­­­­­­Constructing Feature Models Us­­ing a Cross-Join Merging Operator

Li Yi1,2, Wei Zhang1,2, Haiyan Zhao1,2, Zhi Jin1,2, Hong Mei1,2

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 existing FMs, instead of constructing from scratch. In this paper, we propose an FM merging operator named *cross-join merge operator*. The operator ensures that *all valid products* of input FMs are preserved in the output FM, *joined* with unique features of all the input FMs, and the input relationships are preserved as well. We give a formal definition of the cross-join operator, and propose a rule-based algorithm to implement the operator. We also give a mathematical proof to show that the correctness of the implementation. An example merging on 6 FMs of mobile phone (all of them are available online) is shown, and some important issues emerging from the example is discussed.

**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 feature modeling is to encapsulate software artifacts (e.g. requirements or code) into a set of features and relationships among the features, and then to reuse these encapsulated artifacts by selecting a subset of features from an FM, while maintaining relationships 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 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 existing FMs in the same domain, instead of constructing from scratch. With the increasingly use of FMs in practice, such existing FMs are not hard to find in several domains. For example, the SPLOT online feature model repository[[1]](#footnote-1) contains more than 10 FMs of the mobile phone domain.

Basically, an FM consists of a set of features and a set of relationships, so the merging of existing FMs (we call them *input FMs* in the remainder of this paper) should handle input features and input relationship properly. First, the merging result (called *output FM* from now on) and the products derived from the output FM should combine unique features of different input FMs. For example, if an input FM of mobile phone proposes a *Wi-Fi* feature, and another input FM states a *3G* feature, one would expect that the output FM can produce a mobile phone with both *Wi-Fi* and *3G* features. Second, because the relationships between features play a critical role in understanding and using FMs, the output FM should preserve input relationships to disallow invalid products; in other words, only valid products are preserved during the merging procedure.

However, existing FM merging methods cannot properly fulfill above needs. They either do not support the combinations of unique features, or do not preserve input relationships. In this paper, we propose an FM merging operator named *cross-join merge operator*. The operator ensures that *all valid products* of input FMs are preserved in the output FM, *joined* with unique features of all the input FMs, and the input relationships are preserved as well. We give a formal definition of the cross-join operator, and propose a rule-based algorithm to implement the operator. We also give a mathematical proof to show that the correctness of the implementation. An example merging on 6 FMs of mobile phone (all of them are available online) is shown, and some important issues emerging from the example is discussed.

The remainder of this paper is organized as follows. Section 2 gives some preliminaries about FM. Section 3 gives the definition of cross-join operator after showing a motivating example. Section 4 proposes an implementation of the operator, including a proof of correctness. Section 5 gives an example merging and discusses some issues. Section 6 compares our work with related work. Finally, 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, and root of the tree is called *root feature*. Figure 1 depicts an example FM of the mobile phone domain. The refinements between a parent feature and its children can be categorized into:



**Figure 1. An example feature model.**

* *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.

Given an FM, *products* can be derived from the FM by selecting and deselecting the features, while maintaining the relationships between them.For example, a valid product of the FM depicted in Figure 1 is:

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

3 Definition of the Merging Operator

In this section, we first give a motivating example of FM merging, and show properties of the merging operator emerging from this example. After that, we define the merging operator by formalizing the properties.

3.1 A Motivating Example

Figure 2 illustrates two input FMs of the *Mobile Phone* domain and an expected result of merging. The input FMs may have common features (e.g. *Mobile Phone, Wi-Fi,*



**Figure 2. A motivating example of merging feature models.**

*3G,* and *Screen*) and unique features (e.g. *HD, SD, Touch,* and *Non-Touch*). Besides, the refinements among the common features may be different. For example, both *Wi-Fi* and *3G* are optional features in one input, but they belong to an *or-group* in the other.

The rationale behind the expected output FM is that the product of the output FM is a valid product of an input FM properly joined by unique features of another input FM. Table 1 describes the products of input and output FMs. For example, a product of the output FM -- *a mobile phone with an HD touch screen but no Wi-Fi and 3G functions* -- can be derived from a valid product of the first input FM (*A mobile phone with an HD screen*) plus a unique feature of the second input FM (*Touch*).

The words “properly joined” stated in the rationale express that the expected output FM is that it preserves the constraints among the unique features of an input FM while adding them to another input FM’s products. For example, the second input FM has an *exclude* constraint between *Touch* and *Non-touch,* and the output FM preserves the constraint so that it disallows unexpected products such as *a mobile phone with an HD touch non-touch screen*.

**Table 1. Product description of the illustrative input and output FMs**

|  |  |  |
| --- | --- | --- |
| **Input FM 1** | **Input FM 2** | **Output FM** |
| Mobile phone with a   * SD or HD screen   and  *0, 1, or 2* of these modules:   * Wi-Fi * 3G | Mobile phone with a   * touch or non-touch screen   and  *1 or 2* of these modules:   * Wi-Fi * 3G | Mobile phone with a screen of two characteristics:   * SD or HD, and * Touch or Non-Touch   and  *0, 1, or 2* of these modules:   * Wi-Fi * 3G |

3.2 The Merging Operator

In this sub-section, we formalize the semantics of the merging operator in the form of pre and post conditions. Firstly, we define some symbols for expressing the feature set, the product set, and the root feature of an FM.

**Definition 1 (Feature Set).** Given an FM, *m*, we use the symbol *FS(m)* to denote the set of its features.

**Definition 2 (Product).** Given an FM, *m*, a product *p* is a subset of *m*’s features, that is:

**Definition 3 (Product Set).** Given an FM, *m*, we use the symbol *PS(m)* to denote the set of its products, that is:

**Definition 4 (Root Feature).** Given an FM, *m*, we use the symbol *Root(m)* to denote the root feature of *m.*

In the motivating example, we need to identify common and unique features of the input FMs, so we need to find out whether two features are equal. We define feature equality as below.

**Definition 5 (Feature Equality).** Given two features, *f1* and *f2*, we define that *f1 is equal to f2*, denoted as *f1 = f2,* if *f1* and *f2* have the same name.

**Definition 6 (Cross-Join Merging Operator).** Given two FMs, *m1* and *m2*, we define a binary operator on FMs (denoted by) as a cross-join merging operatoron *m1* and *m2*, if the following conditions are satisfied:

* Pre-condition

(1)

* Post-conditions

, (2)

where *i, j = 1, 2, ij*. (3)

The pre-condition demands that the root feature of input FMs must be equal to each other. The reason is that if and the symbol *Root(Output)* denotes the root feature of the output FM, then we can deduce that:

. (4)

However, it is a trivial fact that the root feature belongs to every product of an FM, so Formula (4) violates Formula (3).

The post-conditions ensure that the merging result is still an FM, and it satisfies the *cross-join* property: the products of the output FM is formed by joining products of an input FM () and valid unique features of another input FM (). The name “cross-join” indicates that unique features acrossinput FMs are joined into output products.

4 Implementation of the Merging Operator

In this section, we present an FM merging algorithm which implements the merging operator introduced above. We first give a process overview of the algorithm, and then describe its steps in details. Finally, we prove that the implementation satisfies the definition of the operator (see Definition 6).



**Figure 3. Main steps of the merging algorithm.**

4.1 Overview

Figure 3 gives an overview of the merging algorithm. The algorithm contains three main steps. First, the feature trees of input FMs are merged into an output feature tree which contains all the common and unique features. During the merging of feature trees, there may be some *feature clones* (i.e. a feature appears more than once) in the output feature tree, so an intermediate step handles the feature clones to ensure the post-conditions. Finally, the cross-tree constraints (i.e. *requires* and *excludes*) are merged and added to the feature tree so that the output FM is generated.

4.2 Merge Feature Trees

The input feature trees are merged recursively (see Algorithm 1). Firstly, if the pre-condition is not satisfied, the merging procedure ends immediately (Line 1). Then the

**Algorithm 1. Merging Feature Trees**

|  |
| --- |
| **merge\_tree** (root1: Feature, root2: Feature): Feature {  1 if () { return null; }  2 root root1.copy();    *// Merge the common children of root1 and root2.*  3 common root1.children() root2.children();  4 for each (Feature c common) {  5 child **merge\_tree** (root1.get\_child(c), root2.get\_child(c));  6 rf merge\_refinement\_by\_rule (root­1.get\_refinement(c), root2.get\_refinement(c));  7 root.append\_child (child, rf);  8 }    *// Append the unique children of root1 and root2.*  9 for each (Feature u1 ) {  10 root.append\_child (u1, root1.get\_refinement(u1));  11 }  12 for each (Feature u2 ) {  13 root.append\_child (u2, root2.get\_refinement(u2));  14 }  15 return root;  **}** |

**Table 2. Rules of merging refinements on common features**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Mandatory | Xor | Or | Optional |
| Mandatory | Mandatory | Or | Or | Optional |
| Xor |  | Xor | Or | Optional |
| Or |  |  | Or | Optional |
| Optional |  |  |  | Optional |

root feature of the output FM is created (Line 2). Common children of input root features are merged one by one (Line 3 to Line 5), and refinements between the root feature and the merged children are computed according to the input refinements and the rules listed in Table 2. Finally, unique children of both input root features are simply appended to the output root (Line 9 to Line 14).

**Rationale of the Rules.** We first give the following definitions for clarity.

**Definition 7 (Partial Product on a Set of Features).** Given a product of an FM, *p,* and a set of features *S*, we define the partial product of *p* on the feature set *S* as:

**Definition 8 (Partial Product Set on a Set of Features).** Given an FM *m* and a set of features *S*, we define the partial product set of *m* on the feature set *S* as:

The rules of merging refinements on common features (Table 2) produce the *union* of input partial product sets on common features, that is, given input FMs *m1* and *m2*, the output FM *m* satisfies:

where (5)

We can check Table 2 against Formula (5) in the following way: Given a common parent feature *r*, and a set of *r*’s common children {*c1, c2, …, ck*}, we construct four FMs:

* *m1*: all children are mandatory (the “Mandatory” row and column in Table 2),
* *m2*: all children are xor-grouped (the “Xor” row and column in Table 2),
* *m3*: all children are or-grouped (the “Or” row and column in Table 2),
* *m4*: all children are optional (the “Optional” row and column in Table 2).

Let , we can deduce that:

and

Therefore we get the merged refinements filled in Table 2.

4.3 Handle Feature Clones

An important issue in FM merging is the *hierarchical mismatch* between common input features, that is, a common feature has different parent features in different input



**Figure 4. An example of hierarchical mismatch and feature clones.**

feature trees. Figure 4 gives an example, in which the common features *Wi-Fi* and *3G* have different parent features in input FMs. According to Algorithm 1, hierarchically mismatched common features appear in every output feature sub-tree rooted by their parent features, so that there are feature clones in the output FMs. In the example shown in Figure 4(c), *Wi-Fi* and *3G* are cloned in the output.

In order to satisfy the post-conditions of the merging operator, we add *mutual requires* relationship between each pair of feature clones. It ensures that the status of clones (i.e. whether a clone appears in a certain product) are always identical, as if there is only one instance in the output FM.

**Algorithm 2. Merging Constraints**

|  |
| --- |
| merge\_constraints (c1: ConstraintSet, c2: ConstraintSet, common: FeatureSet): ConstraintSet {  1 result new ConstraintSet();  2 for each (Constraint ) {  3 matched *false*;  // Type 1: Matched constraints  4 if (x.features common) {  5 for each (Constraint y c2) {  6 if (y.features = x.features) {  7 matched *true*;  8 m compute\_constraint\_by\_rule (x, y);  9 result result m;  10 c2 c2 y; // Remove y from c2.  11 }  12 }  13 }  // Type 2: Non-matched constraints  14 if (matched = *false*) {  15 result result x;  16 }  17 }  18 if () {  19 result result c2;  20 }  21 return result;  } |

**Table 3. Rules of merging constraints on common features**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Input 1** | **Input 2** | **Output** |
| 1 | A requires B | A requires B | A requires B |
| 2 | A requires B | B requires A | *No constraint between A and B* |
| 3 | A requires B | A excludes B | *No constraint between A and B* |
| 4 | A excludes B | A excludes B | A excludes B |

4.4 Merge Constraints

Algorithm 2 shows the process of merging constraints. Most of the process is to find the *match* between input constraints (Line 3 to Line 7), that is, two constraints from different input FMs involve the same set of features. Matched constraints are merged by rules (Line 8 to Line 10), and non-matched ones are simply added to the result (Line 14 to Line 20).

**Rationale of the Rules.** The rationale of rules listed in Table 3 is similar to rules of merging refinements (Section 4.2), that is, given input FMs *m1* and *m2*, the output FM *m* satisfies:

where (6)

We can similarly check Table 3 against Formula (6) by constructing four FMs and computing their partial product set on a given feature set *S = {A, B}*, as below:

* *m1*: *A* requires *B*,
* *m2*: *B* requires *A*,
* *m3*: *A* excludes *B*
* *m4*: No constraint between *A* and *B*.

Then we can deduce that:

and

Therefore we have got Rule 2 and Rule 3 (Rule 1 and Rule 4 are trivial), so that Table 3 satisfies Formula (6).

4.5 Proof of Correctness

In this sub-section, we want to prove that when pre-condition (1) is satisfied, the implementation described above satisfies post-condition (2) and (3).

The proof of satisfaction of post-condition (2) is trivial, since Algorithm 1 always generates a valid feature tree when pre-condition (1) is satisfied. We focus on post-condition (3) here, and re-state it as Theorem 1.

**Theorem 1.** Let *m1, m2* be two input FMs, and *m* be the output FM generated by the implementation described above, then it can be deduced that:

where *i, j = 1, 2, ij.*

**Proof.** We use *F* to represent the whole set of features of input and output FMs, that is:

We divide *F* into two subsets: one is the *hierarchical matched common features* of *m1* and *m2*, denoted by *H*; the set difference contains the rest common features and the unique features.

First, we consider the partial product set of *m* on the feature set *H.* According to Formula (5) and (6), it is trivial to see that:

(7)

Second, we consider the partial product set of *m* on the feature set According to Algorithm 1 and 2, the refinements and constraints of input FMs on *D* are kept unchanged in *m*, so it can be deduced that:

(8)

In addition, a product *w* can be express as:

(9)

Therefore, for any product *x* of an input FM *mi* (i = 1 or 2), there exists a product *y* of another input FM *mj* (j = 1 or 2 and ), and a product *p* of the output FM *m*, such that:

(According to Formula (7).)

and . (According to Formula (8).)

We take the union of the above two equations and get that:

*p*

Thus we immediately get that. In addition, we have:

. (According to Formula (9).)

Therefore Theorem 1 has been proved.

5 An Example

In this section, we give an example of merging 6 FMs which are available online. We also discuss some issues found in the example and their possible impacts on FM merging in the real world.

5.1 The Input and Output FMs

The input FMs are taken from the SPLOT Feature Model Repository. There are 11 FMs on *Mobile Phones* in the repository, some of them are totally identical, so finally we get 6 different FMs. Table 4 shows the information about them. They can be

**Table 4. Summary of the input FMs.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No. | Total Features | Non-unique Features | | Unique Features | | Constraints |
| Total | Renamed | Total | Renamed |
| 13 | 10 | 8 | 0 | 2 | 0 | 2 |
| 25 | 11 | 6 | 3 | 5 | 0 | 2 |
| 55 | 14 | 1 | 0 | 13 | 1 | 0 |
| 59 | 15 | 12 | 3 | 3 | 0 | 2 |
| 67 | 16 | 6 | 2 | 10 | 0 | 0 |
| 107 | 25 | 12 | 4 | 13 | 1 | 2 |

viewed in the “Feature Model Editor” page on the repository website, according to their numbers (the “No.” in Table 4). The same feature may have different names in different FMs, and different features may have the same name in different FMs. We manually rename them to avoid such situations before the merging, and Table 4 shows how many renaming occurred. In Table 4, we use the term “non-unique features” instead of “common features” because they may not be common in all 6 FMs. The FMs are merged one by one, in the ascend order of their numbers.

Figure 5 shows the final result of merging (the root feature *Mobile Phone* is not drawn for layout concern). The output FM contains 65 features, including 4 pairs of feature clones (drawed in dash line), so there are 61 distinct features. There are 8 constraints generated from input FMs, and 4 *mutual-requires* constraints between the feature clones.

5.2 Discussions

We identify two issues emerging from the simple example. One of them indicates the need of pre-processing on input FMs, and the other shows the need of post-processing on the output FM.

**Feature Equality.** In Section 3.2, we give a definition of feature equality based on the equality of feature names. However, in real FM merging, it is not rare that the same feature has different names in different FMs, and different features share the same name in different FMs. In our example, we manually rename the features to handle the problem (the *Renamed* column in Table 4).

However, manual renaming is not scalable because one has to view and analyze all features of input FMs. In the real world, FMs often have hundreds or even thousands of features so the work load is unacceptable. We suggest that an automated method should be used to compute feature equality, and such a method should consider at least two kinds of information:

* **Textual Information.** Equality might be computed as the similarity between names and descriptions of two features. The computation can utilize a dictionary of domain terminology as well.
* **Structural Information.** We have observed that many equal features have similar descendant features. For example, the feature *Connectivity* in Figure 5 are named as *Wireless*, *Communication, Data Transfer* and *Connectivity* in four input FMs, respectively. We determine that they are equal features because all of

them have child features like *Bluetooth* and *Wi-Fi*. The computation of equality should take account of such structural similarity information.

**Feature Clones.** An FM with feature clones may not be a desirable result, so a post-precossing step should be incorporated to refactor the output FMs. In our example such a refactoring seems easy (i.e. re-merge the feature trees rooted by *Call* and *Message,* respectively). However, we do not incorporate the re-merge steps in our algorithm, because such steps might violate the post-condition (3) defined in the merging operator, and it is highly possible that in more complex situations, re-merge steps cannot eliminate all feature clones. Therefore, we leave the refactoring work to be manually performed.



**Figure 5. The result of merging 6 FMs of mobile phone domain.**

6 Related Work

Researchers have proposed several kinds of FM merging operators. In this section, we compare their work with ours in two dimensions: definition and implementation of merging operators.

6.1 Different Definitions of Merging Operators

Besides the cross-joinoperator proposed in this paper, there are two kinds of operator proposed in the literature: *union operator* and *intersection operator*. Given two input FMs, *m1* and *m2*, the output FM *m* satisfies the following post-condition:

(Union) (*Strict union* if the equality holds),

(Intersection)

**Compare with Union.** The union merging operator is implemented in [1, 2, 5, 7, 10, and 11]. Comparing with the cross-join operator, the main drawback of union operator is that it cannot handle unique features properly. Figure 6 shows the result of union merging of FMs in our motivating example (Section 3.1). The strict union operators [1, 2, 5, 7, and 10] do not allow combination of unique features in the input FMs (e.g. *mobile phones with HD touch screen* cannot be derived from the output FM). The non-strict union operator [11] does not preserve the origin constraints among unique features, so that invalid products (e.g. *SD-HD screen*) are not eliminated.

The main advantage of union operator is that it perfectly preserves product sets of input FMs. Therefore the union operator is a better choice when the mapping between products and input FMs must be preserved in the output FM. For example, in [8], FMs and corresponding products are provided by various vendors, and the customer want to create a master FM to manage the vendor FMs and vendors’ products can be re-derived from the master FM on demand.

**Compare with Intersection.** The intersection operator [1, 10] eliminates all unique features of input FMs. However, it preserves all constraints of input FMs. Schobbens et al. [10] proposed a scenario in which constraints on *a* *common set of features* are added independently by many developers and their constraints need to be merged. In such a scenario the intersection is practical because there is no unique feature to lose. However, the missing of unique features is the major obstacle to apply intersection operator to other FM merging scenarios in practice.

6.2 Different Implementation Ways

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



**Figure 6. The motivating example revisited by union and intersection operator.**

**Direct Mapping Approach.** The algorithms by Hartmann et al. [7] and Schobbens et al. [10] are in this style. The idea is to put input FMs side-by-side and add proper constraints between them to implement the merging operator. In other words, an input FM is directly mapped into a certain part of the output FM. Compared to our approach, which is rule-based, the major advantage of direct mapping approach is its simplicity. However, the quality of its output FM is not satisfying, because there are lots of redundancies (each common feature appears at least twice in the output FM) and more importantly, constraints between the features cannot be clearly understood. Therefore a significant amount of refactoring work on the output FM is needed.

**Rule-based Approach.** The implementations by Acher et al. [1], Broek et al. [5], and Segura et al. [11] are in this style, as well as ours. However, the implementation in [1] does not merge the cross-tree constraints. Both implementations in [5] and [11] require that input FMs do not contain hierarchical mismatches, which confines their use because hierarchical mismatch is common in practice.

**Logic-based Approach.** Acher et al. [2] propose an implementation in which the input FMs are transformed into logical formulas using the idea of [4], and then the output logical formula is constructed according to post-conditions of merging operators, and finally the output logical formula is transformed to an FM with the help of [6]. Compared to rule-based approaches, the major advantage of logic-based approach is that the correctness of the implementation can be strictly proved. However, there are three main drawbacks in the logic-based approach. First, it is much harder to implement. Second, its computational complexity is exponential to the number of features, while our implementation is polynomial, so the scalability of logic-based approach is doubtful. Finally, transforming a logical formula to an FM [6] produces a mal-structured FM (it cannot distinguish between a parent and its mandatory children, and all cross-tree constraints are converted into refinements). Therefore a considerable amount of refactoring work is still needed after the merging.

7 Conclusions

In this paper, we propose an FM merging operator named cross-join merging operator. The operator ensures that *all valid products* of input FMs are preserved in the output FM, *joined* with unique features of all the input FMs, and the input relationships are preserved as well. We give a formal definition of the cross-join operator, and propose a rule-based algorithm to implement the operator. We also give a mathematical proof to show that the correctness of the implementation. An example merging on 6 FMs of mobile phone is shown, and we also discuss the issues about feature equality and feature clones in FM merging.

Our future work focuses on applying the operation in practice to explore its usability and scalability. We also want to address the feature equality problem, and we plan to incorporate text mining techniques to compute feature equality based on their names, descriptions, and structural characteristics.

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.

1. http://www.splot-research.org/ [↑](#footnote-ref-1)