Constructing Feature Models Using Merging Techniques with Cross-Product Semantics

Li Yi[1,2], Wei Zhang[1,2], Haiyan Zhao[1,2], Zhi Jin[1,2], Hong Mei[1,2]

Key Laboratory of High Confidence Software Technology (Peking University), Ministry of Education, China.[1]
Institute of Software, School of EECS, Peking University, 100871 Beijing, China.[2]
{yili07, zhangw, zhhy, zhijin}@sei.pku.edu.cn, meih@pku.edu.cn

Abstract. In software reuse, feature models (FMs) provide an effective way to organize and reuse software artifacts in specific domains. It has been observed that the FM of a complex domain often contains thousands of features, and with increasingly use of FMs in practice, the construction of FMs is becoming more and more complex for developers. However, most existing feature modeling methods provide little support to cope with the complexity of FM construction. One possible solution is to transform the construction of a complex FM into the merging of a set of simple FMs, instead of constructing from scratch. In this paper, we propose an FM merging algorithm based on cross-product semantics and rich-refinement types. The cross-product semantics ensures that each configuration of the target FM is a combination of valid configurations from all source FMs, and the rich-refinement types improve the understandability of the target FM by introducing characteristic features into the target structure. Evaluation shows that most features and relationships in the merged FM are acceptable to FM developers.

Keywords: Feature model, Merge, Algorithm.

1 Introduction

In software reuse, feature models (FMs) provide an effective way to organize and reuse software artifacts in specific domains. The concept of FM is first introduced in the FODA method [9]. The idea of FM is to encapsulate software artifacts (e.g. requirements or codes) into a set of features and dependencies among the features, and then to reuse these encapsulated artifacts by selecting a subset of features from a feature model, while maintaining dependencies among the features.

It has been observed that the FM of a complex domain often contains thousands of features, and with increasingly use of FMs in practice (e.g. in software product lines), the construction of FMs is becoming more and more complex for developers. However, most existing feature modeling methods provide little support to cope with the complexity of FM construction. For example, in the FODA method feasibility study [9], the authors point out that, an FM with a little more than 100 features is about to reach the limit of complexity that can be handled by manual construction from scratch.

One possible solution is to transform the construction of a complex FM into the merging of a set of simple FMs, instead of constructing manually from scratch. In this paper, we propose an FM merging algorithm, which is based on cross-product semantics and rich-refinement types. First, the cross-product semantics allows unique features from source FMs to be combined into each configuration derived from the target FM, and also preserves the original constraints among the features from source FMs. In other words, the cross-product semantics ensures that each configuration of the target FM is a combination of valid configurations from all source FMs. Second, the rich-refinement types improve the understandability of the target FM by attaching additional semantic information on the source refinements and introducing characteristic features into the target structure. A major difference between our algorithm and existing merging algorithms is that existing algorithms either do not support combination of unique features, or do not preserve original constraints. Evaluation of the algorithm shows that it generates target FMs with acceptable features, refinements and constraints.

The remainder of this paper is organized as follows. Section 2 gives some preliminaries about feature model. Section 3 discusses the requirements and semantics of our merging operation. Section 4 shows the implementation of the merging operation, i.e. the merging algorithm, and Section 5 gives the evaluation of the algorithm. Related work is presented in Section 6. Section 7 describes future work and concludes this paper.

2 Preliminaries: Feature Model

Feature models are used to describe commonality and variability of products in a specific domain, in terms of *features*. A feature can be defined as an increment in product functionality [9]. Features in an FM connect with each other by two kinds of relationships: *refinements* and *constraints*. The refinements organize features into a tree-like hierarchical structure. Figure 1 depicts an example FM of the mobile phone domain. The refinements between a parent feature and its children can be categorized into:

- *Mandatory*. If a child feature is mandatory, it must be included in the products in which its parent feature appears. For example, every mobile phone must have features like *calls* and *screen.*
- *Optional.* If a child feature is optional, it can be optionally included in the products in which its parent feature appears.
- *Exclusive-Or Relation (XOR)*. If a group of child features have an exclusive-or relation with their parent, only and exactly one child feature can be included in the products in which its parent feature appears. For example, the screen of a mobile phone can be either *basic*, *color,* or *HQ*.
- *Or-Relation*. If a group of child features have an or-relation with their parent, one or more child features can be included in the products in which its parent feature appears.

In addition to the refinements, an FM can also contain cross-tree *constraints* between features. There are typically two kinds of constraints:

- *Requires.* If a feature *X* requires a feature *Y*, the inclusion of *X* in a product implies the inclusion of *Y* in the same product. For example, a mobile phone with a camera must equip a high quality (HQ) screen.
- *Excludes.* If a feature *X* excludes a feature *Y*, both features cannot be included in the same product. For example, a mobile phone with basic screen cannot support GPS functionality.

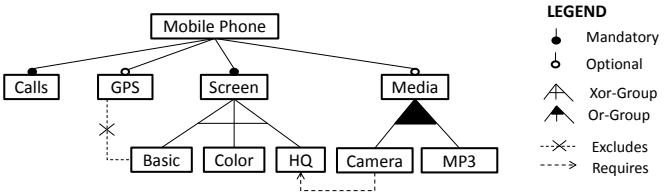

Fig. 1. An example feature model.

Given an FM, products can be derived from the FM by selecting and deselecting the features, while maintaining the relationships between them. In the remainder of this paper, these derived products are called *configurations*. For example, a valid configuration of the FM depicted in Figure 1 is:

{*Mobile Phone, Call, Screen, High Resolution, Media, Camera, MP3*}.

3 Requirements and Semantics of the Merging Operation

In this section, we discuss the requirements and semantics of our merging operation. We first give an illustrative merging scenario, and show the requirements of the merging operation emerging from this scenario. After that, we introduce the semantics which fulfills such requirements.

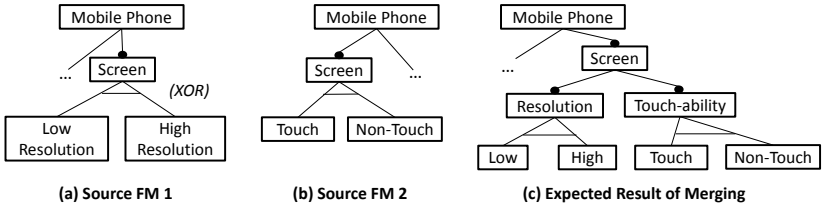

Fig. 2. An illustrative scenario of merging feature models.

3.1 An Illustrative Scenario

Figure 2 illustrates two source FMs to be merged and an expected result. A feature *Screen* is refined to *High Resolution* and *Low Resolution* in one source, but it is refined to *Touch* and *Non-Touch* in another source. In other words, the same feature is refined from different perspectives (*resolution* and *touch-ability*). In this situation, one should expect that the merged FM can express both perspectives so that configurations like a high resolution touch screen can be derived.

3.2 Requirements of the Merging Operation

The illustrative scenario highlights two requirements of the merging operation. First, if the same feature in two source FMs is refined into *unique* child features from different perspectives, the target FM should allow combination of the unique features from both sources.

Second, the combination of unique child features should preserve original constraints among these features in the sources. For example, the original constraints in the sources ensure that a *screen*'s *resolution* must be either *high* or *low*, and it must be either *touch* or *non-touch*. Therefore, one would expect that the target FM should disallow configurations like a *high resolution screen with both touch and non-touch ability*.

3.3 Semantics of the Merging Operation

The semantics of merging operation is defined by the relation between the *configuration set* of source and target FMs. First we represent a configuration of an FM as a set features contained in the configuration. For example, the feature set {*Screen, High Resolution}* denotes a specific configuration. Then Let *f* be an FM and the notation [[*f*]] denote the configuration set of *f*, therefore [[*f*]] is a set of sets, and the semantics of the merging operation is defined as follows:

$$[[Source\ 1]] \otimes [[Source\ 2]] \subseteq [[Target]],$$

where the *cross-product operator* $\otimes$ is defined as (A and B are sets of sets):

$$A \otimes B = \{a \cup b |\ a \in A, b \in B\}\ .$$

The semantics defined above ensures the configurations of the target FM are combinations of valid configurations of both sources. Therefore it fulfills the requirements mentioned before. Table 1 gives the effect of cross-product of the sub-trees rooted *Screen* in the illustrative scenario.

Table 1. Applying cross-product on the illustrative scenario.

[[*Source 1*]]	[[*Source 2*]]	[[*Source 1*]] ⊗ [[*Source 2*]]
{*Screen, High Resolution*}, {*Screen, Low Resolution*}	{*Screen, Touch*}, {*Screen, Non-touch*}	{*Screen, High Resolution, Touch*}, {*Screen, High Resolution, Non-touch*}, {*Screen, Low Resolution, Touch*}, {*Screen, Low Resolution, Non-touch*}

4 The Merging Algorithm

In this section, we present an FM merging algorithm which implements the requirements and the semantics introduced above. We first give a process overview of the algorithm, and describe the steps and related artifacts in this process in details. Finally, we discuss some issues related to the algorithm.

4.1 Overview

Figure 3 gives an overview of the merging algorithm. The algorithm contains two manual steps and two automated steps. First, the preprocessing step adds additional semantics information to refinements in source FMs, and then the source FMs contain rich-refinements. Then the refinements (tree structure) and cross-tree constraints of source FMs are merged based on a set of rules. During the merging of refinements, some auto-generated new features called *characteristic features* may be added to the target FM, and they remain unnamed until the post-processing step assigns proper names for them.

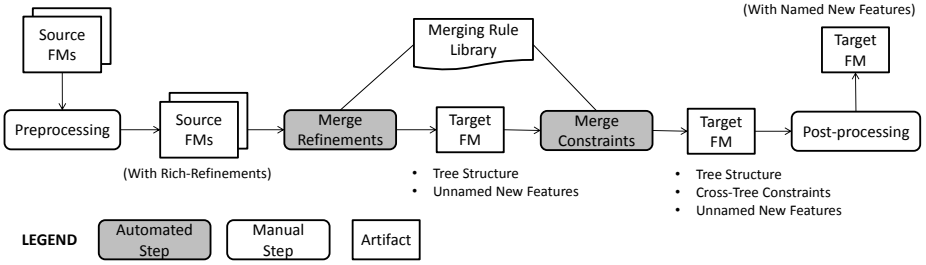

Fig. 3. Main steps of the merging algorithm.

4.2 Preprocessing

The refinements introduced in Section 2 are categorized according to their variability, namely *mandatory, optional, xor-group* and *or-group*. However, the FODA report [9] has proposed that the relationships between parent and child features can also be treated as composed-of, specialization, and implemented-by. Inspired by FODA, our previous work on feature modeling [12] [13] has proposed a classification of refinements based on their semantics, namely *decomposition, specialization* and *characterization.* Decomposition indicates a kind of whole-part refinement, for example, a car can be refined to several parts like engine and light (see Figure 4). Specialization indicates a kind of general-special refinement, for example, a screen can be refined to special cases like touch screen and non-touch screen. Characterization indicates a kind of entity-characteristic refinement, for example, a house can be refined by characteristics like area and height.

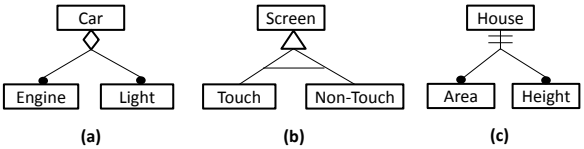

Fig. 4. Semantics of refinements: (a) decomposition, (b) specialization, (c) characterization.

In the merging of feature models, we find that the three kinds of semantics can improve the structure of target FM when merging unique features of the sources.

(Details are described in Section 4.4.) Furthermore, the classification by semantics is orthogonal to the traditional classification by variability, for example, in Figure 4(a) the *Car* is decomposed into two *mandatory* parts, in Figure 4(b) the *Screen* has two *mutually exclusive* specializations, and in Figure 4(c) the *House* has two *mandatory* characteristics. We introduce a preprocessing step to ask developers to mark the semantics for refinements in the sources. As a result, the refinements now carry both variability and semantics information, so we call them *rich-refinements*.

4.3 Merge the Refinements

The merging of refinements is a recursive algorithm (see Algorithm 1). First of all, the root feature of both sources must be matched (i.e. with the same name), and the root feature of target FM is created by copying one of the sources (Line 1 and Line 2). After that, each common child will be merged recursively (Line 4), and the target refinement relationship between the target root and the merged child is calculated according to the source refinement (Line 5), finally the merged child is appended to the target root with target refinement (Line 6). The last step is to merge the unique children of the source roots. For each source root, its unique children are grouped by the refinement semantics (i.e. decomposition, specialization and characterization). These groups are merged one by one, according to rules that will be introduced in Section 4.4 (Line 11 to Line 15).

Algorithm 1. Merging the tree structure.

```
    merge (root1: Feature, root2: Feature): Feature
1     if not-match (root1, root2) then return null;
2     newRoot ← root1.copy();

      // Merge the common children of root1 and root2.
3     for each child ∈ (root1.children() ∩ root2.children())
4       newChild ← merge (child of root1, child of root2);
        // See Section 4.5 for details about calc_target_refinement.
5       newRefinement ← calc_target_refinement (root1, root2, child);
6       newRoot.append_child (newChild, newRefinement);
7     end

      // Merge the unique children of root1 and root2.
8     unique1 ← root1.children() – root2.children();
9     unique2 ← root2.children() – root1.children();
10    while (unique1 ≠ ∅ or unique2 ≠ ∅)
11      group1 ← get_children_of_the_same_refinement_semantics (unique1);
12      group2 ← get_children_of_the_same_refinement_semantics (unique2);
        // Merge group1 and group2, and add to newRoot. (Detailed in Section 4.4.)
13      merge_by_refinement_semantics (group1, group2, newRoot);
14      unique1 ← unique1 – group1;
15      unique2 ← unique2 – group2;
16    end

17    return newRoot;
    end
```

4.4 Rules for Merging Unique Children

The semantics of refinements introduced in Section 4.2 is the basis for merging unique children. The idea is that merging different combinations of semantics should generate different hierarchical structure. Figure 5 shows the rule and an example for each combination. There are seven rules in total:

- *Rule 1: Two Decompositions (Figure 5a).* Merging two decompositions means merging different *parts* of the same feature, so the target FM contains all the parts from both source FMs.
- *Rule 2: Two Specializations (Figure 5b).* Merging different specializations of the same feature needs clarifying the perspectives of the specialization (for example, in Figure 5b one perspective is *resolution*, and another is *touch-ability*), and the perspectives should be outputted as mandatory *characteristics,* and then the source child features become specializations of the characteristic features.
- *Rule 3: Two Characterizations (Figure 5c).* Merging different characteristics of the same feature generates a feature with all these characteristics from the sources.
- *Rule 4: Decomposition and Specialization (Figure 5d).* We first identify the perspective of the specialization (for example, in Figure 5d the perspective is the network *standard* of mobile phones), and then output the perspective as a characteristic feature, as well as the original parts from the decomposition.
- *Rule 5: Decomposition and Characterization (Figure 5e).* We simply generate a target FM with all the parts and characteristics.
- *Rule 6: Specialization and Characterization (Figure 5f).* As we did in previous situation, the perspective of the specialization is outputted as a characteristic. The original characteristics are kept as well.
- *Rule 7: Empty Children and Non-empty Children (Figure 5g).* If the root feature has no child in one source, the result is just a copy of the other source, and the semantics of refinement does not matter.

In the situations described in rule 2, rule 4 and rule 6, there will be some auto-generated *characteristic features* (e.g. *resolution* and *touch-ability*). The names of such feature are actually left *unassigned* during the execution of the algorithm, until the post-processing step in which the FM developers will be asked for assigning proper names for these features. However, for the sake of clarity we fill the names for them in the examples in Figure 5.

An important property of the rules is that the variability information of refinement relationships in the source FMs are kept in the target FM, so that the combination of unique child features does not violate original constraints. Another property is that the rules ensure the configuration set of target FM is exactly the cross-product of source FMs' configuration sets, so the semantics of merging operation is preserved. For example, if we check the example shown in Figure 5(a), we get:

[[*Source 1*]] = { {*Phone, Calls*}, {*Phone, Calls, GPS*} },
[[*Source 2*]] = { {*Phone, Screen*}, {*Phone, Screen, Media*} },

[[*Target*]] = { {*Phone, Calls, Screen*}, {*Phone, Calls, Screen, GPS*}, {*Phone, Calls, Screen, Media*}, {*Phone, Calls, Screen, GPS, Media*} }
= [[*Source 1*]] ⊗ [[*Source 2*]].

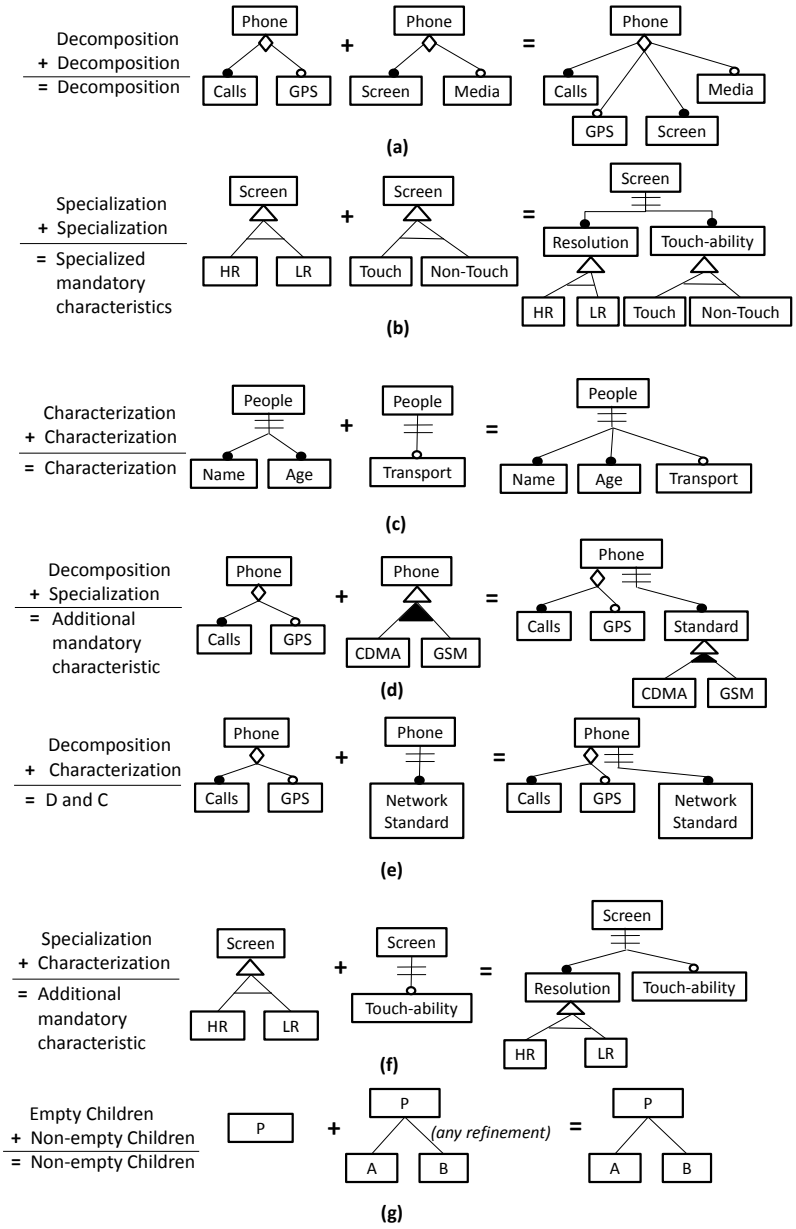

Fig. 5. Rules (at the left) and examples of merging unique children.

4.5 Rules for Merging Common Children

In the merging of unique children described before, we focus on the semantics of the refinements (*decomposition, specialization* and *characterization*), not the variability

of them (*mandatory, optional, xor-group* and *or-group*), because the variability is kept unchanged to preserve original constraints. However, when merging common children, the semantics of refinements in the inputs are exactly the same (since the features in the inputs are exactly the same), so the variability of refinements matters.

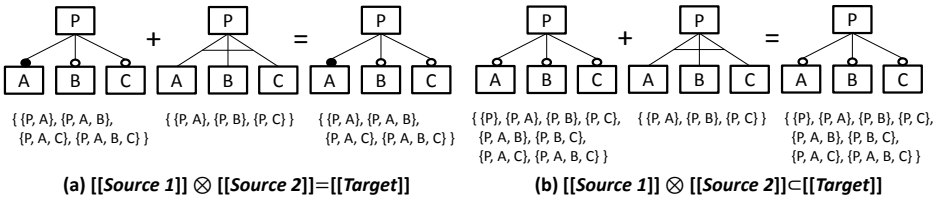

Fig. 6. Examples of merging common children involving *optional* and *xor-group*.

As shown in Table 2, we define ten rules for merging different combination of the variability of refinements in the source FMs. Two of them do not hold the equality in the semantics of our merging operation *when all the children in one source FM are optional*; in other words, they only preserve [[*Source 1*]] ⊗ [[*Source 2*]] ⊂ [[*Target*]] in such case. However, they always hold the equality in other cases. Examples are shown in Figure 6. The other rules always preserve the equality. Therefore in general, the ten rules satisfy the semantics of our merging operation, that is, [[Source 1]] ⊗ [[Source 2]] ⊆ [[Target]].

Table 2. The rules of merging common children according to the variability of refinements

	Mandatory	**Optional**	**Xor**	**Or**
Mandatory	Mandatory	Mandatory	Mandatory	Mandatory
Optional		Optional	Optional *	Optional *
Xor			Or	Or
Or				Or

*: Holds "[[*Source 1*]] ⊗ [[*Source 2*]] ⊂ [[*Target*]]" when all the children in one source FM are optional.

4.6 Merging the Constraints

After the merging of refinements is finished, we merge the cross-tree constraints based on another set of rules. We handle two kinds of constraints, namely *requires* and *excludes*. Each constraint in the source FMs is checked by the rules to generate an output. Table 3 gives definition of the rules, and reasoning of the rules is also based on the cross-product semantics of merging operation. For example, the rationale of rule No.9 is shown as follows:

[[*Source 1*]] = { *Products without A and B*, *Products with B, Products with A and B* },
[[*Source 2*]] = { *Products without A and B, Products with A, Products with A and B* },
[[*Target*]] = { *Products without A and B, Products with A, Products with B, Products with A and B* }
= [[*Source 1*]] ⊗ [[*Source 2*]].

Table 3. The rules of merging constraints

No.	Source 1	Source 2	Target
1	A requires B	*A, B ∉ FS(Source 2), (FS: Feature Set)*	A requires B
2	A excludes B	*A, B ∉ FS(Source 2)*	A excludes B
3	A requires B	*B ∉ FS(Source 2)*	*No constraints between A and B*
4	A requires B	*A ∉ FS(Source 2)*	A requires B
5	A excludes B	*A ∉ FS(Source 2)* or *B ∉ FS(Source 2)*	*No constraints between A and B*
6	A requires B	*A, B ∈ FS(Source 2), but no constraints between A and B*	*No constraints between A and B*
7	A excludes B	*A, B ∈ FS(Source 2), but no constraints between A and B*	*No constraints between A and B*
8	A requires B	A requires B	A requires B
9	A requires B	B requires A	*No constraints between A and B*
10	A requires B	A excludes B	*No constraints between A and B*
11	A excludes B	A excludes B	*No constraints between A and B*

4.7 Post-processing

In the post-processing step, the FM developers should assign proper names for the *characteristic features* generated automatically in the merging of unique children, such as *Resolution* and *Touch-ability* in previous examples. At this point, the merging is finished, and FM developers can continue their work on modifying the target FM.

4.8 Other Issues

In this sub-section, we discuss two related issues of the merging algorithm. Although these issues do not affect the execution of our algorithm, they need to be addressed for better understanding and using of the algorithm. It is also noteworthy that these issues apply to all merging algorithms in the literature so far.

Feature Renaming. In merging algorithms, feature names are treated as the identifier of features. When merging two FMs, two features with the same name in both sources may express different meanings, or two features with different names in both sources may have the same meaning. To handle such situations, *renaming* the source features before merging is necessary. The renaming process needs browsing all the source features, so this could impose considerable work load on developers in practice. Some kind of automatic approach should be introduced to help the situation, but this is beyond the scope of this paper, and we will not discuss it further.

Parent Compatibility. Two FMs are said *parent compatible* if the same feature has the same parent feature in both source FMs [5]. If the source FMs are not parent compatible (in other words, there are hierarchy mismatch between them), our algorithm still works as usual but the target FM will contain *clone features*. An example is shown in Figure 7. Some existing FM merging algorithms [5] [11] do not

allow clone features, and therefore they assume that source FMs are always parent compatible.

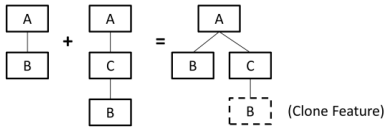

Fig. 7. An example of merging parent-incompatible FMs.

We believe that eliminating parent-incompatibility between the source FMs can only be done case-by-case and manually. Therefore we do not handle incompatibility specifically: we simply generate target FMs with highlighted clone features and leave them to be refactored by developers.

5 Evaluation

In this section, we evaluate the quality of the target FM generated by the merging algorithm. First we describe the design of our evaluation, and then give the results of the evaluation.

5.1 Design of the Evaluation

We conduct five experiments on merging two FMs. (See Table 4.) The source FMs of *Mobile Phone* are taken from [11]. For the *Music Player* experiment, two groups of students (with a few years' experience on feature modeling) build the source FMs independently. The source FMs of *Job Finding Website* are built in the same way. For *Online Shop* and *Laptop*, we first ask a group of students to build a source FM, and we extract another source FM from the literature [2] [3], so the source FMs can also be considered as independently built. Most experiments contain a few renaming on identical features (see "Renamed Common Features" in Table 4).

Table 4. The source FMs of the experiments

FM Name	Features in Source 1		Features in Source 2		Common Features	
	Total	Unique	Total	Unique	Total	Renamed
Mobile Phone [11]	16	7	17	8	9	0
Music Player	48	23	35	10	25	6
Job Finding Website	91	31	77	17	60	19
Online Shop [3]	33	28	21	16	5	2
Laptop [2]	31	22	44	35	9	1

To evaluate the quality of each target FM, we manually check the target FMs in the following aspects (as far as we know, there is no automated way to evaluate the quality of FMs):

- *Correctness of the generated characteristic features (CFs)*. As shown in Section 4.4, the algorithm generates additional characteristic features when merging unique

specializations. We need to check whether each generated feature is necessary or not, and unnecessary characteristic features should be removed from the target FM. (See the "Generated CFs" in Table 5.)

- *Clone features (Parent compatibility).* As discussed in Section 4.8, the algorithm generates clone features when encounters feature with incompatible parents. After the merging, only one of the clones can be kept and others need to be removed by developers. (See the "Clones to Remove" in Table 5.)
- *Correctness of the refinements.* The basic principle is that a refinement is said correct if both the parent-child relation and the variability are correct.
- *Correctness of the constraints.* We also evaluate the correctness of the constraints in target FMs.

Table 5. The evaluation of target FMs of the experiments

FM Name	Features Total	Generated CFs		Clones to Remove	Refinements		Constraints	
		Total	Keep		Total	Keep	Total	Keep
Mobile Phone	26	2	2	0	25	25	1	1
Music Player	67	4	2	5	66	59	4	4
Job Finding Website	108	10	7	8	107	96	4	4
Online Shop	41	2	0	0	40	38	8	8
Laptop	69	1	1	2	68	66	3	3

5.2 Results of the Evaluation

Table 5 gives the results of evaluation on the target FMs. The *features total* includes the generated CFs and clone features. We find that most of the generated CFs are kept in the evaluation, and the unnecessary CFs are generated from the merging of two *partial specializations of the same perspective*. Figure 8 gives an example in the Online Shop experiment, in which the feature *Payment Types* is specialized into *Credit Card / Debit Card,* and *Cash / Vouchers* in the two sources, respectively. The two sources are partial specialization of the same perspective (*payment types,* in this case), therefore the two generated CFs (*CF1* and *CF2*) are unnecessary and need to be removed from the target FM.

When removing unnecessary CFs and clone features, the refinements relating to them should also be removed, this happens in four of the five experiments. However, all of the remaining refinements are considered acceptable; therefore the overall quality of the target refinements is good. Besides, all the target constraints are kept after the evaluation.

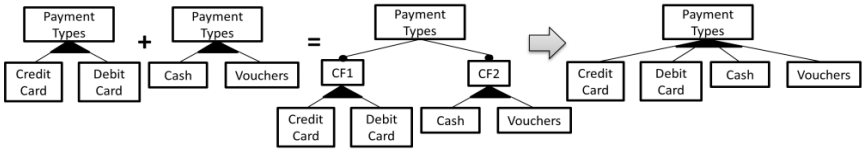

Fig. 8. An example of merging partial specializations of the same feature.

In summary, the evaluation of the merging algorithm shows that most of the generated CFs are reasonable, and target refinements and constraints are highly acceptable. Therefore the FM generated by our algorithm can be a suitable basis for further FM construction. (Figure 9 shows the source and target FMs of *Laptop*.)

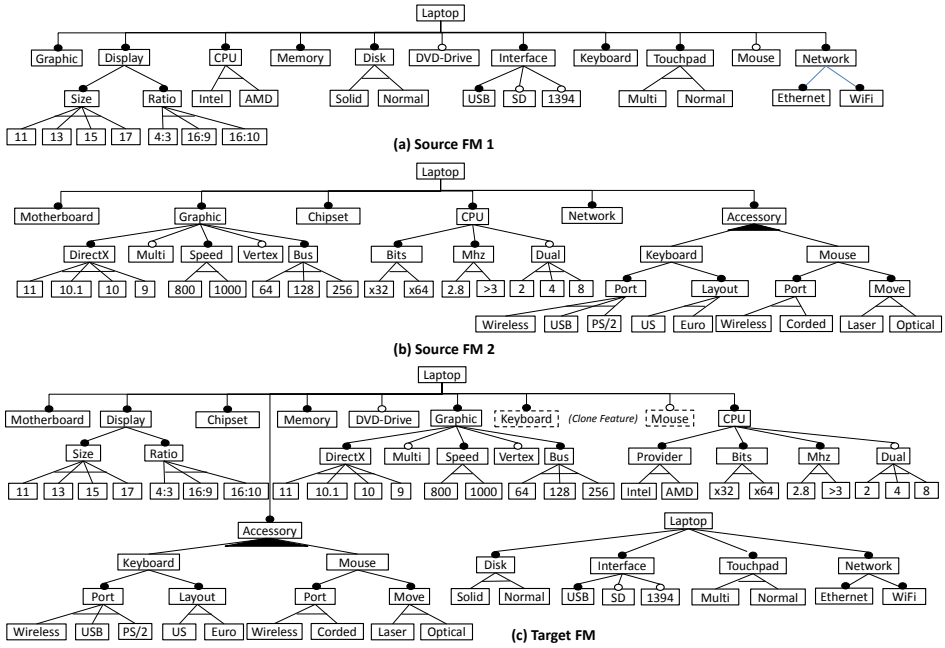

Fig. 9. Source and target FMs of the *Laptop* experiment.

6 Related Work

Some researchers have proposed different kinds of FM merging operation. In this section, we compare their work with ours in two dimensions: semantics and implementation of merging operations.

6.1 Semantics

Besides the cross-product semantics discussed in this paper, there are two kinds of semantics proposed in the literature: *union* and *intersection*. Let f be an FM and $[[f]]$ denotes the configuration set of f, the semantics of union and intersection merging is defined as follows.

Union: $[[Target]] \supseteq [[Source\ 1]] \cup [[Source\ 2]]$. (*Strict union* if the equality holds.)
Intersection: $[[Target]] = [[Source\ 1]] \cap [[Source\ 2]]$.

Compare with Union. The merging algorithms [1, 2, 5, 7, 10, 11] implement the union semantics. The union semantics and cross-product semantics complement each other in some degree because both of them have advantages in different aspects of the merging:

In the merging of unique children, each source can be treated as a partial refinement of a common feature (i.e. a set of partial configurations), but the strict union algorithms [1, 2, 5, 7, 10] do not support feature combination (i.e. the partial configurations cannot be merged into a whole), and the non-strict union algorithm [11] converts all unique children into optional ones which can lead to violation of original constraints. The cross-product semantics is designed to handle such situations in order to provide better results. Figure 10 gives an example of merging two specializations.

In the merging of common children and constraints, the union semantics is actually a more natural way. However it is interesting that most of the merging rules are identical in the union algorithms and ours (6 out of 10 rules in merging common children, and 9 out of 11 rules in merging constraints), this might explain why the common children and constraints generated in our experiments are well-accepted by developers.

As a result, the proportion of common/unique features in source FMs might be a factor in the quality of target FMs. Some kind of "hybrid semantics" seems to be attractive, however a straightforward combination (i.e. cross-product the unique children and union the others) will make the overall semantics of merging operation unclear. We plan to work on this issue in the future.

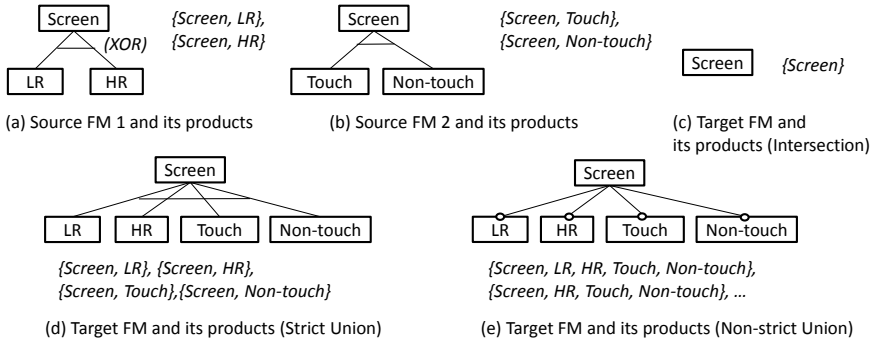

Fig. 10. The illustrative scenario revisited by union and intersection merging.

Compare with Intersection. The intersection operation [1, 10] will eliminate all unique features of source FMs completely. (See Figure 10c for an example.) However, it will preserve all constraints of source FMs when merging common features, therefore it also can be treated as a complement to cross-product semantics.

Choice of the Semantics. Since none of the three kinds of semantics is perfect so far, the choice between them mainly depends on the problem:

Choose union semantics when the original configurations have to be preserved. For example, in [8], a family of configurations is supplied by various vendors, and a master FM is created by merging the vendors' FMs, and its purpose is to manage the FMs and to recreate the supplied configurations on demand.

Choose intersection semantics when all the original constraints have to be preserved but the unique features are not needed. Schobbens et al. [10] proposed a scenario in which constraints on a common set of features are added independently by two developers and their constraints needs to be merged. However, the missing of unique features might be the major obstacle to use intersection for merging FMs in practice.

Choose cross-product semantics when the unique features along with original constraints between them have to be preserved but not the original configurations. For example, given a set of basic features, if they have been refined independently by different stakeholders, then cross-product semantics can help developers construct a master FM via merging these parts.

6.2 Implementation

The implementation of merging operations in the literature can be classified into three styles: direct mapping, rule-based, and logic-based. Our algorithm is actually a rule-based approach.

Direct Mapping Approach. The algorithms by Hartmann et al. [7] and Schobbens et al. [10] are in this style. The idea is to put source FMs side-by-side and add proper constraints between them to make the semantics of merging operation satisfied. In other words, a source FM can be directly mapped into a certain part of the target FM. Compared to our approach, their major advantage is that their algorithms are easier to implement. However, the quality of their target FM is not satisfying, because there are lots of redundancies (each common feature appears at least twice in the target FM) and more importantly, the constraints between the features cannot be clear seen. Therefore a significant amount of modifications on the target FM is needed in developers' further work.

Rule-based Approach. The algorithms by Archer et al. [1], Broek et al. [5], and Segura et al. [11] are in this style, as well as our algorithm. The algorithm in [1] does not merge the cross-tree constraints. Both algorithms in [5] and [11] require that the source FMs are parent compatible, which confines their use because incompatible parents are common in real world.

Logic-based Approach. Archer et al. [2] propose an algorithm in which the source FMs are transformed into logical formulas using the idea of [4], and then calculate the target logical formula which satisfies the semantics of merging operation, and finally transform the logical formula back to target FM with the help of [6]. Compared with our approach, there are three main drawbacks in their algorithm. First, the logic-based algorithm is much harder to implement. Second, the computational complexity of their algorithm is exponential to the number of features in source FMs, while our algorithm is polynomial, so the scalability of their algorithm is doubtful. Third, transforming a logical formula to an FM [6] gets a mal-structured FM (for example, it cannot distinguish between a parent and its mandatory children, and all cross-tree constraints are converted to refinements of the equal logical formula). Therefore a considerable amount of refactoring work is still needed after the merging.

7 Conclusions

In this paper, we propose an FM merging algorithm based on cross-product semantics and rich-refinement types. The cross-product semantics ensures that configurations of the target FM is combinations of valid configurations from source FMs, and the rich-refinement types improve the understandability of the target FM by introducing characteristic features into the target structure. Experiments show that the features and relationships generated in target FMs can be well-accepted by developers.

Our future work focuses on improving the merging operation by integrating the advantages of union and cross-product semantics, and introducing a new kind of semantics for such purpose. We also plan to apply the operation in practice to explore its usability and scalability.

Acknowledgements

This research is supported by the National Natural Science Foundation of China under Grant No. 60821003, 60873059, 90818026; the National Basic Research Program of China (973) under Grant No. 2009CB320701.

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: 2nd International Conference on Software Language Engineering (SLE'09). Volume 5969 of LNCS. (2009) 62–81.
2. Acher, M., Collet, P., Lahire, P., France, R. Managing multiple software product lines using merging techniques. 2010.
3. Antkiewicz M., Czarnecki K. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In: Proceedings of the 204 OOPSLA Workshop on Eclipse Technology.
4. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC'05. Volume 3714 of LNCS. (2005) 7–20.
5. Broek van den, Pim and Galvao, Ismˆenia and Noppen, Joost (2010) Merging Feature Models. In: 14th International Software Product Line Conference, 14 September 2010, Jeju Island, South Korea.
6. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC 2007. (2007) 23–34.
7. Hartmann, H., Trew, T., Matsinger, A.: Supplier independent feature modeling. In: SPLC'09, IEEE Computer Society (2009) 191–200.
8. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC'08, IEEE (2008) 12–21.
9. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S. Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.
10. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. 51(2) (2007) 456–479.
11. Segura, S., Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated merging of feature models using graph transformations. In: GTTSE '07. Volume 5235 of LNCS., Springer-Verlag (2008) 489–505.
12. Zhang, W., Mei, H., Zhao, H.Y. A Feature-oriented Approach to Modeling Requirements Dependencies. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE `05), 2005, 273–282.
13. Zhang, W., Mei, H., Zhao, H.Y. Transformation from CIM to PIM: A Feature-Oriented Component-Based Approach. In: MoDELS'05, Volume 3713 of LNCS. (2005) 248–263.