A BDD-Based Approach to Verifying Clone-Enabled Feature Models' Constraints and Customization

Wei Zhang[1,2], Hua Yan[1,2], Haiyan Zhao[1,2], and Zhi Jin[3]

[1] Key Laboratory of High Confidence Software Technology, Ministry of Education of China
[2] Insititute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
[3] Chinese Academy of Sciences, Beijing, China
{zhangw,yanhua07,zhhy}@sei.pku.edu.cn, zhijin@amss.ac.cn

Abstract. In this paper, we present a kind of semantics for constraints in clone-enabled feature models, which resolves the problem of what kinds of constraint should be added to a feature model after some features are cloned. The semantics is composed of two patterns: the *generating* pattern and the *adapting* pattern, to address the two problems of what kind of constraints should be imposed on a clonable feature and its clones, and how an existing constraint should be transformed in the context that features involved in the constraint are cloned, respectively. After that, we propose a BDD-based approach to verifying clone-enabled feature models, an approach that makes efficient use of the BDD (*binary decision diagram*) data structures, by considering the specific characteristics of feature models' verification. Experiments show that this BDD-based approach is more efficient and can verify more complex feature models than our previous method.

Keywords: Feature models, Clonable features, Constraints, Customization, Verification.

1 Introduction

Feature models have been recognized as an important technique to capture and organize the reusable requirements in a specific software domain [7,8,5,2,1,9,13]. One important purpose of feature models is to facilitate the reusing of these reusable requirements, and this purpose is usually achieved by using a customizing-based approach. That is, when developing a new application in a software domain, you do not need to elicit and analyze the application's requirements from scratch, but can just customize the domain's feature model (selecting a subset of features from it), and use the customizing result as a starting point for the application's requirements engineering activity.

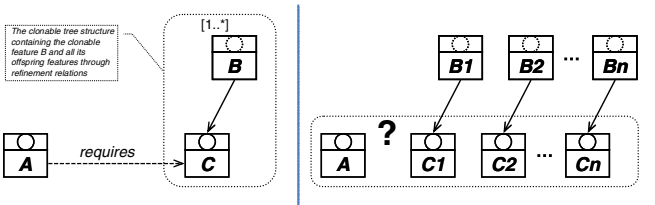
One problem in a feature model's customization is the verification problem [9]. This problem is caused by the fact that not any subset of features from a feature model is a valid customizing result. Usually, there are constraints among features, and a valid customizing result must satisfy all these constraints. For this reason, when a customizing

H. Mei (Ed.): ICSR 2008, LNCS 5030, pp. 186–199, 2008.

decision[1] is made on a feature model, we need to verify that those constraints among features are not violated by the decision (namely, *the verification of feature models' customization*). Otherwise, the inappropriate decision will be propagated implicitly to latter customizing activities, and thus decrease the efficiency of customization. In addition, before customization, we should first ensure the correctness of constraints among features (namely, *the verification of feature models' constraints*).

The difficulty of the verification problem is caused by its *NP-hard* nature. In essence, the verification of feature models is a constraint satisfaction problem (CSP), and researchers have recognized that the CSP is an *NP-hard* problem in general [11]. In our experience, when a feature model contains a large number of features with a complex set of constraints among them, the verification using a third-party's model checker usually consumes an intolerable period of time, or even runs into a live-lock state. The *NP-hard* nature makes it difficult to find an efficient way to solve the verification problem of feature models.

Another problem relating to the verification of a feature model's customization is caused by the introduction of clonable features into the feature model[2]. In customization, the tree structure containing a clonable feature and all its offspring features can be cloned into many copies, and each copy can be customized individually. The problem caused by clonable features is that some constraints among features will lose their original semantics after a feature is cloned [3]. As a result, we will lose the capability of verifying whether a customizing result is a valid one based on the constraints among features.



(a) Feature model with three features *A*, *B* and *C*. *B* is a clonable feature and has a child feature C. A and C are optional features. There is a constraint: *A requires C*.

(b) If *B* is cloned into a set of features: *B1 B2, …, and Bn*, then, what is the semantic of the original constraint "*A requires C*"?

Fig. 1. The semantic-losing problems caused by clonable features: an example

An example of the problem is depicted in Fig. 1 (see Table 1 for the exact meaning of the symbols). The constraint "*A requires C*" means that if *A* is bound (i.e. selected) in a customizing result, then *C* should also be bound in it. In customization, if *C* is cloned into a set of clones: *Ci* (*i = 1, 2, …, n*), how should the constraint "*A requires C*" be adapted to these clones? Should the binding of *A* require or be independent of the binding of these clones.

According to the two problems above, the main contributions of this paper are two-fold. For the semantic-losing problem, we present a kind of semantics for constraints

[1] A customizing decision on a feature model means deciding whether to make a feature remaining in the customizing result (*binding a feature*) or to remove the feature from the result (*removing a feature*).

[2] In this paper, a feature model with clonable features is called a clone-enabled feature model.

in clone-enabled feature models. For the verification problem, we propose a BDD-based approach to verifying both feature models' constraints and customization, an approach that makes efficient use of the BDD (*binary decision diagram*) data structures based on the specific characteristics of feature models' verification.

The rest of this paper is organized as following. Section 2 introduces some preliminary knowledge. Section 3 presents the semantics for constraints in clone-enabled feature models. Section 4 proposes the BDD-based approach to verifying feature models. Related work is discussed in Section 5. Finally, Section 6 concludes this paper with a short summary.

2 Preliminary

In this section, we first introduce a notation for clone-enabled feature models, and a propositional logic based definitions of constraints among features. After that, we clarify a fact about clonable features, that is, there is actually a clonable structure related to each clonable feature.

2.1 A Notation for Feature Models

Table 1. Symbols in the notation for feature models

Symbol	Name	Explanation
	A *mandatory feature* with the name "*X*".	A mandatory feature must be selected in a customizing result, if its parent feature is selected. If its parent is removed, it must also be removed. If it hasn't a parent feature, then it must be selected in any customizing result.
	An *optional feature* with the name "*Y*".	An optional feature can either be selected in or be removed from a customizing result, if its parent feature is selected or it hasn't a parent. If its parent is removed, it must also be removed.
	A feature that can be either mandatory or optional.	In our presentation, we use this symbol to denote a feature that can be replaced by either a mandatory feature or an optional feature.
	A *feature reference*.	A reference to a feature that has the name Z.
[*a*..*b*]	A symbol for *clonable* features.	When the symbol is placed at the top of a feature, it means that the feature is clonable. In the symbol, *a* and *b* are two integers satisfying the property: $0 < a \leq b$, and the meanings is that the number of the clonable feature' clones should not less than *a* and not greater than *b*.
	A *refinement* relation between two features.	A refinement relation connects two features. The feature connecting to the non-arrow end is called the *parent* of the feature connecting to the arrow end. A feature can only have one parent feature at most.
	A *refinement path*	In our presentation, we use this symbol to denote a path containing one or more refinement relations, and zero or more features. Each feature connects to two different refinement relations' arrow and non-arrow ends, respectively.

Table 1. (*continued*)

	A *requires* constraint between two features.	A *requires* constraint connects two features. The feature connecting to the non-arrow end is called the *requirer*, and the other the *requiree*. This constraint means that if the *requirer* is bound in a customizing result, the *requiree* also be bound.
	An *excludes* constraint between two features.	An *excludes* constraint connects two features. This constraint means that the two features should not be both bound in a same customizing result.
	A *binding predicate* among a set of features and binding predicates	The left end connects to a *composite* constraint or to one of the right ends of a binding predicate. The right ends connect to a set of features and binding predicates, respectively. We define three *types* of binding predicate: *and* (denoted by $\wedge$); *or* (denoted by $\vee$); *xor* (denoted by ***1***). See **Table 2** for the formal definition of binding predicates.
	A *composite* constraint between two binding predicate	We define two *types* of composite constraint: *requires* (denoted by $\longrightarrow$); *excludes* (denoted by $\nrightarrow$). See **Table 3** for their formal definition.

Table 2. The formal definition of binding predicates. In this table, A and B denotes features, and p and q denotes binding predicates. For a feature F, bind(F) is a predicate; it is true if F is bound, and false if removed. In our notation, we only use binding predicates as constituent parts of the composite constraints, but not use them to represent individual constraints.

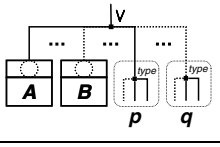
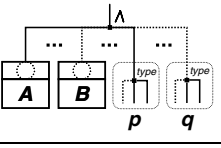
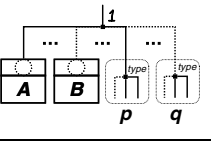
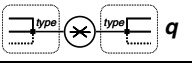
	or(*A*, …, *B*, …, *p*, …, *q*)	*and*(*A*, …, *B*, …, *p*, …, *q*)	*xor*(*A*, …, *B*, …, *p*, …, *q*)
Binding Predicate			
Formal Definition	$bind(A)\vee...\vee\neg bind(B)$ $\vee...\vee p\vee...\vee\neg q$	$bind(A)\wedge...\wedge\neg bind(B)$ $\wedge...\wedge p\wedge...\wedge\neg q$	$bind(A)\otimes...\otimes\neg bind(B)$ $\otimes...\otimes p\otimes...\otimes\neg q$

Table 3. The formal definition of composite constraints. In this table, *p* and *q* denotes binding predicates. In the situation that *p* and *q* only contains one feature, the two types composite constraints becomes the *requires* and the *excludes* constraints between two features.

Composite Constraint	*requires*(*p*, *q*)	*excludes*(*p*, *q*)
		
Formal Definition	$p \rightarrow q$	$p \rightarrow \neg q$

2.2 The Clonable Structure Related to a Clonable Feature

A clonable feature does not mean that only the feature itself can be cloned into many copies. Usually, it means that a structure related to the clonable feature can be cloned into many copies. The structure is formed from three kinds of element: the clonable feature, all its offspring features, and all the refinement relations between these features.

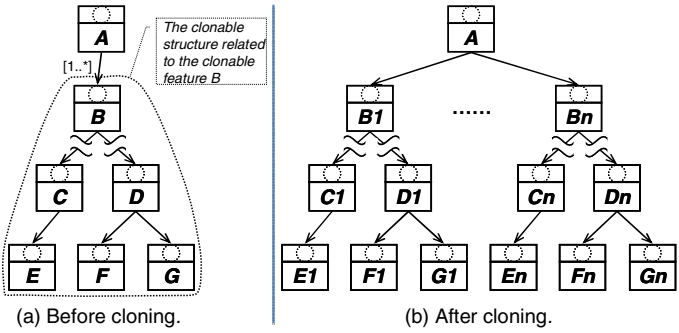


Fig. 2. The clonable structure related to a clonable feature: an example

Such a structure is exemplified in Fig. 2 (a), in which, feature *B* is clonable, and the dashed shape shows the clonable structure related to *B*. In customization, the cloning of *B* actually leads to the cloning of the related structure, and after cloning, each clone of *B* becomes a child feature of *B*'s parent feature *A* (see Fig. 2 (b)). For any feature in a clonable structure, the property whether it is mandatory or optional is not changed after cloning. In the rest of this paper, we use *the cloning of a clonable feature* to denote the meaning of *the cloning of the clonable structure related to the clonable feature*. For a clonable feature *F*, we use *cs(F)* to denote the set that contains all the elements in the clonable structure related to *F*.

3 Semantics for Constraints in Clone-Enabled Feature Models

In this section, we present a kind of semantics for constraints in clone-enabled feature models. The semantics is composed of two patterns: the *generating* pattern, and the *adapting* pattern. The former handles the problem of what kind of constraints should be imposed on a clonable feature and its clones. The latter deals with the problem of how an existing constraint should be adapted in the context that some features involved in the constraint are cloned. Before giving more details about the semantics, we first introduce a description structure for the two patterns' definitions.

3.1 A Description Structure for the *Generating* and the *Adapting* Patterns

Table 4 shows the components contained in the description structure and the descriptions of these components.

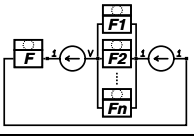
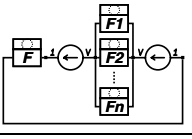
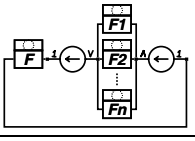
3.2 The *Generating* Pattern

One question related to a clonable feature is whether we should impose any constraint on the feature and its clones. In this paper, we adopt a positive answer to this question. We treat the relation between a feature and its clones as the *type-instance* relation. One understanding of a type is that it is a set consisting of all the type's instances. Based on this understanding, we can derive that, if a type is removed, any of its instances should also be removed. However, this understanding does not tell us how many instances should be bound if the type is bound.

Table 4. Components in the description structure

Components		Description
Pattern Name		*A meaningful name for a pattern.*
Context	Clonable Feature	*A clonable feature.*
	Cloned Features	*All the clones of the clonable feature.*
	Source Constraint	*An existing constraint that will be transformed by the pattern.*
Trigger Condition		*A condition satisfied by components in the context. If the trigger condition is true, the pattern must be applied, that is, transforming the source constraint into the target constraints.*
Target Constraints		*Constraints transformed from the source constraint.*

Table 5. Definition of the *generating* pattern

Pattern Name		*Generating*		
Context	Clonable Feature	*F*		
	Cloned Features	*Fi*, (*i* = *1*, *2*, …, *n*)		
	Source Constraint	*<Empty>*		
Trigger Condition		*true*		
Target Constraints		*Case 1: single- binding*	*Case 2: multi-binding*	*Case 3: all-binding*
				

Based on the analysis above, we developed the *generating* pattern (see Table 5), to address the problem of what kind of constraints should be imposed on a clonable feature and its clones. The name "*generating*" means that some constraints are generated after the cloning of a clonable feature. The *generating* pattern defines three cases of generated constraints. For a clonable feature *F*, if it is bound, the *single-binding* will require that exactly one of its clones should be bound, the *multi-binding* will require that one or more clones should be bound, and the *all-binding* will require that all it clones should be bound. If *F* is not bound, all the three cases will require that none of its clones can be bound.

Since there are three kinds of target constraints in the pattern, a related question is which kind should be selected when applying the pattern. We think that the question should be answered according to more specific semantics related to each clonable feature. A special situation is that: when a clonable feature is mandatory, only the *all-binding* target constraints are suitable. Otherwise, the mandatory feature may need to be changed into optional.

3.3 The *Adapting* Pattern

In a clone-enabled feature models, a problem related to a constraint is that how the constraint should be adapted in the context that one or more features involved in the constraint belong to a clonable structure and that the structure is cloned.

Table 6. Definition of the *adapting* pattern

Pattern Name		*Adapting*
Context	Clonable Feature	F
	Cloned Features	Fi, ($i = 1, 2, \ldots, n$).
	Source Constraint	$const(A, B, \ldots, C, D, \ldots, E)$: a constraint among a set of features.
Trigger Condition		$\{A, B, \ldots, C, D, \ldots, E\} \cap cs(F) = \{A, B, \ldots, C\} \neq \varnothing$
Target Constraints		$const(A, B, \ldots, C, D, \ldots, E) \wedge$ $(\wedge_{i=(1,2,\ldots n)} (bind(Fi) \rightarrow const(Ai, Bi, \ldots, Ci, D, E, \ldots, F)))$

For this problem, we introduce the *generating* pattern (see Table 6). The name "*adapting*" means that some existing constraints should be adapted after the cloning of a clonable feature F. The target constraints defined in the *adapting* pattern contains two parts. The first part contains exactly the source constraint, which means the source constraint is still maintained after the cloning of F (this is an important characteristic of the *generating* pattern). The second part contains a set of constraints for each of the clones of F, respectively. For each clone Fi, the constraint requires that if Fi is bound, then the original constraint should also be satisfied by replacing each feature X in the constraint that belongs to $cs(F)$ with its clone Xi.

4 BDD-Based Verification of Feature Models

As we can see in Section 3, after a sequence of clone transformations, even simple binary constraints (i.e. *requires* and *excludes*) could be transformed into complex *composite* constraints. This further increases the difficulty of feature models' verification.

In this section, we present a BDD-based approach to verifying feature models. First, we introduce three verification criteria, which are proposed in our previous work [13], and have been proven to be effective in detecting deficiencies in feature models [14]. Base on the three criteria, we proposed a BDD-based algorithm that can check the three criteria's satisfiability by only traversing once to the nodes in a BDD (*binary decision diagram*). We also provide two strategies to improve the efficiency of creating a feature model's BDD. Experiments show that this approach is more efficient and can verify more complex feature models than our previous method.

4.1 Three Criteria for Feature Models' Verification

From the viewpoint of feature models' verification, a feature model can be abstracted into a set of features and a set of constraints among features [13]. According to a

feature's binding state, features in a feature models can be partitioned into three sets. The *bound* set contains exactly all the features having been bound, the *removed* set contains exactly all the features having been removed, and the *undecided* set contains all the other features which will be bound or removed in later customizing activities. A customizing decision to an undecided feature either binds the feature or removes it.

Given a feature model, if any of the following three criteria is not satisfied, there must be errors or deficiencies either in the constraints among features or in the customizing decisions to features [13].

Criterion 1: There exists at least a set of customizing decisions to all features in the *undecided* set that will not violate any constraints among features.

Criterion 2: Each feature in the *undecided* set has a chance to be bound, without violating any constraints among features.

Criterion 3: Each feature in the *undecided* set has a chance to be removed, without violating any constraints among features.

Von der Maßen and Lichter [15] have created a deficiency framework for feature models. Our previous investigation [14] shows that the three criteria can detect most kinds of anomaly and inconsistency among constraints at an early stage (i.e. before customization). Further details about the three criteria and the deficiency framework can be found in [13,14,15].

Although the three criteria are very effective, the checking of them is not easy. *Criterion 1* is a binary *CSP*, and the time complexity of its checking is $O(2^n)$, where n is the number of features in the undecided set. For each undecided feature, *Criterion 2 and 3* can also be easily transformed into two binary CSPs with the time complexity of $O(2^n)$, respectively. That is to say, the three criteria's checking could be transformed into the checking of $2n+1$ binary CSPs, and the total time complexity would be $O(2^n+2n\cdot 2^n)$, which equals to $O((2n+1)\cdot 2^n)$.

4.2 BDD-Based Checking Algorithm for the Three Criteria

Although the three criteria's checking could to be transformed into the checking of $2n+1$ CSPs, there is a shortcoming in such an approach, that is, it treats the $2n+1$ CSPs as independent problems, without considering the connections between these problems. In fact, we could find that the $2n+1$ CSPs are very similar; the only difference between them is that a different undecided feature's binding state is assigned to *bound* or *removed*. If the similarity could be fully explored, the time complexity would be further decreased.

Based on this observation, we investigate the BDD technique and find an algorithm that can check the three criteria's satifiability by only traversing once to the nodes in a BDD. Before giving more details about the algorithm, we first give a short introduction to BDDs.

In general, a BDD is a compact data structure for representing a Boolean function [6]. Fig. 3 shows an example of BDDs. We can see that a BDD is composed of multiple layers, each layer contains a set of nodes related to a propositional variable, and each node connects to right layers' nodes through a *true* branch or a *false* branch, which means that the node is assigned the value of *true* or *false*, respectively. The

rightmost layer contains two nodes of *true* and *false*, which denotes the Boolean function's two possible value. A path from the leftmost node to the *true* node means that the function's value is *true* in the value assignment indicated by the path, and a path to the *false* node means the function's value is *false*.

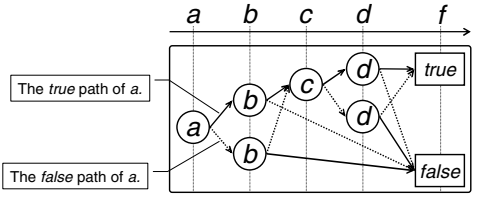


Fig. 3. The BDD representation of Boolean functions: an example. This show a BDD of the Boolean function: $f = (a \leftrightarrow b) \wedge (c \leftrightarrow b)$, where, *a*, *b*, *c*, and *d* are four propositional variables. The path "*a true*, *b true*, *c false*, *d false*, *true*" means that, in the following value assignment: *a=true*, *b=true*, *c=false*, and *d=false*, the function *f*'s value is *true*. Similarly, the path "*a false*, *b true*, *false*" means that *f*'s value is *false* in the value assignment indicated by the path.

Now, we explain how to check the three criteria's satisfiability efficiently, in the context that the set of constraints among features are transformed into a BDD[3]. For *Criterion 1*, the checking method is simple; if there is a node whose *true* path or *false* path connects to the *true* node, then this criterion is satisfied. For *Criterion 2* and *3*, we use the idea illustrated in Fig. 4 to check their satisfiability.

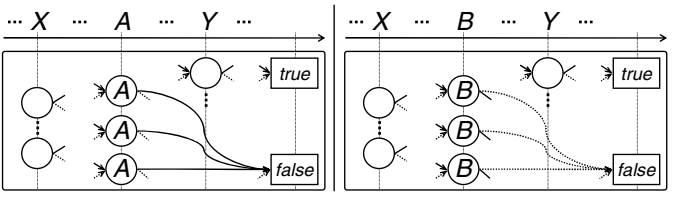


Fig. 4. The idea to check the satisfiability of *Criterion 2* and *3*. For a feature *A*, in order to check whether it has a chance to be bound, we only need to examine whether all the *true* paths of A's nodes connect to the *false* node (see the left part). The answer *yes* means that *A* has no chance to be bound, and the answer *no* means *A* still has the chance. Following the same idea, we can check whether a feature has a chance to be removed. The only difference is to examine whether all the *false* paths of the feature's nodes connect to the *false* node (see the right part).

To realize the idea above into an algorithm, we have to consider the situation that a BDD contains crossing paths. A crossing path eliminates some nodes from a BDD in order to maintain the BDD's compactness. We need to recover those eliminated nodes, before applying the idea above. Fig. 5 shows an example of this situation.

Based on the general idea and the special situation, we develop the following algorithm to check the satisfiability of *Criterion 2* and *3*, an algorithm that take a breadth-first traversal to a BDD's nodes.

[3] See section 4.3 for how to transform a set of constraints among features into to a BDD.

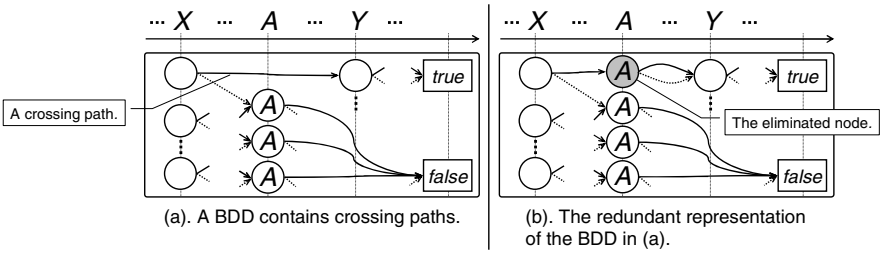


Fig. 5. A BDD containing crossing paths and its redundant representation. In (a), although all the *true* paths of A's nodes connect to the *false* node, *A* still has a chance to be bound. This is caused by the crossing path that eliminates a node of *A*. If recovering the eliminated node, we can get a redundant representation of the BDD. In the redundant BDD, there is a node of A, whose *true* path does not connect to the *false* node, and thus *A* still could be bound.

A BDD-based algorithm for *Criterion 2* and *3*. Where, *get_true_branch(Node e)* returns *e*'s child node through the *true* path, *get_false_branch(Node e)* returns *e*'s child node through the *false* path, and *isNonCrossedLayer(Layer layer)* returns whether the *layer* is crossed by any crossing paths. The three functions are created in a BDD's construction.

```
Input: The root of BDD
Output: A set that contains all the features violating Crite-
rion 2 or 3.

Verify(root){
   Set violatedFeatures :=  ∅;
   for (layer = getLayer(root) to getLayer(0)){
       If (isNonCrossedLayer(layer)=true){
           isCriterion1Violated := true;
           isCriterion2Violated := true;
           for each node e of layer{
               if (get_true_branch(e)!=false_node)
                  isCriterion1Violated := false;
               if (get_false_branch(e)!=false_node)
                  isCriterion2Violated := false;
           }
           if (isCriterion1Violated = true ||
               isCriterion2Violated = true    ){
            featureName = getFeatureName(layer)
              violatedFeatures.add(featureName);
           }
       }
   }
    return violatedFeatures;
}
```

4.3 Constructing a BDD for a Feature Models

Constructing a BDD for a feature model is to transform the conjunction of constraints in the feature model into a BDD. There are two issues to be considered:

1. How to get a BDD with a smaller size?
2. How to ensure that the constructing process consume less memory space?

We adopt two strategies to deal with the two issues.

Strategy 1: Use the order of the depth-first traversal to feature trees as the variable order of BDD.

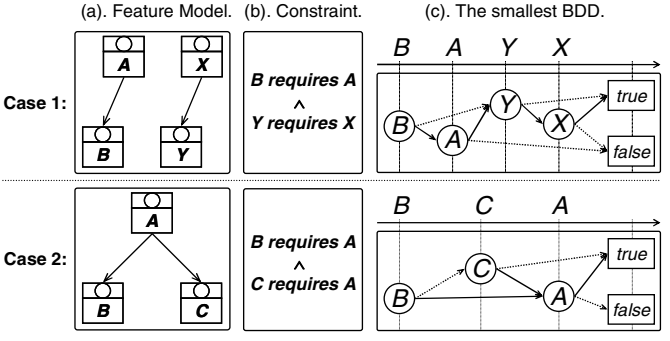


Fig. 6. The smallest BDDs in two basic cases. In case 1 with two feature trees, a smallest BDD has a variable order, in that, any child feature precedes its parent or the inverse, and variables belonging to different feature trees do not mix. In case 2 with a parent feature and its two children, a smallest BDD has a variable order, in that, the parent is the last or the first variable. A depth-first order to feature trees (whether in pre-order or post-order) can satisfy both of the two cases. The analysis above can also apply to feature models with multiple feature trees, in each of which, there may be multi-levels of features, and a feature may have three or more children.

This strategy is concluded from two basic cases in feature models (see Fig. 6). In the two cases, we only consider the feature trees (formed by features and refinement relations between them) in feature models. As recognized in our previous research [13], for a refinement between two features, there is a constraint: *child requires parent*. Based on these constraints, we find that it leads to a smallest BDD by using the variable order generated from the depth-first traversal to feature trees.

Strategy 2: Construct BDDs for each of the feature trees and for each of the constraints in a feature model, then combine these BDDs into the final BDD.

The purpose of this strategy is trying to decrease the possibility that the intermediate results in BDDs' construction consume huge memory space. The idea behind this strategy is to avoid considering too many constraints at one time. For the words limitation, we will not give further details about this strategy.

4.4 Experiments

To examine the approach's efficiency and capability, we apply it to verify two sets of designed feature models. One set contains 20 feature models only with binary constraints, and the number of features in them varies from 10, 20 to 90, and then from 100, 200, to 1000. The other set contains 20 feature models with both binary and composite constraints, and the number of features also varies from 10 to 1000. We

also use the same two sets of feature models to examine the effect of our previous SMV-based method for feature models' verification [13], a method which transforms feature models' verification into *2n+1* independent CSPs, and uses the model checker SMV [10] to verify these CSPs. The environment for our experiments is a notebook with a 2.0G HZ CPU, 512 MB memory, and a Windows XP OS.

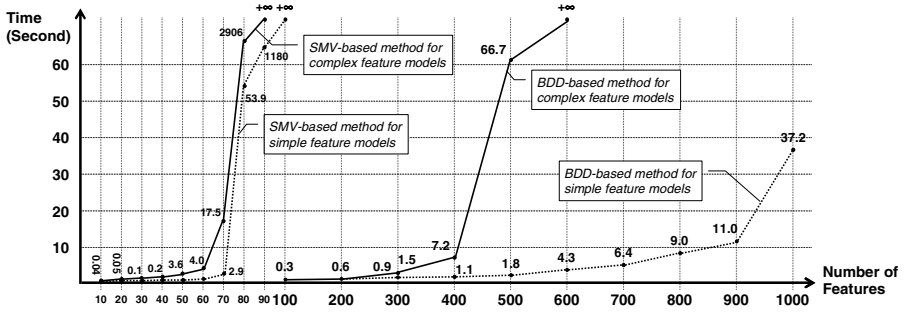


Fig. 7. The result of experiments

Fig. 7 shows the result of our experiments. We can see that, the SMV-based approach can not handle feature models with more than 100 features, while for feature models with 100 features, the BDD-based approach only needs a time less than one second. Furthermore, the BDD-based approach can verify complex feature models with 500 features using 66.7 seconds, and verify simple feature models with 1000 features using 37.2 seconds. The experiments show that the BDD-based approach is more efficient and can verify more complex feature models than the SMV-based approach.

5 Related Work

Feature models are first proposed by Kang et al. [7] in the feature-oriented domain analysis (FODA) method, and then developed by many researchers in the field of software reuse [8,5,2,9]. In these researchers, Czarnecki et al. introduced feature models into the generative software reuse [2] and proposed the concepts of clonable features [4]. Czarnecki et al. also recognized the semantic-losing problem caused by clonable features [3], but they did not give a systematic method to resolve this problem. As far as our knowledge, we do not find any researchers who have given solutions to this problem.

Mannion [9] proposed a verifying method of feature models, in which, constraints among features are formalized using the propositional logic. Based on his research, we classified constraints in feature models into several types. For each of them, we gave its formal definition, and a graphic representation of it, which is used for feature modelers to create constraints in an easy way. We also proposed the three criteria to verify feature models [13], and have examined their effectiveness according to the deficiency framework created by Von der Maßen and Lichter [15]. However, for the checking problem of the three criteria, we transformed it into *2n+1* independent binary CSPs, without considering the connections between them.

Our research on BDD-based verification is inspired by Czarnecki's research [3], in which, Czarnecki used a commercial BDD package to verifying properties of feature models. However, it seems that Czarnecki only considered simple binary constraints between features (i.e. *requires*, and *excludes*) and those local composite constraints between a feature and its children. In addition, Czarnecki did not give details about how to decide the BDD's variable order for a feature model, and how to use a BDD in efficient ways.

Batory [1] proposed a LTMS-based approach to detect deficiencies in constraints or customization. As we have pointed out [14], this algorithm can check most of the deficiencies that our criteria can check, but in a later stage (i.e. after certain customizing decisions have been made). In addition, this approach's time complexity is same with our approach, namely, $O(2^{n+1})$. Where, n is the number of features in a feature model. In this approach, the transformation from constraints to a CNF (*conjunctive normal form*) needs a $O(2^n)$ time, and the checking of deficiencies also needs a $O(2^n)$ time, since it have to traverse all the disjunction clauses in the CNF. In our approach, a BDD's construction needs a $O(2^n)$ time, and the traversal of a BDD also needs a $O(2^n)$ time.

In addition, based on our previous work, we develop a graphical notation for constraints in clonable-enabled feature models in this paper. We do not notice that there are other researchers who have proposed such kind of graphical notations.

6 Conclusions

In this paper, we provided a kind of semantics for constraints in clone-enabled feature models. The semantics resolved two problems related to clone-enabled feature models. One is the problem of what kind of constraints should be imposed on a clonable feature and its clones, and the other is the problem of how an existing constraint should be transformed after some features in the constraint are cloned. To verify feature models with complex constraints, we proposed a BDD-based approach, which makes efficient use of the BDD data structures by considering the characteristics of the three verification criteria for feature models. Experiments showed that the BDD-based approach proposed in this paper is more efficient and can handle more complex feature models than our previous approach.

Acknowledgments. This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321805, the Hi-Tech Research and Development Program of China under Grant No. 2006AA01Z156 and 2006AA01Z189, and the Natural Science Foundation of China under Grant No. 90612011, 60528006 and 60703065.

References

1. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714. Springer, Heidelberg (2005)
2. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)

3. Czarnecki, K., Kim, C.H.P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: OOPSLA 2005 International Workshop on Software Factories (online proceedings) (2005)
4. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing Cardinality-based Feature Models and their Specialization. Software Process Improvement and Practice, special issue of best papers from SPLC 2004 10(1), 7–29 (2005)
5. Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: Proceedings of Fifth International Conference on Software Reuse, pp. 76–85. IEEE Computer Society, Canada (1998)
6. Hu, A.J.: Techniques for Efficient Formal Verification using Binary Decision Diagram. PhD thesis, Stanford University (1995)
7. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis Feasibility Study. Technical reports, Software Engineering Institute, Carnegie Mellon University (1990)
8. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering 5, 143–168 (2004)
9. Mannion, M.: Using First-Order Logic for Product Line Model Validation. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 176–187. Springer, Heidelberg (2002)
10. The SMV System. Carnegie Mellon University, `http://www.cs.cmu.edu/~modelcheck/smv.html`
11. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press, London (1993)
12. Yang, B.: Optimizing Model Checking Based on BDD Characterization, PhD thesis, CMU (1999)
13. Zhang, W., Zhao, H., Mei, H.: A Propositional Logic-Based Method for Verification of Feature Models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 115–130. Springer, Heidelberg (2004)
14. Zhang, W., Mei, H., Zhao, H.: Feature-Driven Requirements Dependency Analysis and High-Level Software Design. Requirements Engineering, vol. 11(3), pp. 205–220. Springer, London (2006)
15. von der Maßen, T., Lichter, H.: Deficiencies in feature models. In: Workshop on Software Variability Management for Product Derivation, in Conjunction with the 3rd Software Product Line Conference (2004)