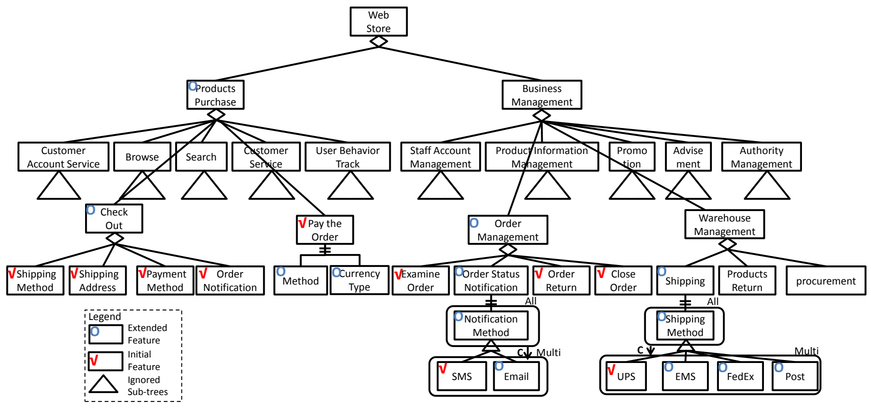


Figure 14: The Web Store Feature Model

This refinement task consists of three steps. First, initial features are selected to express what we may want to refine in the feature model. Then, other features and relationships that may help us to refine the feature model are automatically extracted and organized into an updatable view. Finally, we focus on generated view and modifies it. The modifications on the view are reflected on the original feature model automatically.

7.1 Select initial features

The selection of the initial features is to tell the system what we wants to consider in the process of refinement.

10 initial features are selected to indicates the concerns on the shopping- related parts of the feature model. These initial features are marked with a red hook on the top left corner of the feature, as shown in Figure 14.

7.2 Slice the feature model into an updatable view

Based on the initial features, features and relationships are automatically extracted from the feature model and organized into a view. We use the GRoundTram System to make the view updatable.

25 relationships and 12 extended features are extracted according to the slicing criterion, as shown in Figure 14. The extended features are 3 sub-trees of the original feature model. These five sub-trees located in different part of the feature model, and the roots of these sub-trees are not in the same level, which makes it hard to understand. 3 artificial features, 2 annotations, and 5 refinement relationships are created to organize the result of the extraction into a view, as shown in Figure 15.

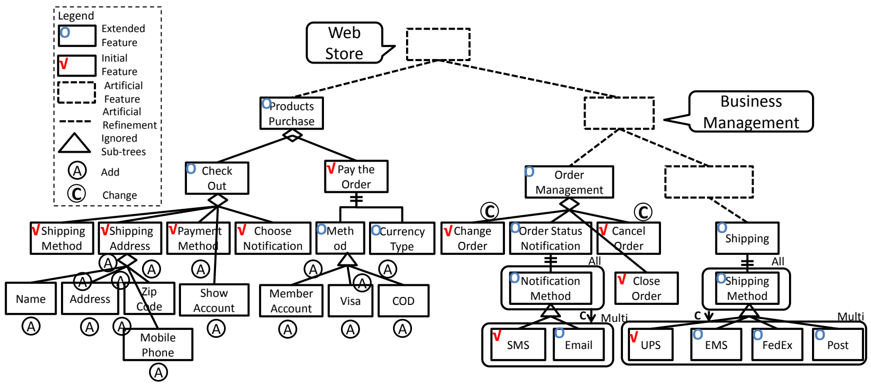


Figure 15: The View of the Web Store Feature Model

7.3 Refine the view

The generated view collects the useful information to help us focus on parts of the original feature model. Then we can refine the feature model by performing the valid operations defined in Section 6 on the view.

Eight features and eight relationships are refined from existing features. Two features are changed in the view. Due to the limit of space, the result of the backward transformation on the feature model is not presented here.

During the refinement, three of eight added features are created with the help of extracted features and relationships, as shown in Figure 15. Features *UPS*, *EMS* and *FedEx* remind us that if web stores support express delivery, it can supports cash on delivery. So feature *COD* is added as a specialized feature of feature *Payment Method*. In addition, if a web store allow customers using a member account to pay an order, the system should show the balance of the member account when customers checks out the order. So feature *Show Account* is created as the child of feature *Checks Out*. Feature *SMS* indicates that the system can notify the status of the order by sending customers an instant message. The system should let customers enter the phone number when they check out. Therefore, feature *Mobile Number* is refined from feature *Shipping Address* for this reason.

We can conclude from the example that the extracted and organized view help us in ways of collecting information and providing some hints to refine the feature model.

8 Related Work

Feature models represent the commonality and variability of the applications in the domains. Basic feature models [1][2] and cardinality-based feature models [3] are both constructed by capturing the commonality in abstraction and variability in step-wise refinement, with the help of a set of principles.

The original concept of a program slice was introduced by Weiser [11]. Weiser defined a program slice as a reduced, executable program that is abstracted from the original program and can produce the specified subset of behavior of the original program. The task of computing program slices is called program slicing. Feature model slicing is inspired by the concept of program slicing. It can be defined by abstracting the meaningful parts from a feature model to form a view. Different from the program slicing, our slicing criterion focus on feature relations rather than program semantics. Besides, we also have an organization algorithm to organize the sliced feature. Kagdi et al. [15] propose UML model slicing. Their definition is general and their approach targets the UML model. Our approach is focused on the feature model.

View updating (bidirectional transformation) has been intensively studied in the database community [16][17][18][19][20]. Recently, a lot of work has been devoted to design of bidirectional languages for developing bidirectional transformation, which has many applications [8] including synchronization of replicated data in different formats [21], presentation-oriented structured document development [22], and interactive user interface design [23], coupled software transformation [24]. Our work is the first attempt of applying the bidirectional transformation to the feature model refinement.

9 Conclusion

In this paper, we show the importance of slicing in feature model refinement, which has not be recognized so far. With the view updating technique, we are able to refine large and complicated feature models. The main features of our approach are two folds: a set of powerful slicing rules to slice the feature model into a view automatically, and a novel use of a bidirectional transformation language to make the view updatable. The updatable view allows domain engineers to refine feature models in an effective way; they can get the extracted and organized information, and refine the feature model by directly modifying the view. Our future work will focus on exploring more accurate slicing rules, working on more practical examples, and investigating applicability of our approach.

References

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

[2] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Ann. Software Eng.*, vol. 5, pp. 143–168, 1998.

[3] K. Czarnecki, S. Helsen, and U. W. Eisenecker, “Formalizing cardinality-based feature models and their specialization,” *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[4] D. S. Batory, D. Benavides, and A. R. Cortés, “Automated analysis of feature models: challenges ahead,” *Commun. ACM*, vol. 49, no. 12, pp. 45–47, 2006.

[5] F. Loesch and E. Ploedereder, “Optimization of variability in software product lines,” in *SPLC*, 2007, pp. 151–162.

[6] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, “Introducing pla at bosch gasoline systems: Experiences and practices,” in *SPLC*, 2004, pp. 34–50.

[7] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, “The variability model of the linux kernel,” in *VaMoS*, 2010, pp. 45–51.

[8] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, “Bidirectional transformations: A cross-discipline perspective,” in *ICMT*, 2009, pp. 260–283.

[9] W. Zhang, H. Mei, and H. Zhao, “Feature-driven requirement dependency analysis and high-level software design,” *Requir. Eng.*, vol. 11, no. 3, pp. 205–220, 2006.

[10] S. Q. Lau, “Domain analysis of e-commerce systems using feature-based model template,” MS in Applied Science, University of Waterloo, Department of Electrical and Computer Enginerring, 2006.

[11] M. Weiser, “Program slicing,” in *ICSE*, 1981, pp. 439–449.

[12] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, “Towards a compositional approach to model transformation for software development,” in *SAC*, 2009, pp. 468–475.

[13] ——, “A compositional approach to bidirectional model transformation,” in *ICSE Companion*, 2009, pp. 235–238.

[14] P. Buneman, M. F. Fernandez, and D. Suciu, “UnQL: A query language and algebra for semistructured data based on structural recursion,” *VLDB J.*, vol. 9, no. 1, pp. 76–110, 2000.

[15] H. H. Kagdi, J. I. Maletic, and A. Sutton, “Context-free slicing of uml class models,” in *ICSM*, 2005, pp. 635–638.

[16] F. Bancilhon and N. Spyratos, “Update semantics of relational views,” *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 557–575, 1981.

[17] U. Dayal and P. A. Bernstein, "On the correct translation of update operations on relational views," *ACM Transactions on Database Systems*, vol. 7, no. 3, pp. 381–416, 1982.

[18] G. Gottlob, P. Paolini, and R. Zicari, "Properties and update semantics of consistent views," *ACM Transactions on Database Systems*, vol. 13, no. 4, pp. 486–524, 1988.

[19] S. J. Hegner, "Foundations of canonical update support for closed database views," in *ICDT '90: Proceedings of the Third International Conference on Database Theory*. London, UK: Springer-Verlag, 1990, pp. 422–436.

[20] J. Lechtenbörger and G. Vossen, "On the computation of relational view complements," *ACM Transactions on Database Systems*, vol. 28, no. 2, pp. 175–208, 2003.

[21] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem." in *POPL '05: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 2005, pp. 233–246.

[22] Z. Hu, S.-C. Mu, and M. Takeichi, "A programmable editor for developing structured documents based on bidirectional transformations," *Higher-Order and Symbolic Computation*, vol. 21, no. 1-2, pp. 89–118, 2008.

[23] L. Meertens, "Designing constraint maintainers for user interaction," Jun. 1998, http://www.cwi.nl/ lambert.

[24] R. Lämmel, "Coupled Software Transformations (Extended Abstract)," in *First International Workshop on Software Evolution Transformations*, Nov. 2004.