Lecture Notes in Computer Science: Authors' Instructions for the Preparation of Camera-Ready Contributions to LNCS/LNAI/LNBI Proceedings

Alfred Hofmann[1,1], Brigitte Apfel[1], Ursula Barth[1], Christine Günther[1], Ingrid Haas[1], Frank Holzwarth[1], Anna Kramer[1], Leonie Kunz[1], Nicole Sator[1], Erika Siebert-Cole[1] and Peter Straßer[1],

[1] Springer-Verlag, Computer Science Editorial, Tiergartenstr. 17, 69121 Heidelberg, Germany
{Alfred.Hofmann, Brigitte.Apfel, Ursula.Barth, Christine.Guenther, Ingrid.Haas, Frank.Holzwarth, Anna.Kramer, Leonie.Kunz, Nicole.Sator, Erika.Siebert-Cole, Peter.Strasser, LNCS}@Springer.com

Abstract. The abstract should summarize the contents of the paper and should contain at least 70 and at most 150 words. It should be set in 9-point font size and should be inset 1.0 cm from the right and left margins. There should be two blank (10-point) lines before and after the abstract. This document is in the required format.

Keywords: We would like to encourage you to list your keywords in this section.

1 Introduction

In software reuse, feature models provide an effective way to organize and reuse requirements in specific software domains. The concept of feature models was first introduced in the FODA method [1]. The idea of feature models is to encapsulate requirements into a set of features and dependencies among features, and then to reuse these encapsulated requirements by selecting a subset of features from a feature model, while maintaining dependencies among the features. In order to maximize the effect of such reuse, an elementary task is to construct high quality feature models, which requires systematic analysis of commonality and variability in specific software domains.

As feature-oriented methods (e.g. software product lines) have been widely used in industries, they are now facing major scalability problems since real feature models with thousands of features are not uncommon. Especially, constructing such large

[1] Please note that the LNCS Editorial assumes that all authors have used the western naming convention, with given names preceding surnames. This determines the structure of the names in the running heads and the author index.

feature models can be a very complex activity. This problem reflects a need for methods and tools that feature model developers can use to reduce the complexity. One possible way is to avoid building a feature model from scratch; instead, a feature model can be constructed via reusing other feature models or part of them in the same domain. One way to implement such reuse is to merge the existing feature models into a new feature model, and the feature model developer can then start from the new feature model.

In this paper, we propose a generic algorithm to merge input feature models in the purpose of constructing a new feature model. The basic idea is to divide the merging process into two sub-processes: one is combining matching features (common features) in two input feature models; the other is weaving unique features of the inputs in a way that products of the new feature model are proper combinations of the unique features from both inputs. Besides, we also handle the constraints between the features in the inputs. One property of the merged feature model (the output) is that its products are combinations of valid products from both inputs. Another property is that the output feature model is well structured and is close to manually merged results.

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 an example of merging feature models. Related work is presented in Section 6. Section 7 describes future work and concludes this paper.

2 Preliminaries: Feature Model

Feature models (FMs) 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 [5]. 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.
- *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.
- *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.
- *Excludes.* If a feature *X* excludes a feature *Y*, both features cannot be included in the same product.

Given an FM, products can be configured from the FM by selecting and deselecting the features, while maintaining the relationships between them. For example, given an FM depicted in Figure 1, a mobile phone product with the following set of features can be configured:

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

3 The Merging Operation

In this section, we discuss the semantics of our merging operation, that is, the relation between input FMs and the output FM. We want to first give an example merging scenario, and show the requirements of the merging operation emerging from this scenario. Then we give the semantics which fulfills such requirements.

Figure 2 illustrates two input FMs to be merged, in the purpose of creating a new FM for FM developers to start their work.

3.1 Requirements

The example shown above highlights some requirements for the merging operation. First, if the same feature in two input FMs has been refined into unique child features, the expected result of merging should allow combination of child features from both inputs. For example, the feature *Screen* is refined in different perspectives in the inputs – from its *resolution*, and its ability of *touching*. In such scenario, one would expect that the different perspectives can be combined so that products like *high resolution touch screen* can be configured from the output FM.

Second, the combination of unique child features should preserve original constraints among these features in the inputs. For example, the original constraints in the inputs ensure 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 output FM should disallow products like *high resolution screen with both touch and non-touch ability.*

3.2 Semantics

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

$$[[Input\ 1]] \otimes [[Input\ 2]] \subseteq [[Result]],$$

where the *cross product operator* 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 products of the output FM are combinations of valid products of both inputs. Therefore it fulfills the requirements mentioned before: first, different refinements of the same feature can be combined; second, original constraints can be preserved. For example, the cross product of the sub-tree rooted *Screen* in inputs leads to:

[[*Input 1*]] = { {*Screen, High Resolution*}, {*Screen, Low Resolution*} },

[[*Input 2*]] = { {*Screen, Touch*}, {*Screen, Non-touch*} },

[[*Result*]] = { {*Screen, High Resolution, Touch*}, {*Screen, High Resolution, Non-touch*}, {*Screen, Low Resolution, Touch*}, {*Screen, Low Resolution, Non-touch*} }.

This is exactly the expected result.

4 The Merging Algorithm

In this section, we present the merging algorithm which is the implementation of the merging operation discussed before. We first give an overview of the algorithm, and explain each step in details.

4.1 Overview

Figure 3 gives an overview of the merging algorithm. The algorithm can be divided into two main steps: merge the refinements and merge the constraints. The main steps are rule-based and automated. Besides, the algorithm needs a pre-processing and a post-processing, in order to generate highly human-readable hierarchical structure of the result FM.

4.2 Preprocessing: Mark the Semantics for Refinements

Most feature modeling methods categorize refinements into four forms according to variability: *mandatory, optional, xor-group* and *or-group*, as introduced in Section 2. However, in our previous work on feature modeling method [11], we find that the refinements can also be classified in three kinds according to its 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.

In the merging of feature models, we find that the three kinds of semantics are extremely helpful for merging unique features in the input FMs. (Details are described in Section 4.4.) Furthermore, the classification by semantics is orthogonal to the traditional classification by variability. Therefore, we introduce a preprocessing step

to ask developers to mark semantics for refinements in the inputs. Figure 5 shows the result of preprocessing of the feature model introduced in Section 2.

4.3 Merge the Refinements

The merging of refinements is a recursive algorithm (see Figure 6). First of all, the root feature of both inputs must be the same (with the same name). The root feature of output FM is created first, and then common children and unique children of the root in the inputs are handled, respectively. A common child will be merged recursively, and the merged child is appended to the root with a newly calculated refinement. The unique children left will be merged according to the three kinds of semantics of refinements marked in the preprocessing step. Finally, the root is returned by the algorithm.

Algorithm 1: Merge the tree structure

```
merge (root1: Feature, root2: Feature): Feature
   □ The root of the input FMs must be matched
   if not-match (root1, root2) then return null

   newRoot ← root1.copy()

   □ Merge the common children of root1 and root2. (See Section 4.5.)
   for each child ∈ (root1.children() ∩ root2.children())
      □ Merge the child of the inputs recursively
      newChild ← merge (child of root1, child of root2)

      □ Then calculate the new refinement relationship between newRoot and child,
          according to the corresponding refinement relationships in the inputs
      newRefinement ← calcNewRefinement (root1, root2, child)

      □ Finally, add the newChild to newRoot with the newRefinement relationship
      newRoot.appendChild (newChild, newRefinement)
   end

   □ Merge the unique children of root1 and root2 by the semantics of the
       refinements. (See Section 4.4 for details.)

   return newRoot
end
```

4.4 Merge 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 rules for each combination. For example, in Figure 5(a), merging two refinements with semantics of

decomposition should also generate decomposition; in other words, merging the same feature with different *parts* generates a feature with all the parts from both inputs. However, in Figure 5(b), merging two different *specialization* of the same feature needs clarify the perspectives of the specialization (in the example, one perspective is from *resolution*, and another is from *touch-ability*), and then the result should treat the perspectives as *characteristics*. Besides, In Figure 5(c), merging some *characteristics* of a feature and some *specialization* of the feature generates a feature with one more characteristic according to the specialization. The next three rules can be explained in a similar way.

The last rule shown in Figure 5(g) means that if the root has no child in one of the inputs, the output is just a copy of another input. It also preserves the cross product semantics of merging operation.

In the situations described in Figure 5(b) and 5(c), 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, and the post-processing step will ask the developer 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 original form of refinement relationships in the inputs are kept in the output, so that the combination of children from both inputs does not violate original constraints in the children. Besides, the rules ensure that the product set of output FM is exactly the cross product of input FMs' product sets, so the semantics of merging operation is preserved.

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

As shown in Table 1, we define the output refinement according to each combination of the forms of refinements in the inputs. There are totally 10 possible combinations. Each item in Table 1 can be considered as a rule, and Figure 6 gives an example of one rule. Reasoning of the rules is based on the cross product semantics of our merging operation.

4.6 Merging the Constraints

We defined a set of rules to merge the cross-tree constraints. We handle two kinds of constraints, namely *requires* and *excludes*. Each constraint in the input FMs is checked by the rules to generate output. Table 2 gives definition of the rules, and

reasoning of the rules is also based on the cross product semantics of merging operation. For example, the fourth rule defines that if one input FM contains *A requires B* but another input FM contains only the feature *B*, then the output FM contains *A requires B*. (The merging of tree structure ensures that the output FM has already contained both feature *A* and *B*.) We can check the product sets as follows:

[[*Input 1*]] = { {*A, B*}, {*B*} },
[[*Input 2*]] = { {*B*} },
[[*Output*]] = { {*A, B*}, {*B*} } = [[*Input 1*]] ⊗ [[*Input 2*]].
Therefore the cross product semantics holds.

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. An example can be found in Figure 6.

4.8 Other Issues

We discuss two related issues of the merging algorithm in this sub-section. Although these issues do not affect the execution of our algorithm, they need to be addressed for better understanding and using of the algorithm.

Feature Renaming. In the merging algorithm, we treat feature name as the identifier of features. When merging two FMs, two situations may happen: two features with the same name in both inputs express different meanings, or two features with different names in both inputs have the same meaning. We provide a *renaming* operation that allows developers handle such situations before merging.

Parent Compatibility. Two FMs are said *parent compatible* if the same feature has the same parent feature in both inputs. If the input FMs are not parent compatible (in other words, there are *hierarchy mismatch* between them [5]), our algorithm still works as usual and the output FM will contain feature clones (the same feature appears more than once in an FM). Two typical cases are shown in Figure 7. Since our algorithm allows feature clones, this is not a problem. However, some existing feature model merging algorithms [4] [6] do not allow clones, and therefore they assume the input FMs are always parent compatible.

We believe that a perfect merging of parent-incompatible FMs needs significant refactoring of the tree-structure of the inputs, and such refactoring can only be done case-by-case. Therefore, as a generic feature model merging algorithm, we do not handle such situation specifically. We generate output FMs with feature clones and leave the clones to be refactored manually by developers.

5 An Example

6 Related Work

Several previous works propose some kinds of FM merging operation. The comparison with these works can be done in two dimensions: semantics and implementation of the merging operation. A summary of these works is shown in Table 3.

6.1 Semantics

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

Union: [[*Result*]] ⊇ [[*Input 1*]] ∪ [[*Input 2*]].
Intersection: [[*Result*]] = [[*Input 1*]] ∩ [[*Input 2*]].

Compare with Union. Most existing merging algorithms [1, 2, 4, 5, 6, 7] implement the union semantics. The reason is that their algorithms need to be used in a scenario where products of input FMs have been derived and have been running *before* the merging, and the output FM is intended to replace the input FMs. Therefore keeping the original products untouched is critical, and the union semantics suits the scenario. By contrast, in the scenario described in this paper, the concern is to build a new FM by reusing existing FMs, but not to replace the existing FMs, so the preservation of original products becomes not so important, and the ability to properly combine features of input FMs becomes critical. In such situation, the cross product semantics is better than union semantics, especially when input FMs contains unique feature. Figure 8 gives an example in which some algorithms [1, 4, 5, 6, 7] do not support feature combination (in Figure 8(c)), and some [2] allows violation of original constraints (in Figure 8(d), a *screen* can have both *high* and *low resolution*.) Only our algorithm gives expected result (in Figure 8(f)).

Compare with Intersection. The intersection semantics implemented in [3, 8] is also unsuitable for the scenario described in this paper, because it will eliminate the unique features of the inputs completely (see Figure 8(e)).

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 Algorithms. The algorithms by Schobbens et al. [1] and Hartmann et al. [2] are in this style. The idea is to put input FMs side-by-side and add proper constraints between them to make the semantics of merging operation satisfied. In other words, the input FM can be directly mapped into a certain part of the output FM. Compared to our approach, their major advantage is that their algorithms are easier to implement. However, the quality of the output FM in their algorithms is not satisfying, because there are lots of redundancies (each common feature appears at least twice in the output FM) and more importantly, the constraints between the features cannot be easily seen by looking at the output. Therefore a significant amount of modifications on the output is needed in developers' further work. By contrast, the output FM in our algorithm is close to manually merged result, so the work load of FM developers is reasonably low.

Rule-based Algorithms. The algorithms by Segura et al. [4], Archer et al. [5] and Broek et al. [9] are in this style, as well as our algorithm. The algorithm in [5] does not merge the cross-tree constraints. Both algorithms in [4] and [9] require that the input FMs are parent compatible, which confines their use because incompatible parents are not uncommon in real world. In addition, as mentioned before, the major problem of these algorithms is that their merging operation holds the union semantics, and it is not so suitable for the scenario described in this paper.

Logic-based Algorithms. Archer et al. [10] propose an algorithm in which the input FMs are transformed into logical formulas [6], and then calculate the logical formula of output FM to satisfy the semantics of merging operation, and finally transform the logical formula back to FM [7] to get the result. Compared with our approach, there are three 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 inputs, while our algorithm is polynomial, so the scalability of their algorithm is doubtful. Third, transforming a logical formula to an FM [7] 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 with the same logical formula). Therefore a considerable amount of refactoring work is needed after the merging.

7 Conclusion

References

1. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. 51(2) (2007) 456–479

2. Hartmann, H., Trew, T., Matsinger, A.: Supplier independent feature modeling. In: SPLC'09, IEEE Computer Society (2009) 191–200
3. 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
4. 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
5. 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
6. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC'05. Volume 3714 of LNCS. (2005) 7–20
7. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC 2007. (2007) 23–34
8. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. SIAM Journal on Computing, 1(2):131–137, 1972.
9. 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.
10. Acher, M., Collet, P., Lahire, P., France, R. Managing multiple software product lines using merging techniques. 2010.