

Tolerating Inconsistency in Feature Models

Bo Wang
Key Laboratory of High
Confidence Software
Technologies
(Ministry of Education)
Peking University, China
wangbo07@sei.pku.edu.cn

Zhenjiang Hu
GRACE Center
National Institute of
Informatics
Tokyo, Japan
hu@nii.ac.jp

Yingfei Xiong
Generative Software
Development Lab
The University of Waterloo
Waterloo, Canada
yingfei@swen.uwaterloo.ca

Haiyan Zhao
Key Laboratory of High
Confidence Software
Technologies
(Ministry of Education)
Peking University, China
zhhy@sei.pku.edu.cn

Wei Zhang
Key Laboratory of High
Confidence Software
Technologies
(Ministry of Education)
Peking University, China
zhangw@sei.pku.edu.cn

Hong Mei
Key Laboratory of High
Confidence Software
Technologies
(Ministry of Education)
Peking University, China
meih@pku.edu.cn

ABSTRACT

Feature models have been widely adopted to reuse the requirements of a set of similar products in a domain. When constructing feature models, it is difficult to always ensure the consistency of feature models. Therefore, tolerating inconsistencies is important during the construction of feature models. The usual way of tolerating inconsistencies is to find the minimal unsatisfiable core. However, identifying the minimal unsatisfiable core is time-consuming, which decreases itself the practicability.

In this paper, we propose a priority based approach to tolerating inconsistencies in feature models efficiently. The basic idea of our approach is to find the weaker unsatisfied constraints, while keeping the rest of the feature model consistent. Our approach tolerates inconsistencies with the help of priority based operations while building feature models. 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 tolerate inconsistencies in feature models. Experiments have been conducted to demonstrate that our system can scale up to large feature models.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software—*Domain engineering*

Keywords

Feature Model, Constraint Hierarchy, Tolerate Inconsistency

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LWI '2010 Antwerp, Belgium

Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

1. INTRODUCTION

Feature models [6, 7] have been widely adopted to reuse the requirements of a set of similar products in a domain. During the process of products reuse, specific products that satisfy all the constraints are derived from feature models. Inconsistent feature models contain contradictory constraints that cannot be satisfied at the same time, leading to no valid products derivable from them [13]. However, it is difficult to always ensure the consistency of feature models, during the construction of feature models. Therefore, tolerating inconsistencies is important when constructing feature models.

The usual way of tolerating inconsistencies is to find the minimal unsatisfiable core in inconsistent feature models. However, identifying the minimal unsatisfiable core is time-consuming [9], which decreases itself the practicability.

In this paper, we propose a priority based approach to tolerating inconsistencies in feature models, and report an implementation of a system that not only automatically tolerates inconsistencies by identifying weaker unsatisfied constraints, but also supports domain analysts to handle the tolerated inconsistencies, with the help of priority based operations. To this end, we adopt the *constraint hierarchy theory* [5], a known practical theory in user interface construction, to express the degree of domain engineers' confidence on constraints (i.e. the priorities of constraints) and tolerate inconsistencies in feature models. The main contributions of our paper are summarized as follows:

- We show the importance of the constraint hierarchy theory in tolerating inconsistencies in feature models, and we adopt it to divide a feature model into the *consistent feature model* part and *pending constraint set* part, which will help tolerate inconsistencies in feature models.
- We make the first attempt of conducting a constraint hierarchy system¹ for tolerating inconsistencies in feature models, through adapting and extending an existing incremental algorithm-SkyBlue [10, 11].

¹See <http://sei.pku.edu.cn/~wangbo07/> for more details.

- We have conducted the experiments on our system, which demonstrates that our approach scales up to very large feature models.

The rest of this paper is organized as follows. Section 2 introduces some preliminary knowledge on feature models, constraint hierarchy and SkyBlue. Section 3 gives an overview of our approach. Section 4 amplifies our approach. Section 5 illustrates the scalability of our approach. Section 6 discusses the related work, and Section 7 concludes the paper and highlights the future work.

2. PRELIMINARIES

In this section, we first describe feature models, followed by the introduction to the constraint hierarchy theory and SkyBlue. All these three serve as the fundamental supports for tolerating inconsistencies in feature models.

2.1 Feature Model

A feature model organizes the requirements of the products of a domain, in terms of features and the relationships between them. A simplified feature model of the mobile phone domain [3], which adopts our meta model of feature models [15], is shown in Fig. 1.

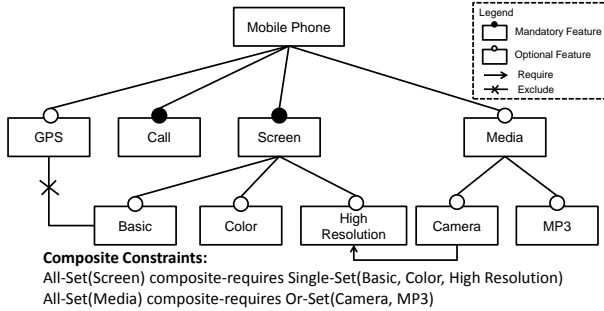


Figure 1: A simplified example of the mobile phone domain

A feature is a software characteristic with sufficient user or customer value, which essentially denotes a cohesive set of individual requirements [14]. In feature models, if a feature is bound (i.e. selected and implemented in a product), so is its parent. A *mandatory* feature should be bound if its parent is bound. An *optional* feature can be unbound (i.e. deselected and not implemented in a product), even if its parent is bound.

There are three kinds of *simple constraints* on two features, namely *requires*, *m-requires*, and *excludes*. If feature *A* *requires* feature *B*, it indicates that *B* must be bound 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 they cannot be bound at the same time.

There are three kinds of *predicates* on a set of features, namely *All*, *Alternative* and *Or*. Predicates *All*, *Alternative*, and *Or* indicate these predicates are true only if all, only one, and at least one features are bound in their feature sets, respectively. For instance, *Single-Set (Basic, Color, High Resolution)* indicates that this predicate is true when only one kind of screens can be chosen in a product.

Based on the predicates, there are three kinds of *composite constraints* on two feature sets, *composite-requires*, *composite-m-requires*, and *composite-excludes*. For example, given *All-Set(Media)* *composite requires Or-Set(Camera, MP3)*, if *All-Set(Media)* is true, *Or-Set(Camera, MP3)* must be true. For the details of the composite constraints, see sub-section 4.1.

Products are derived from a feature model by binding and unbinding constraints. A valid derived product must satisfy all the constraints in the feature model. A feature model contains inconsistencies if no valid products can be found to satisfy all the constraints in this feature model [13]. These inconsistencies are caused by the contradictory constraints in feature models.

2.2 Constraint Hierarchies and SkyBlue

When a solver is used to check inconsistent models, it is not enough for the solver to just signal the detected inconsistencies. The *constraint hierarchy theory* [5] provides a way to handle the detected inconsistencies through maintaining constraint hierarchies. A constraint hierarchy contains a set of constraints, each assigned with a priority, indicating the importance of the constraint. Given an inconsistent model, a constraint solver makes sure that stronger constraints are satisfied, through unsatisfying the contradictory weaker constraints.

SkyBlue is an incremental constraint solver that uses local propagation to maintain the constraint hierarchies. It has been successfully applied in many GUI systems. SkyBlue requires that methods can be derived from constraints (explained later in this sub-section), and thus is not applicable to some kinds of constraints. One important finding of our approach is that the constraints in feature models satisfy the prerequisite of SkyBlue (with minor extension) and thus can enjoy the performance boost of SkyBlue (see Section 4).

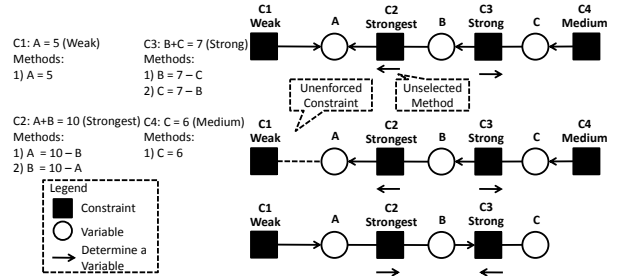


Figure 2: A simple example for SkyBlue

The input of SkyBlue is a set of variables and the constraints on these variables. The output of SkyBlue is a set of values that satisfy stronger constraints and leave the contradictory weaker constraints unsatisfied.

In SkyBlue, each constraint is equipped with one or more *methods*. SkyBlue satisfies a constraint by selecting one of its methods and executing the selected method. SkyBlue *enforces* a constraint by choosing one method for this constraint and *revoke* a constraint by choosing no methods for this constraint. A constraint is *enforced* if it has a selected method, otherwise, it is *unenforced*. The variables and the constraints form the constraint graph. The constraint graph, together with the selected methods, forms the method graph.

The output of SkyBlue, the value set for variables, 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) [10].

As a simple example, the method graphs in Fig. 2 has four constraints $C1$, $C2$, $C3$ and $C4$ on three variables A , B and C . Each constraint has one or more methods to make the constraint hold (for instance, two methods are given to satisfy $C3$ by either calculating B from C or calculating C from B). To satisfy every constraint, SkyBlue tries to select a method from each constraint, as shown in the upper right of Fig. 2, but there is a method conflict (inconsistency): variable A is determined by two methods, namely, $A=5$ and $A=10-B$. To resolve this conflict, we have to revoke some weaker constraint to enforce the stronger constraints. SkyBlue finds the strong 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 middle right of Fig. 2, where $C1$ is revoked. After executing the selected methods in the LGB method graph, A equals to 9, B equals to 1, and C equals to 6, which satisfy the three stronger constraints, namely $C2$, $C3$ and $C4$. $C1$ may be reenforced automatically when its contradictory constraints are deleted. For example, if $C4$ is deleted, a new LGB method graph is constructed, in which constraint $C1$ is reenforced by selecting method “ A equals to 5”, as shown in the lower right of Fig. 2.

3. APPROACH OVERVIEW

In this section, we first give an overview of our approach, and then we use an example to illustrate how to tolerate inconsistencies in feature models.

3.1 Feature Model Inconsistency Tolerance

In our approach, a feature model is divided into two parts, namely the *consistent feature model* part (called CFM part) and the *pending constraint set* part (called PCS part). The PCS contains weaker constraints that conflict with some stronger constraints in the CFM. If the PCS is empty, the feature model is consistent. Domain analysts work on the CFM to construct the feature model and work on the PCS to handle the tolerated inconsistencies. An overview of the inconsistency tolerance is shown in Fig. 3.

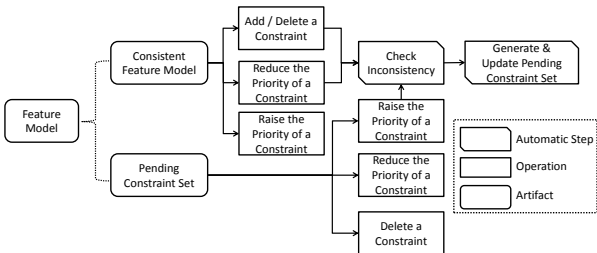


Figure 3: Feature model inconsistency tolerance

We divide a feature models into the CFM and the PCS through constructing LGB method graphs. The CFM consists of the enforced constraints in the LGB method graph,

and the PCS consists of the unenforced constraints in the LGB method graph. After a new LGB method graph is constructed, the CFM and the PCS are updated.

1. If the constructed LGB method graph does not contain any unenforced constraints, the PCS is empty and the CFM contains all the constraints in the feature model. At this moment, the feature model is consistent.
2. If the constructed LGB method graph contains one or more weaker unenforced constraints, the constraints in the PCS are replaced with these unenforced constraints and the constraints in the CFM are replaced with the enforced constraints in the LGB method graph. At this moment, the feature model is inconsistent.

Four kinds of operations on the CFM are provided to help domain analysts construct feature models. When constructing feature models, domain analysts can add a constraint with priority into the CFM or delete a constraint from it. Domain analysts can change priorities of constraints when constructing feature models.

There are three conditions on which the enforced constraints in the CFM may become unenforced and thus are put into the PCS: 1) their priorities are reduced; 2) the priorities of their contradictory weaker constraints in the PCS are raised; 3) some contradictory stronger constraints are added. When these conditions are met, we generate a new LGB method graph to update the CFM and the PCS.

Three kinds of operations on the PCS are provided to help domain analysts handle the tolerated inconsistencies. If the domain analysts become more confident about a constraint in the PCS, he can raise its priority. The possibility of reenforcing this constraint become larger as its priority rises. If the domain analysts become less confident about a constraint, he can reduce its priority. The possibility of enforcing this constraint becomes smaller as its priority decreases. If domain analysts believe some constraints do not represent the correct relationships among the features, they can delete them from the in pending constraint set.

There are three conditions on which the unenforced constraints in the PCS can be re-enforced again, and thus are put into the CFM: 1) their priorities are raised; 2) their contradictory stronger constraints in the CFM are deleted; 3) the priorities of their contradictory stronger constraints in the CFM are reduced. When these conditions are met, we generate a new LGB method graph to update the CFM and the PCS.

3.2 An Example

To demonstrate how we tolerate inconsistencies in feature models, let us see how to find the CFMs and the PCSs, and handle the tolerated inconsistencies in the 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). The feature model is consistent before adding “feature C excludes feature D ”, since the LGB shown in Fig. 4(b) contains no unenforced constraints. At this moment, the PCS is empty. Note that even some variables are determined by more than one method in the LGB method graph, no conflict happens, because these variables are set to the same value (see Section 4 for more detail).

When the “exclude” constraint is added and enforced, a LGB method graph, in which the constraints “Mandatory D ”

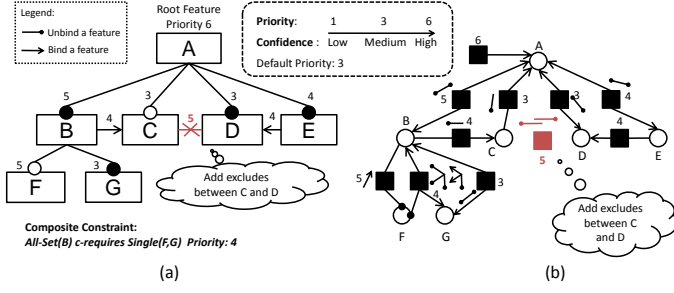


Figure 4: An example of feature model inconsistency tolerance

and “feature E requires feature D ” are revoked, is generated. The PCS consists of these two revoked constraints.

Domain analysts can delete the constraint “feature D requires feature E ” if they believe the “require” constraint does not represent the correct relationship between feature D and E . If domain analysts have more confidence on the “Mandatory D ” than before, they raise its priority to 5. Then our approach will try to enforce it by constructing a new LGB. In the new LGB method graph, only “feature B requires feature C ” is revoked. The PCS is updated, and it only contains the “require” constraint.

4. TOLERATE INCONSISTENCIES IN FEATURE MODELS

In this section, we will describe how we adopt the constraint hierarchy theory by revising and extending SkyBlue to tolerate inconsistencies in feature models.

4.1 Map Feature Models to Constraint Graphs

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

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.

SkyBlue cannot be generalized to derive methods from some “inequality-like” constraints. But feature models are different from this general case. In feature models, each feature can have only two states: 1) bound; 2) unbound. Therefore, it is possible to derive methods for constraints in feature models, through combinations of the states of “certain” features. Concrete rules for the mapping from feature models to constraint graphs are listed in Tables 1 and 2.

Binding a feature ($Bind(feature)$) sets the bind state of the feature *bound*. Unbinding a feature ($Unbind(feature)$) sets the bind state of the feature *unbound*. *Predicate* on a feature set represents the value of the predicate of a feature set. In our approach, simple constraints can be represented by a composite constraint. For example, “feature A excludes feature B ” can be represented as “*All-Set(A) composite-excludes All-Set(B)*”.

In Table 2, each kind of group predicates is associated with a set of methods that can be executed to set the predicate *True* or *False*. These predicate methods, together with the composite constraint methods, can map a com-

Table 1: Methods for constraints

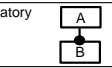
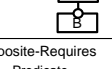
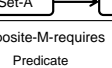
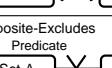




Relationship	Number of Methods	Methods
Mandatory 	2	{Bind(A), Bind(B)} or {Unbind(A), Unbind(B)}
Optional 	2	{Bind(A)} or {Unbind(B)}
Composite-Requires 	2	{Predicate(Set-A) = False} or {Predicate(Set-B) = True}
Composite-M-requires 	2	{Predicate(Set-A) = False, Predicate(Set-B) = False} or {Predicate(Set-A)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

Table 2: Methods for predicates

Predicate	Value	Number Of Methods	Methods
All 	True	1	{Bind(A ₁), Bind(A ₂) ... Bind(A _n)}
	False	n	{Unbind(A ₁)} or {Unbind(A ₂)} or ... {Unbind(A _n)}
Alternative 	True	n	{Bind(A ₁), Unbind(A ₂), 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
Or 	True	n	{Bind(A ₁)} or {Bind(A ₂)} or ... {Bind(A _n)}
	False	1	{Unbind(A ₁), Unbind(A ₂) ... Unbind(A _n)}

posite constraint to an SBC. For example, given a composite constraint “*All-Set(A, B) composite-excludes Alternative-Set(C, D)*”, methods are generated through combination of the states of the features in the two sets. The four derived methods are { $Unbind(A)$ }, { $Unbind(B)$ }, { $Unbind(C)$, $Unbind(D)$ }, and { $Bind(C)$, $Bind(D)$ }.

4.2 Construct LGB Method Graphs

In our approach, we divide a feature model into the CFM and the PCS, and provide priority-based operations through constructing LGB method graphs. To construct LGB method graphs for feature models’ tolerance, we have to extend and revise SkyBlue through: 1) redefining method conflicts; 2) specializing the execution process.

An LGB method graph is constructed under the following conditions: 1) a new constraint is added/deleted to the CFM; 2) the priority of a constraint in the CFM/PCS is changed. The pseudo codes are shown below.

Add/delete a constraint in CFM

```

ConstructCFM(Constraint SBC, Boolean isAdd){
  If(isAdd){
    ConstructLGB(Constraint SBC)
  }
  Else {
    UnenforcedCnsSet =
      collectUnenforcedConstraints();
    While(UnenforcedCnsSet != null){
      unenforcedCn = UnenforcedCnsSet.get();
      ConstructLGB(SBC);
    }
  }
}

```

Changing a constraint's priority

```

ChangePriority(Constraint SBC, Priority p){
    oldPriority = SBC.priority;
    SBC.priority = p;
    If (oldPriority < p){
        If (SBC.selectedMethod == null)
            ConstructLGB(SBC);
    }
    Else If (oldPriority > p){
        If (SBC.selectedMethod != null)
            ConstructLGB(SBC);
    }
}

```

Constructing an LGB method graph involves enforcing the constraints in the constraint graph. To enforce a constraint, we select a method for it, change the methods of the constraints with the same or stronger priorities, or revoke one or more weaker constraints. This process is called constructing a *method vine* or *mvine*. When an mvine for the newly-added SBC is built, the SBC is 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 an LGB method graph, these revoked constraints are added to the unenforced constraint set. Then our algorithm repeatedly tries to enforce all the constraints in the unenforced constraint set 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 an LGB method is shown below.

Construct an LGB method graph

```

ConstructLGB(Constraint SBC){
    //clean the unenforced constraint
    //set before enforce the newly-added SBC}
    clearUnenforcedCnSet();
    addToUnenforcedCnSet(SBC);
    While(UnenforcedCnSet != null){
        unenforcedCn = UnenforcedCnSet.get();
        buildMvine(unenforcedCn, unenforcedCnSet);
    }
}

```

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

Build a Mvine for a unenforced constraint

```

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

```

To apply the SkyBlue to detect and tolerate inconsistencies in feature models, our algorithm redefines *method conflict* and revises the SkyBlue algorithm for building mvines. In SkyBlue, a conflict happens when a variable is determined by more than one method. In our approach, the variables

in the constraint graph can only be *bound* and *unbound*. Therefore, even if a variable is determined by more than one method, it may not cause a conflict (e.g. see variable *B* in Fig. 4). A conflict happens only when this variable is set to different value by different methods.

In SkyBlue, if an LGB method graph contains directed cycles, it is not possible to find an execution sort to satisfy all the variables in the LGB method graph. However, our approach can just execute all the methods to satisfy all the constraints, because all the methods set a variable to a fixed value.

5. PERFORMANCE

In this section, we investigate whether our approach can scale up to large feature models. To evaluate the scalability, we randomly generate feature models² and tolerate the inconsistencies in the generated feature models. We choose to use generated models because none of the large models are publicly available. The 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 child feature 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%). The priorities of constraints are randomly set between 1 to 5.

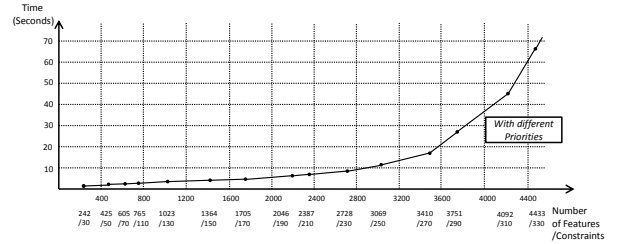


Figure 5: Experiments results for randomly generated feature models

The environment for our experiments is a Win 7 PC with a 2.66 GHz CPU, 2GB memory and the result is shown in Fig. 5. A mandatory feature or optional feature brings constraints with their parents, *m-requires* and *requires*, respectively. The constraints showed in the results are the constraints explicitly modeled into the feature models, they do not contain the simple constraints that are brought with the *Mandatory* and *Optional* feature.

In our approach, we check inconsistency and generate the PCS 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 generate the PCS 475 times in total and cost 1.2s in all. The results indicate that our approach can scale up to large feature models.

6. RELATED WORK

Feature models are first proposed by Kang et al. [7] in the feature-oriented domain analysis (FODA) method. Since

²See <http://sei.pku.edu.cn/~wangbo07/> for our system and the feature model random generation algorithm.

then many researches focus on the detection of inconsistencies in feature models [3]. Maßen and Lichter [13] 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. [8] was the first to use propositional formulas to find inconsistencies. Batory [2] proposed an approach to detecting deficiencies with SAT Solver. Benavides et al. [4] were the first to use constraint programming for analysis on feature models. Our previous work [16] focused on how to analyze feature models using BDD.

However, these approaches do not focus on how to find the unsatisfied constraints and tolerate inconsistencies in feature models. Balzer [1] pointed out the importance of tolerating inconsistencies, when the inconsistencies cannot be fixed. Trinidad et al. [12] focus on the explanation of inconsistencies in feature models, which helps find unsatisfied constraints. Nakajima et al. [9] propose some heuristics rules to find the unsatisfied core. However, these approaches do not provide explicit support to handle the tolerated inconsistencies and the scalability of these approaches is also not clear. Zowghi et al. [17] propose an approach to handling inconsistencies as a consequence of evolution changes performed on requirements specification, while our approach focuses on the inconsistencies in 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 effectively tolerate inconsistencies in feature models. The feature model is divided into two parts, consistent feature model part and the pending constraint set part, through building LGB method graphs. Domain analysts can construct the feature model by working on the CFM while handling the tolerated inconsistencies that are expressed explicitly by the PCS. Three operations are defined and supported by our system, with the purpose of helping domain analysts handle the tolerated inconsistencies. Our future work will focus on investigating the applicability of our approach.

8. ACKNOWLEDGMENTS

The authors would like to thank 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 National High Technology Research and Development Program of China (863) under Grant No. 2009AA01Z139, the Natural Science Foundation of China under Grant No. 60703065, 60873059, and the National Institute of Informatics (Japan) Internship Program.

9. REFERENCES

- [1] R. Balzer. Tolerating inconsistency. In *ICSE*, pages 158–165, 1991.
- [2] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.
- [3] D. Benavides, S. Segura, and A. R.-R. Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 2010.
- [4] D. Benavides, P. Trinidad, and A. R. Cortés. Using constraint programming to reason on feature models. In *SEKE*, pages 677–682, 2005.
- [5] A. Borning, B. N. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.
- [6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
- [7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU-SEI, November 1990.
- [8] M. Mannion. Using first-order logic for product line model validation. In *SPLC*, pages 176–187, 2002.
- [9] S. Nakajima. Semi-automated diagnosis of foda feature diagram. In *SAC '10*, pages 2191–2197, New York, NY, USA, 2010. ACM.
- [10] M. Sannella. The skyblue constraint solver and its applications. In *PPCP*, pages 258–268, 1993.
- [11] M. Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *ACM Symposium on User Interface Software and Technology*, pages 137–146, 1994.
- [12] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883 – 896, 2008. Agile Product Line Engineering.
- [13] T. von der Maßen and H. Lichter. Deficiencies in feature models. In *Workshop on Software Variability Management for Product Derivation, in Conjunction with SPLC*, 2004.
- [14] B. Wang, W. Zhang, H. Zhao, Z. Jin, and H. Mei. A use case based approach to feature models' construction. *IEEE International Conference on Requirements Engineering*, 0:121–130, 2009.
- [15] W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requir. Eng.*, 11(3):205–220, 2006.
- [16] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A bdd-based approach to verifying clone-enabled feature models' constraints and customization. In *ICSR*, pages 186–199. Springer, 2008.
- [17] D. Zowghi and R. Offen. A logical framework for modeling and reasoning about the evolution of requirements. In *RE*, page 247. IEEE Computer Society, 1997.