A Dynamic-Priority based Approach to Fixing Inconsistent Feature Models

Bo Wang[1,2], Yingfei Xiong[3], Zhenjiang Hu[4], Haiyan Zhao[1,2]*, Wei Zhang[1,2], and Hong Mei[1,2]

[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
[2]Institute of Software, School of EECS, Peking University, Beijing, 100871, China
{wangbo07,zhhy,zhangw}@sei.pku.edu.cn, meih@pku.edu.cn
[3]Generative Software Development Lab, The University of Waterloo, Canada
yingfei@swen.uwaterloo.ca
[4]GRACE Center, National Institute of Informatics, Japan
hu@nii.ac.jp

Abstract. In feature models' construction, one basic task is to ensure the consistency of feature models, which often involves detecting and fixing of inconsistencies in feature models. Several approaches have been proposed to detect inconsistencies, but few focus on the problem of fixing inconsistent feature models. In this paper, we propose a dynamic-priority based approach to fixing inconsistent feature models, with the purpose of helping domain analysts find solutions to inconsistencies efficiently. The basic idea of our approach is to first recommend a solution automatically, then gradually reach the desirable solution by dynamically adjusting priorities of constraints. To this end, we adopt the constraint hierarchy theory to express the degree of domain analysts' confidence on constraints (i.e. the priorities of constraints) and resolve inconsistencies among constraints. Two case studies have been conducted to demonstrate the usability and scalability of our approach.

Key words: Feature Model, Priority, Inconsistency Fixing

1 Introduction

Feature models [1, 2] have been widely adopted to reuse the requirements of a set of similar products in a domain. During the process of requirements reuse, specific products that satisfy all the constraints are derived from feature models. However, inconsistent feature models (called IFMs) contain contradictory constraints that cannot be satisfied at the same time, leading to no valid products derivable from IFMs [3]. Therefore, in the construction of feature models, one basic task is to ensure the consistency of feature models, which often involves the detecting and fixing of inconsistencies in feature models.

* corresponding author

Although several approaches have been proposed to detect inconsistencies, there lacks an effective approach to aiding domain analysts to fix the inconsistencies of feature models. Finding a solution to fix inconsistencies requires quantitative analysis of certain parts of the IFMs. Even if one solution is found, it is still unclear whether there exist alternative or better solutions. Moreover, finding solutions becomes more and more difficult when feature models grow large. The largest feature model [4] reported in academy has more than 5000 features. In industry, feature models often grow up to thousands of features [5].

In this paper, we propose a dynamic-priority based approach to the interactive fixing of inconsistencies in feature models, and report an implementation of a system that not only automatically recommends a solution to fixing inconsistencies, but also supports domain analysts to gradually reach the desirable solution by dynamically adjusting priorities of constraints. To this end, we adopt the *constraint hierarchy theory* [6], a known practical theory in user interface construction [7], to express the degree of domain analysts' confidence on constraints (i.e. the priorities of constraints) and resolve inconsistencies by deleting one or more weaker constraints.

The main contributions of our paper are summarized as follows:

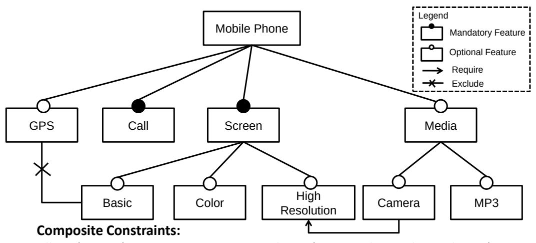
– We show the importance of the constraint hierarchy theory in fixing IFMs, and implement an efficient constraint hierarchy system[1] for fixing IFMs by adapting and extending an existing incremental algorithm, SkyBlue [7, 8].
– We extend the constraint hierarchy theory with a dynamic-priority based mechanism to help domain analysts find the desirable solution; if domain analysts are not satisfied with the solution the system recommends, they can declaratively adjust the priorities of weaker constraints so that a new solution can be produced.
– We successfully apply our system to check and fix the feature model of the web store domain and the randomly generated feature models, which indicates that our approach is promising and potentially useful in practice.

The rest of this paper is organized as follows. Section 2 introduces some preliminary knowledge. Section 3 gives an overview of our approach and illustrates it with an example. Section 4 amplifies the whole process of our approach. Section 5 illustrates usability and scalability of our approach through case studies. Section 6 describes the related work, and Section 7 concludes the paper and highlights the future work.

2 Preliminaries

In this section, we first give a short introduction to feature models, and then introduce the theory of constraint hierarchies and a constraint solver-SkyBlue. The three above are the fundamentals for fixing inconsistencies in feature models.

[1] See http://sei.pku.edu.cn/˜ wangbo07/ for more detail



Composite Constraints:
All-Set(Screen) composite-requires Single-Set(Basic, Color, High Resolution)
All-Set(Media) composite-requires Or-Set(Camera, MP3)

Fig. 1. A simplified feature model of the mobile phone domain

2.1 Feature Model

A feature model defines a set of possible products of a domain, in terms of features and the relationships between them. Fig. 1 shows a simplified feature model of the mobile phone domain (used in [9]), which adopts our meta-model of feature models [10]

A feature model is hierarchically organized. Features with different abstract levels and granularities form a hierarchy structure through *refinement relationships* between them. Refinements relationships bring constraints on features. The root feature should be bound in all products. In feature models, if a feature is bound (i.e. selected in a specific product), so it is parent. A *mandatory* feature means that it should be bound, if its parent is bound. An *optional* feature indicates that it can be unbound (i.e. deselected in a specific product), even if its parent is bound.

There are three kinds of *simple constraints* on two features, namely *require*, *m-requires*, and *excludes*. If feature A *requires* feature B, it means that B cannot be unbound when A is bound. If feature A *m-requires* feature B, it means that A and B should be bound or unbound at the same time. If feature A *excludes* feature B, it indicates that at most one of them can be bound. A mandatory feature or optional feature brings constraints with their parents, *m-requires* and *requires*, respectively.

There are three kinds of *predicates* on a set of features, namely *All*, *Alternative* and *Or*. Predicates *All*, *Alternative*, and *Or* mean these predicates are true only if all, one, and at least one features are bound in their feature sets, respectively. For example, *Or-Set(Camera, MP3)* indicates that the *Or* predicate is true when at least one features from this set are bound.

Based on predicates, there are three kinds of *composite constraints* on two feature sets, *composite-requires*, *composite-m-requires*, and *composite-excludes.* For example, *All-Set(Screen) composite-requires Single-Set(Basic, Color, High Resolution)* means if *Screen* is bound, one feature of the single feature set should be bound. For the details of the composite constraints, see Section 4.1.

Inconsistent Feature Models A feature model is *inconsistent* if it cannot produce any valid product that satisfies all the constraints of the feature model

[3]. Inconsistency is a severe problem, since we reuse feature models by deriving products from them. The inconsistencies in feature models happen when some elements of feature models are overconstrainted by contradictory constraints.

2.2 Constraint Hierarchies and SkyBlue

When overconstrainted models are checked by a constraint solver, it is not enough for the solver to signal an inconsistency and wait the modeler to fix the detected inconsistency. The *constraint hierarchy theory* [6] provides a way to specify how the overconstrainted model should be handled by maintaining constraint hierarchies. A constraint hierarchy contains a set of constraints, each assigned with a priority, indicating the importance of the constraint. Given an overconstrainted model, the constraint solver can leave weaker constraints unsatisfied in order to satisfy stronger constraints.

SkyBlue is an incremental, scalable, and efficient constraint solver that uses local propagation to maintain the constraints hierarchy. The input of SkyBlue is a set of variables and constraints on these variables. The output of SkyBlue is a set of values that satisfy stronger constraints and leave contradictory weaker constraints unsatisfied.

In SkyBlue, each constraint is equipped with one or more *methods*; SkyBlue satisfies a constraint by selecting and executing one of its methods. For example, "feature *B excludes* feature *C*" has two methods: 1) Unbind(B); 2) Unbind(C)(see Fig. 2(b)). This constraint can be satisfied by executing any one of these two methods. A constraint is *enforced* if it has a selected method, otherwise, it is unenforced. Choosing one method for a constraint is known as *enforcing.* Choosing no methods for a constraint is known as *revoking.* The variables and constraints form the *constraint graph.* The constraint graph, together with the selected methods, form the *method graph.*

The output of SkyBlue, the value set for constraints, is calculated through constructing and executing a *locally-graph-better* (called LGB) method graph. A method graph is LGB if there are no method conflicts and there are no unenforced constraints that could be enforced by revoking one or more weaker constraints (and possibly changing the selected methods for other enforced constraints with the same or stronger strength) [8].

As a simple example, consider the IFM (in Fig. 2(a)) and its corresponding constraint graph (in Fig. 2 (c)). Each constraint in the feature model (*C1*-*C4*) has one or more methods to make the constraint hold. (in Fig. 2 (b)). To satisfy every constraint, SkyBlue tries to select a method from each constraint, as shown in the upper of Fig. 2 (c), but there is a method conflict: variable *C* is determined by two methods (i.e. Bind(C) and Unbind(C)) and determined to different value, from *C3* and *C4*, respectively. To resolve this conflicts, SkyBlue finds the stronger constraints that can be enforced, while leaving the weaker constraints unenforced by constructing LGB method graph. The LGB method graph of this example is shown in the lower of Fig. 2 (c), in which *C4* is revoked. After executing the selected methods in the LGB method graph, *A, B* and *C* equal bound (selected

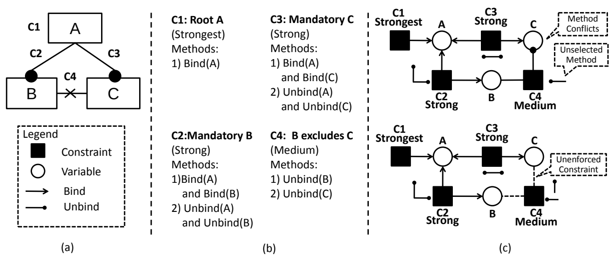


Fig. 2. A simple example for SkyBlue

in the product), which satisfy the three stronger constraints, namely *C1, C2* and *C3*.

3 Approach Overview

In this section, we give an overview of our approach, before using an example to illustrate how to fix inconsistencies.

3.1 Dynamic-Priority based IFM Fixing Process

In our approach, we detect and fix inconsistencies of feature models incrementally; we start with an empty feature model and then add constraints one by one. Every time a constraint is added into the feature model, we check inconsistencies, recommend a solution and help domain analysts find a more desirable solution. An overview of our approach is shown in Fig. 3.

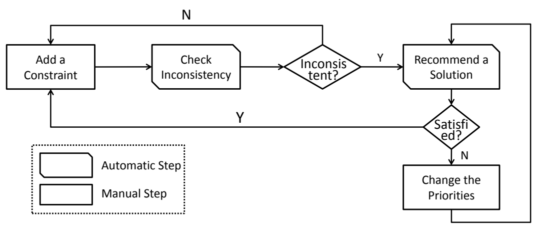


Fig. 3. The dynamic-priority based IFM fixing process

After a constraint is added to the feature model, the feature model may become overconstrainted because of the newly-added constraint. We check the inconsistency by first mapping the newly-added constraint to a SkyBlue constraint (called SBC), then trying to enforce the SBC through constructing a LGB method graph. If the constructed LGB method graph does not contain any unenforced constraints, the feature model is consistent.

If the constructed LGB method graph contains unenforced constraints and the newly-added SBC is enforced in this LGB, the feature model becomes inconsistent because of the newly-added SBC. We recommend a solution to domain analysts to fix the inconsistent feature model. This solution is composed of the unenforced constraints in the LGB method graph, and can be executed to fix the inconsistencies by deleting the unenforced constraints.

Domain analysts can examine the recommended solution. If they do not want some unenforced constraints deleted because of the newly added constraint, they can raise the priorities of these unenforced constraints, with the help of the dynamic-priority mechanism provided in our approach. We will recommend another solution according to the new priorities. When domain analysts are satisfied with the solution, the solution is performed, and the feature model becomes consistent again.

If the newly-added SBC is unenforced in the constructed LGB method graph, the newly-added SBC conflicts with some same or stronger constraint in the feature model. Our approach will recommend dropping this newly-added constraint.

For all the unenforced constraints in the LGB, we provide constraints with the same or higher priorities as potential conflict information to domain analysts, with the purpose of aiding them find desirable solutions.

Note that our approach not only supports the checking and fixing of feature models from scratch, but also supports these of a feature model that has already been constructed. Given a constructed feature model, we extract all its constraints, and map them to the SBCs. We first add and enforce the root feature to the constraint graph and then the other SBCs, according to their priorities, from weaker constraints to stronger constraints. After each SBC is added, we recommend solutions when inconsistencies detected, help domain analyst find desirable solutions according to their feedback, perform the solutions to fix inconsistencies. After all the SBCs are added, the feature model is checked and fixed completely.

3.2 An Example

To demonstrate the process of dynamic-priority based IFM fixing, let us see how to fix the inconsistent feature model in Fig. 4.

Suppose all the constraints have been added into the feature model except "feature C *excludes* feature D" (the red part in Fig. 4). These constraints are first transformed into SBCs, according to the concrete rules in Tables 1 and 2. They are then added to the constraint graph by enforcing themselves and construing LGB method graph one by one. The feature model is consistent before adding "feature C *excludes* feature D", since the LGB method graph shown in Fig. 4(b) contains no unenforced constraints. Note that even some variables are determined by more than one method in the LGB method, there is no conflicts, because these variables are set to a same value (see Section 4.2 for our definition for method conflicts in feature models).

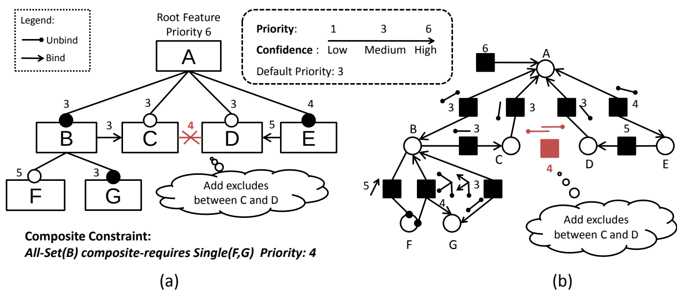

Legend:
Unbind
Bind
Root Feature Priority 6
Priority: 1 3 6
Confidence: Low Medium High
Default Priority: 3
Add excludes between C and D
Composite Constraint:
All-Set(B) composite-requires Single(F,G) Priority: 4
(a)
(b)

Fig. 4. An example of dynamic-priority based IFM fixing

After the "*exclude*" constraint is added, the feature model become inconsistent. In the generated LGB method graph, constraint "feature *B requires* feature *C*" is unenforced. We recommend deleting this constraint to fix inconsistencies.

If domain analysts are not satisfied with the recommended solution, they adjust priorities of the unenforced constraints to find the desirable solution with the help of the potential conflict information, and then we recommend other solutions according to the new priorities. For example, if the domain analysts think the "*require*" constraint should not be deleted, they raise the priority of it to 4, then we recommend another solution by constructing a new LGB method graph, in which the "*Mandatory* feature *B*" is unenforced. Therefore this constraint is recommended to be deleted.

4 Fix IFM with Dynamic Priority

In this section, we first describe how we implement the constraint hierarchy theory in fixing IFMs, through revising and extending SkyBlue. Then we show how to reach the desirable solution by adjusting priorities.

4.1 Map Feature Models to Constraint Graphs

To use SkyBlue to detect and fix inconsistencies, the first thing is to map the elements of feature models to the elements of SkyBlue constraint graphs.

Generally speaking, the mapping consists of two steps: 1) each feature of the feature model is mapped to a variable of the SkyBlue constraint graph; 2) each constraint of the feature model is mapped to a SkyBlue constraint (called SBC) that is represented by a set of methods. In feature models, each feature can have only two states: 1) bound; 2) unbound. Therefore, it is possible to derive methods from the constraints through combinations of the states of features. Concrete rules for the mapping from constraints of feature models to SBCs are listed in Tables 1 and 2.

Bind(feature) means the bind state of the feature is *bound*, and *Unbind(feature)* means the bind state of the feature is *unbound. Predicate(feature-set)* represents the value (*True* or *False*) of the predicate on the feature set.

Table 1. Methods for constraints

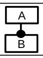
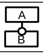
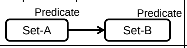
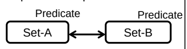
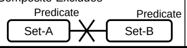
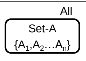
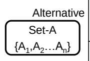
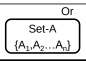
Relationship	Number of Methods	Methods
Mandatory 	2	{Bind(A), Bind(B)} or {Unbind(A), Unbind(B)}
Optional 	2	{Bind(A)} or {Unbind(B)}
Composite-Requires 	2	{Predicate(Set-A) = False} or {Predicate(Set-B) = True}
Composite-M-requires 	2	{Predicate(Set-A) = False, Predicate(Set-B) = False} or {Predicate(Set-A) = True, Predicate(Set-B) = True}
Composite-Excludes 	2	{Predicate(Set-A)= False} or {Predicate(Set-B)= False}

Table 2. Methods to determine the values of predicates

Predicate	Value	Number Of Methods	Methods
	True	1	{Bind(A_1),Bind(A_2) …Bind(A_n)}
	False	n	{Unbind(A_1)} or {Unbind(A_2)} or … {Unbind(A_n)}
	True	n	{Bind(A_1),Unbind(A_2),Unbind(A_3)…Unbind(A_n)} {Bind(A_2),Unbind(A_1),Unbind(A_3)…Unbind(A_n)} or … {Bind(A_n),Unbind(A_1),Unbind(A_2)…Unbind(A_{n-1})}
	False	$1+(n^2-n)/2$	{Unbind(A_1),Unbind(A_2)…Unbind(A_n)} or Any two of the features in the group are bound
	True	n	{Bind(A_1)} or {Bind(A_2)} or … {Bind(A_n)}
	False	1	{Unbind(A_1), Unbind(A_2) …Unbind(A_n)}

In our approach, a simple constraint (i.e., *require* and *exclude*) can be represented by a composite constraint. For example, "feature *A requires* feature *B*" can be represented as "*All-Set(A) composite-requires All-Set(B)*". Therefore, we can map simple constraints to SBCs according to these rules.

In order to derive combinations of the states of a composite constraint's features(i.e. methods), our system 1) finds the combinations of values of the composite constraint's predicates according to the last three rows of Table 1; 2) derive the combinations of the bind states of each predicate's features to hold the predicate value determined in the first step, according to Table 2.

For example, given *"All-Set(A) composite-m-requires Alternative-Set(B,C)"*, first two combination of predicates, namely, {*All-Set(A) = True, Alternative-Set(B,C) = True*}, {*All-Set(A) = False, Alternative-Set(B,C) = False*}, are generated. Then the combinations to hold the values of these predicates are generated. After that, the derived methods for this composite constraint are: {*Bind(A), Bind(B), Unbind(C)*}, {*Bind(A), Unbind(B), Bind(C)*}, {*Unbind(A), Bind(B), Bind(C)*}, {*Unbind(A), Unbind(B), Unbind(C)*}.

4.2 Recommend a Solution to Fix IFM

After one constraint is added to the feature model and in turn added to the SkyBlue constraint graph by mapping it to a SBC, our system detect inconsistencies and recommend a solution by two steps: 1) constructing a new LGB method graph through enforcing the newly-added SBC and other SBCs; 2) using the LGB to recommend a solution. Our system uses SkyBlue's LGB construction algorithm, and we extend SkyBlue by redefining method conflicts and specializing the method execution process.

Constructing a LGB method graph involves enforcing the constraints in the constraint graph. To enforce a constraint, SkyBlue selects a method for it, change the methods of same and stronger constraints, or revoke one or more weaker constraints. This process is called constructing a *method vine* or *mvine*. When an mvine for a SBC is build, they are successfully enforced.

Note that each time a constraint is successfully enforced (i.e. an mvine is constructed), one or more weaker constraints may be revoked. To construct a LGB method graph, these revoked constraints are added to the unenforced constraint set. Then our algorithm repeatedly tries to enforce all of them by constructing mvines for these constraints, until none of the constraints can be enforced. This process terminates because of the finite number of constraints. The pseudo code of constructing a LGB method graph is shown below.

Construct a LGB method graph

```
constructLGB(Constraint SBC){
    //clean the unenforced constraint set
    clearUnenforcedCnSet();
    addToUnenforcedCnSet(SBC);
    While(UnenforcedCntSet != null){
        unenforcedCn = UnenforcedCnSet.get();
    //enforce the unenforced constraint,
    //add the revoked constraints to the unenforced constraint set
        buildMvine(unenforcedCn, unenforcedCnSet);
    }
}
```

SkyBlue uses a backtracking depth-first search to build mvines. The pseudo code of building an mvine is shown as follows:

Build an Mvine for a unenforced constraint

```
buildMvine(Constraint root){
    While (root has methods){
        Method m = getMethodFromConstraint(root);
        If(!checkConflicts()){
            return true;
        }Else{
            Constrint cn = getConflictsConstraint();
            If(cn weaker than root){
                revokeConstring(cn);
                return true;
            }Else{
                buildMvine(cn);
            }
        }
    }
    return false; //start backtrack
}
```

The process of building an mvine for an unenforced constraint (called root constraint in the building process) is actually a local propagation process, the building process will end in the following situations:

– if this selected method does not conflict with other methods, this branch of the depth-first search mvine does not extend any further;
– if this selected method conflicts with the selected methods of weaker enforced constraints, SkyBlue just revokes these weaker constraints and adds them into the unenforced constraints set and this branch of the mvine does not extend any further;
– if this selected method conflicts with the selected methods of the same or stronger enforced constraint, SkyBlue selects other methods of these constraints; if these selected methods conflict with yet other constraints, choose other methods.

Our system redefines *method conflicts* and revises the corresponding part of the SkyBlue algorithm for building mvines. In SkyBlue, method conflicts happen when a variable is determined by more than one methods. However, in method graphs of feature models, the variables in constraint graphs can only be *bound* and *unbound*. Therefore, even if a variable is determined by more than one methods, it may not cause a conflict (e.g. see variable B in Fig. 4 (b)). Conflicts happens only when a variable is set to different values.

Our algorithm can also handle graphs that contain directed cycles, when executing methods to satisfy the constraints in the cycle. In SkyBlue, it is not possible to find an execution sort to satisfy the constraints in a cycle. In our system, however, methods that determine a variable set the variable to one fixed value. Therefore, our system can just execute all the methods to satisfy all the constraints.

SkeBlue provides two techniques [8], namely, *Local Collection* and *Walkabout Strength* to optimize the performance when constructing LGB method graphs. Our system successfully implement the *Local Collection* technique based on the new definition of method conflicts. The *Walkabout Strength* technique for feature models is still under construction. However, our scalability case study in Section 5.2 shows that our system can scale up to large feature models without the *Walkabout Strength* technique.

After a LGB method graph is constructed, we recommend a solution to domain analysts. How to analyze the LGB method graph to find a solution is described in Section 3.1.

4.3 Choose other Solutions through Dynamic-Priority

After a solution is recommended to fix inconsistencies, domain analysts may not be satisfied with this solution. In our approach, they can choose other solutions to fix the inconsistencies, by raising the priorities of the constraints in the solution.

The recommended solution consists of a set of weaker unenforced constraints to be deleted. These constraints conflict with some enforced constraints that have the same or higher priorities. Provided with the solution, domain analysts may not want some of the constraints in the solution to be deleted. To get a solution that does contain this constraint, domain analysts should increase the priority of the constraint. After the priority is adjusted, we construct a new LGB method

graph, with the hope of re-enforcing the raised constraint and recommending a new solution based on this new LGB method graph. This process continues until domain analysts are satisfied with the solution. The pseudo code of changing priorities is listed as follows:
Changing a constriaint's prioritiy

```
changePriority(Constraint SBC, Priority p){
    oldPriority = SBC.priority;
    SBC.priority = p;
    If (oldPriority<p){
        If (!SBC.isEnforced())
           ConstructLGB(SBC);
    }
    Else If(oldPriority>p){
        If (SBC.isEnforced())
           ConstructLGB(SBC);
    }
}
```

5 Case Studies

To investigate whether our approach is useful to fix inconsistencies in feature models, we undertook two case studies. The first one is a preliminary case study that focused on whether our approach helps domain analysts fix IFMs efficiently. The second case study investigated whether our approach is scalable to large feature models.

5.1 Usability

In the following, we first describe the process of the usability case study, then give an analysis to the results.

Study setup. In this case study, five participants were asked to build a feature model of the web store domain using our system, which is integrated into a Feature Model Graphical Editor we developed before. These participants have diverse backgrounds: two of them are senior undergraduate students who have little experience with domain engineering. The other three are graduate students whose research interests are software reuse. None of them know the approach until the case study and they are familiar with web store systems.

The five participants took the role of domain analysts to identify main features, refinements, simple constraints, composite constraints of the web store feature model. During the case study, our system recorded usage logs that include the scale of feature models, the number of the detected inconsistencies and the number of the recommended solutions. After the case study, the efficiency of our system is investigated through questionnaires.

Results. The usage log is summarized in Table 3. (The constraints showed in the results are the constraints explicitly modeled into the feature model, they

Table 3. Usage log of the usability case study

Participants	Features	Simple Constraints	Composite Constrains	Number of Inconsistencies	Average Recommendation times	Max Recommendation times	Average Deleted Constraints	Max Deleted Constraints
P1	53	17	0	7	2.57	6	1.71	2
P2	50	6	1	5	1.4	2	1.8	3
P3	57	7	3	6	2.5	3	1.5	2
P4	34	8	3	6	3.17	6	1.83	4
P5	50	6	3	3	1.33	2	1.67	2

do not contain the simple constraints that are brought with the *Mandatory* and *Optional* features.)

Most of the participants built the web store domain feature model containing about 50 features. For all the constructed feature models, there are few composite constraints. On average, when an inconsistency is detected, about 2 recommendations are needed to find the desirable solution, except for participant 4. The relatively small feature model conducted by participant 4 has more constraints. More recommendations are needed to find the desirable solution when inconsistencies detected.

The concerns of the questionnaires are classified into three categories: 1) whether the participants need recommended solutions when fixing inconsistencies; 2) whether our system can help the participants fix inconsistencies; 3) whether assigning priorities to constraints bring a lot of burden. Based on the answers to these concerns, we conclude as follows:

- Three graduate students have experience with feature model construction before. They pointed out that they often did not know how to fix the inconsistencies in relatively large and complex feature models. According to their understanding, two factors lead to this difficulty. The first one is that they have to first find out the meaning of the constraints, then analyze the inconsistencies and finally figure out how to fix them. The second factor is that when analyzing the inconsistencies, some irrelevant features and constraints disturb the domain analysts.
- Four out of five participants think our system is very helpful when fixing inconsistencies, the rest one cannot be sure whether it is helpful. The participants reported that our system helped them focus on where the inconsistencies are and how to solve them, by providing recommendations. Adjusting priorities can help them find alternative or better solutions. The time needed to fix an inconsistency is also reduced greatly.
- All the participants think assigning priorities bring them trouble when constructing constraints, due to the lack of the standards for the priorities of constraints. They think the default priority is rather helpful. They also point out that adjusting priorities is relatively much easier, because they can adjust priorities through comparing constraints.

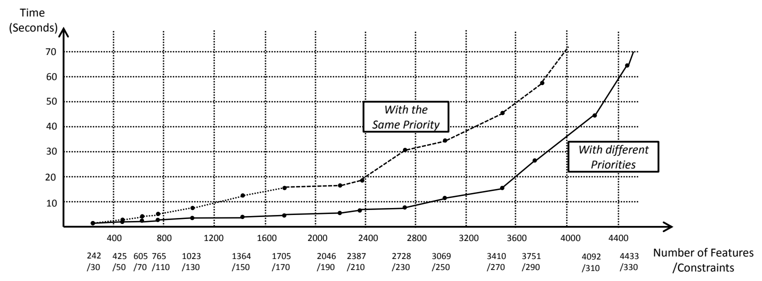


Fig. 5. Experiments results for fixing randomly generated feature models with the same and different priorities, respectively

5.2 Scalability

In this case study, we investigate the scalability of our system. To evaluate the scalability, we randomly generate feature models and fix the inconsistencies in the generated feature models. We use generated models because it is very difficult to get real world large feature models. Although there are publications about large models, none of these models are publicly available. On the other hand, industrial feature models are always confidential.

We implement an algorithm to generate feature models randomly[2]. Each generated feature model contains a root feature. We can specify the number of the subtrees that are connected to the root feature, the height of the subtrees, the number of the chid features for each non-leaf feature in the subtrees, the number of the constraints. The percentage of the variability of features are: Mandatory (25%) and Optional (75%).

To make the study reflects the scalability of our system, we generate two groups of feature model, with the same and different (randomly between 1 and 5) priorities. In our case study, we adopt the first recommended solution to fix inconsistencies.

The environment for our experiments is a Win 7 PC with a 2.66GHz CPU, 2GB memory and the result is shown in Fig. 5. (The constraints showed in the results are the constraints explicitly modeled into the feature model, they do not contain the simple constraints that are brought with the *Mandatory* and *Optional* feature.) Our system checks and fixes inconsistencies incrementally. For example, in the second case, 425 mandatory or optional features are added (each bring a constraint), and 50 constraints are explicitly modeled, we check 475 times in total and cost 0.8s in all.

From the result, we can see that, our system can handle feature models with more than 4000 features and 300 constraints, which is a good support for domain analysts when they fix inconsistencies in feature models.

[2] See http://sei.pku.edu.cn/˜ wangbo07/ for the source code.

6 Related Work

Feature models are first proposed by Kang et al. [1] in the feature-oriented domain analysis (FODA) method. Czarnecki et al. [11] proposed probabilistic feature models, in which soft constraints express the conditional probability of configurations to contain certain features. Our approach use priorities to determine which constraints should stay in feature models, when inconsistency happens.

Many studies focus on the automatic analysis of the deficiencies of feature models [9]. Maßen and Lichter [3] proposed a deficiency framework of feature model. They point out that inconsistency is one of the most severe deficiencies in feature models. Mannion et al. [12] was the first to use propositional formulas to analyze feature models. Batory [13] proposed an approach to detecting deficiencies with SAT Solver. In his work, a Logic Truth Maintenance System was designed to analyze feature models. Benavides et al. [14] were the first to use constraint programming for analysis on feature models. Our previous work [15] focused on how to analyze feature models using BDD.

However, all these works only focus on the detection of deficiencies. Egyed [16] proposed an approach to fixing inconsistencies in UML models. Trinidad et al. [17] focus on the explanation of deficiencies in feature models based on constraint programming, but they do not give a solution to the deficiencies and the scalability of his approach is also not clear. White et al. [18] focus on detect errors on the configuration of a feature model, and propose changes in the configuration in terms of features to be selected or deselected to correct the error. Our approach focuses on the feature model itself, not the configuration of feature models.

7 Conclusion and Future Work

In this paper, we adopt the constraint hierarchy theory and extend the constraint solver-SkyBlue to implement a system that can help domain analysts fix inconsistent feature models effectively. When a constraint is added to the feature model, we automatically check the inconsistencies by constructing a LGB method graph, and recommend domain analysts a solution for fixing the inconsistencies by analyzing the constructed LGB method graph. Furthermore, we can recommend other solutions so that a more desirable solution can be obtained based on the feedback of domain analysts. The feedback is expressed declaratively through the adjustment to the priorities of constraints. Our future work will focus on working on more practical examples, and investigating applicability of our approach to inconsistency fixing of other models such as UML models.

Acknowledgments. The authors would like to thank Shin Nakajima (NII, Japan) and Lu Zhang (Peking University, China) for discussing with us on model inconsistency detection and fixing, and to Hiroshi Hosobe (NII, Japan) for introducing Delta/Skyblue to us. This work is supported by the National Basic

Research Program of China (973) under Grant No. 2009CB320701, the Science Fund for Creative Research Groups of China under Grant No. 60821003, the Natural Science Foundation of China under Grant No. 60703065, 60873059 and the National Institute of Informatics (Japan) Internship Program.

References

1. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU-SEI (1990)
2. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10** (2005) 7–29
3. von der Maßen, T., Lichter, H.: Deficiencies in feature models. In: Workshop on Software Variability Management for Product Derivation, in Conjunction with SPLC. (2004)
4. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: The variability model of the Linux kernel. In: VaMoS. (2010) 45–51
5. Batory, D.S., Benavides, D., Cortés, A.R.: Automated analysis of feature models: challenges ahead. Commun. ACM **49** (2006) 45–47
6. Borning, A., Freeman-Benson, B.N., Wilson, M.: Constraint hierarchies. Lisp and Symbolic Computation **5** (1992) 223–270
7. Sannella, M.: SkyBlue: A multi-way local propagation constraint solver for user interface construction. In: ACM Symposium on User Interface Software and Technology. (1994) 137–146
8. Sannella, M.: The SkyBlue constraint solver and its applications. In: PPCP. (1993) 258–268
9. Benavides, D., Segura, S., Cortés, A.R.R.: Automated analysis of feature models 20 years later: a literature review. Information Systems (2010)
10. Zhang, W., Mei, H., Zhao, H.: Feature-driven requirement dependency analysis and high-level software design. Requir. Eng. (2006) 205–220
11. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: SPLC. (2008) 22–31
12. Mannion, M.: Using first-order logic for product line model validation. In: SPLC. (2002) 176–187
13. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC. (2005) 7–20
14. Benavides, D., Trinidad, P., Cortés, A.R.: Using constraint programming to reason on feature models. In: SEKE. (2005) 677–682
15. Zhang, W., Yan, H., Zhao, H., Jin, Z.: A BDD-based approach to verifying clone-enabled feature models' constraints and customization. In: ICSR. (2008) 186–199
16. Egyed, A.: Fixing inconsistencies in uml design models. In: ICSE. (2007) 292–301
17. Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., Toro, M.: Automated error analysis for the agilization of feature modeling. J. Syst. Softw. **81** (2008) 883–896
18. White, J., Schmidt, D.C., Benavides, D., Trinidad, P., Cortés, A.R.: Automated diagnosis of product-line configuration errors in feature models. In: SPLC. (2008) 225–234