**Constructing Feature Models Us­­ing a Cross-Join Merging Operator**

Authors Name/s per 1st Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail address if desired

*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

# 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 [10]. 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 [10], 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 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.

# 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 [10]. 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:

* 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.



1. An example feature model.

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}.

# 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.

## 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, 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.



1. A motivating example of merging feature models.

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 I 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.

1. Products derived from the motivating 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   * SD or HD plus * touch or non-touch   screen  and *0, 1, or 2* of these modules:   * Wi-Fi * 3G |

## 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 products, and the product set 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* derived from *m* 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 (Cross-Join Merging Operator).** Given two FMs, *m1* and *m2*, we define a binary operator on FMs (denoted by ) be a cross-join merging operator on *m1* and *m2*, if the following pre- and post-conditions are satisfied:

Pre-condition:



Post-conditions:





Equation (3) formalizes that for any product *x* of an input FM, there is a product *p* of the output FM that not only preserves the product *x* (), but also synthesizes unique features of a product *y* of another input FM (). The formalism defines two properties of the operator:

**Property 1 (The Cross-Join Property).** In SQL, a cross-join of two tables produces rows that combine the rows from the first table with the rows from the second table. (The cross-join is normally achieved by the statement “SELECT \* FROM table1, table2”.) By analogy we borrow the name for our operator, since output products are combinations of features from both input FMs..

**Property 2 (The Valid Property).** Furthermore, in SQL, the cross-join statement is rarely used alone – it is often followed by a WHERE statement to ensure that only valid combinations are allowed. Equation (3) also states that an output product (*p*)is a combination of a *valid* input product (*x*) and *valid* unique features () of another input product (*y*). In other words, the constraints from both input FMs are preserved in the output products.

The pre-condition demands that the input FMs are valid, i.e. at least one product can be derived from the FM [5], so that an implementation of the merging operator does not need to worry about error handling and can focus on the merging itself. Note that Equation (3) implies that so that an implementation of the operator should be able to detect and resolve conflicts between input FMs, if any.

Finally, we would like to point out that there is a special case of the operator, when the equality always holds in , in Equation (3). Formally, in such cases, Equation (3) becomes:

The above equation can be simply rewritten as:



Equation (4) is exactly the definition of a *union* merging operator, as defined in most existing FM merging methods [1, 2, 6, 8, 11, 12]. Therefore, the union merging operator is just a special case of the cross-join merging operator.

# 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.

## 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 that 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. The merging process relies on a set of rules defined in the following sub-sections.



1. Main steps of the implementation of the merging operator.

## Merge Feature Trees

In order to present the algorithm of merging feature trees, we first define a symbol for root features*.*

**Definition 5 (Root Feature**). Given an FM, *m*, we use the symbol *Root(m)* to denote the root feature of *m*.

Furthermore, as illustrated in the motivating example, we need to identify common and unique features of the input FMs. To avoid loss of generality, we introduce a matching function that determines the *equality* of features, as shown below.

**Definition 6 (Feature Matching Function)**. Given two sets of features, *F* and *G* (they can be the same set)*,* a feature matching function is a function , where if and only if the two features *x* and *y* are identical (it is also known as *x matches y*).

The matching function is an abstract function that must be specialized to a concrete function in actual merging. For example, most existing research on feature model merging assumes that the concrete function is just a comparison of feature names; that is, one feature matches another if they have the same name, which is obviously not very practical in real use. A better approach might be based on the comparison of feature similarity. However, the definition of such concrete functions is actually independent with the definition and even implementation of merging operators, and therefore it is out of the scope of this paper. We will keep using the abstract function afterwards. (However, we will discuss the choice of concrete functions in Section V.)

The algorithm is shown in Algorithm 1. The input feature trees are merged from the root recursively, with the help of a feature matching function, *h*. Firstly, if the root features do not match, the algorithm simply implements a union operator by creating a virtual root and putting input roots as its alternative children (a XOR group). It is a union operator because a product of the output FM is either a product of the first input FM or the second input FM, plus the virtual root. Since the virtual root is not a real feature, it can be simply removed from the products without any impact.

If input roots are matched, then the root of the output FM is created by cloning one of them (Line 7). Common children of input roots are merged recursively (Line 10). Note that the mismatch of roots would not happen in recursive executions of mergeTree. The refinement between the output root and the merged child is computed according to the input refinements (between input roots and their child) as well as the rules listed in Table II (Line 11). For example, if *c* is a mandatory child of *root1,* and it is an optional child of *root2*, then the merged *child* is an optional child of *root.* Finally, unique children of both input roots are simply appended to the output root as well as corresponding input refinements (Line 14 to Line 19).

|  |
| --- |
| Algorithm 1. Merge Feature Trees |
| mergeTree (root1: Feature, root2: Feature): Feature  **1** if () {  **2** ;  **3** virtual.appendChild(root1, XOR);  **4** virtual.appendChild(root2, XOR);  **5** return virtual;  **6** }  **7** root ← root1.copy();  // Merge the common children of root1 and root2.  **8**  **9** for each () {  **10**  **11**  **12** root.appendChild(child, ref);  **13** }    // Append the unique children of root1 and root2.  **14** for each () {  **15** root.appendChild(u1, root1.getRefinement(u1));  **16** }  **17** for each () {  **18** root.appendChild(u2, root2.getRefinement(u2));  **19** }  **20** return root; |

1. 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 |

It should be noticed that the rules only apply to common features that have a common parent (Line 9 to Line 11). In other words, recursive merging only happens on *hierarchically matched* features, as defined below.

**Definition 7 (Hierarchical Match).** Given two feature sets, *S1* and *S2,* and a feature matching function the two feature sets are hierarchically matched if

That is, if two features are matched, then their parent features are also matched. In the case that their parent features are *not* matched, Algorithm 1 leads to *feature clones*, which will be discussed in the next sub-section.

The rules only affect the hierarchically matched sub-trees of input feature trees (It is easy to see that the root of these sub-trees must be the root of the entire tree). In order to formalize the effect of the rules, we first define the concept of a restricted product and a restricted product set.

**Definition 8 (Restricted Product on a Feature Set).** Given a product of an FM, *p*, and a set of features, *S*, we define the restricted product of *p* on *S* as:

That is, a product is restricted on a feature set by removing the features that do not belong to the set.

**Definition 9 (Restricted Product Set on a Feature Set).** Given an FM, *m*, and a feature set, *S*, we define the restricted product set of *m* on *S* as:

With these definitions, the effect of rules can be expressed as the following equation.



where *H* is the maximal hierarchically matched sub-tree.

**Lemma 1.** Equation (5) holds when *H* is a tree with only one branch feature, that is, all child features of the root is a leaf feature.

**Proof.** Let , where *r* is the root feature. We prove the lemma by induction on *n.*

*H0* only contains the root feature. According to Line 7, Algorithm 1,

therefore Equation (5) holds.

The refinement between *r* and *f1* belongs to one of the following cases (*x* can be *m, m1* or *m2*):

* Mandatory, Xor, Or:
* Optional:

According to Table II, it is easy to show that Equation (5) holds.

We can similarly list the *RPS’s* as:

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

It is also easy to see that Table II ensures Equation (5).

Suppose Equation (5) holds for . When the refinement between *r* and *ck*+1 is one of the four kinds:

**Mandatory.**

**Xor.** We assume that *ck*+1 belongs to the Xor-group *G*, where , therefore

where .

**Or**. We again assume that *ck*+1 belongs to the Or-group *G,* then

**Optional**.

**QED**.

**Theorem 1.** Equation (5) holds for *H* of any structure.

**Proof.**

Theorem 2. Let *C* be common features of *m*1and *m*2*,* the following equation holds:

Proof. Let *H* be the maximal hierarchically matched sub-tree of *m*1 and *m*2. Therefore . We then let That is, *D* denotes those hierarchically mismatched common features.

For any feature we can show that by discussing *f*’s nearest ancestor *t* that Therefore

which implies that

QED.

## 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 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.



1. An example of hierarchical mismatch and feature clones.

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) is always identical, as if there is only one instance in the output FM.

## Merge Constraints

We only handle binary constraints (i.e. *requires* and excludes) in our implementation. We simply scan all input constraints and handle them according to the rules listed in Table III. Constraints involving at least a unique feature are copied to the output FM (Rule 1 and Rule 2). Otherwise the input constraints are merged (Rule 3 to Rule 8). A special case is shown in Rule 7, where there is a conflict between input FMs. In such a case we report the conflict and resolve the conflict by making the two features in the output FM become non-constrained (i.e. no constraint between them). These rules lead to a stronger property than Theorem 1, as stated in Theorem 2.

1. Rules of merging constraints

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Input 1** | **Input 2** | **Output** |
| *Unique Features* | | | |
| 1 | A requires B | *The FM does not contain A or B* | A requires B |
| 2 | A excludes B | A excludes B |
| *Common Features* | | | |
| 3 | A requires B | *No constraints between A and B* | *No constraints between A and B* |
| 4 | A excludes B |
| 5 | A requires B | A requires B | A requires B |
| 6 | A requires B | B requires A | A mutual-requires B |
| 7 | A requires B | A excludes B | *Report a conflict and set “No constraint between A and B”.* |
| 8 | A excludes B | A excludes B | A excludes B |

**Theorem 2.** Let *C* be common features of the FMs. That is, Then the output FM, *m*, satisfies:

Proof. Consider . is partitioned into three subsets: *S*1 *=* products without *f* and *g, S*2 = products with *g* but without *f, S*3 = products with *f* and *g.* There are three cases of *m,* that is, *f* requires *g, f* mutual-requires *g,* or no constraints between *f* and *g.*

Effect 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:

PartSet(m | Common)=PartSet(m\_1 | Common)∪PartSet(m\_2 ┤| Common),

where Common=FS(m\_1 )∩FS(m\_2 ). (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:

PartSet(m\_1│S)∪PartSet(m\_2│S)=PartSet(m\_4│S),

and PartSet(m\_1│S)∪PartSet(m\_3│S)=PartSet(m\_4│S).

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

## 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:

∀x(x∈PS(m\_i )→∃p∃y(p∈PS(m)∧y∈PS(m\_j )∧

x⊆p∧ p∖x⊆y), where i, j = 1, 2, i ≠ j.

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

F=FS(m\_1 )∪FS(m\_2 ).

We divide F into two subsets: one is the hierarchical matched common features of m1 and m2, denoted by H; the set difference D=F∖H, 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:

PartSet(m│H)⊇PartSet(m\_1│H)∪PartSet(m\_2│H). (7)

Second, we consider the partial product set of m on the feature set D. 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:

PartSet(m│D)={d\_1∪d\_2 ┤| d\_1∈PartSet(m\_1│D)∧d\_2∈PartSet(m\_2 |D)}. (8)

In addition, a product w can be express as:

w=Part(w│H)∪Part(w|D). (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 i≠j), and a product p of the output FM m, such that:

Part(x│H)=Part(p│H), (According to Formula (7).)

and Part(x│D)∪Part(y|D)=Part(p|D). (According to Formula (8).)

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

p =Part(p│H)∪Part(p│D)

=Part(x│H)∪Part(x│D)∪Part(y│D)

=x∪Part(y│D)

Thus we immediately get that x⊆p. In addition, we have:

p∖x=Part(y│D)⊆y. (According to Formula (9).)

Therefore Theorem 1 has been proved.

# 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.

## 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 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.

1. Summary of input FMs

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| SPLOT 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 |

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.



1. The result of merging 6 FMs of mobile phone domain (The root feature is omitted).

## 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 workload 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 humans.

# 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.

## Different Definitions of Merging Operators

Besides the cross-join operator 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) PS(m)⊇PS(m\_1 )∪PS(m\_2 ),

(Strict union) if the equality holds),

(Intersection) PS(m)=PS(m\_1 )∩PS(m\_2 ).



1. The motivating example revisited by union and intersection operators.

Compare with Union. The union merging operator is implemented in [1, 2, 6, 8, 11, 12]. 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, 6, 8, 11] 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 [12] 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 [9], 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, 11] eliminates all unique features of input FMs. However, it preserves all constraints of input FMs. Schobbens et al. [11] 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.

## 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.

Direct Mapping Approach. The algorithms by Hartmann et al. [8] and Schobbens et al. [11] 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. [6], and Segura et al. [12] are in this style, as well as ours. However, the implementation in [1] does not merge the cross-tree constraints. Both implementations in [6] and [12] 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 [7]. 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 [7] 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.

# 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 match features based on their names, descriptions, and structural characteristics.

##### 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. Benavides, D., Segura, S., Ruiz-Cortés, A., Automated analysis of feature models 20 years later: A literature review, Information Systems, Volume 35, Issue 6, September 2010, Pages 615-636, ISSN 0306-4379, 10.1016/j.is.2010.01.001.
6. 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.
7. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC 2007. (2007) 23–34.
8. Hartmann, H., Trew, T., Matsinger, A.: Supplier independent feature modeling. In: SPLC’09, IEEE Computer Society (2009) 191–200.
9. 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.
10. 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.
11. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. 51(2) (2007) 456–479.
12. 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.
13. 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.