Feature Model Validation: A Constraint Propagation-Based Approach

Guoheng Zhang, Huilin Ye, and Yuqing Lin
School of Electrical Engineering and Computer Science
University of Newcastle
Callaghan 2308, NSW, Australia

Abstract - *Feature model validation aims to identify errors in feature models. The two major errors, called dead features and false variable features, are caused by contradictory feature relationships in a feature model. Current existing approaches use constraint satisfaction problem (CSP) and CSP solvers to identify these feature model errors. However, CSP is a NP-complete problem and CSP solvers reveal a weak time performance. To overcome this limitation, we develop a constraint propagation based approach to identify dead features and false variable features. The correctness and efficiency of our approach is compared with a well known feature model validation tool, called FAMA, based on a number of large-size feature models which are randomly generated. The time spent for identifying feature model errors is significantly reduced from O (2^n) required by FAMA which uses CSP solvers to O (n^2), while both approaches identified the same set of errors for all the evaluated feature models.*

Keywords: feature model validation, dead features, false variable features, constraint propagation, feature models.

1 Introduction

Currently, feature models are widely used to represent the commonalities and variabilities of software product line (SPL) members [1]. A feature model consists of features and feature relationships among the features. From a feature model, a specific product can be derived during product configuration which is the process of selecting the desired features from a feature model based on customer' requirements and feature relationships specified in a feature model.

In a feature model, there may exist contradictory feature relationships. For example, feature *A* requires feature *B* and feature *B* excludes with feature *A*. Obviously there are contradictory feature relationships between *A* and *B*. The contradictory feature relationships will result in feature model errors, such as dead features and false variable features, which prevent a feature model from representing the right set of product line members desired by the domain. Therefore, these feature model errors must be detected and corrected for an effective product configuration. It is a difficult and time-consuming task to identify these feature model errors for large-size feature models. Therefore, an automated method of identifying feature model errors has been recognized as one of challenges in the area of feature model validation [2].

Several feature model errors have been defined in literature, such as dead features, false variable features, redundancies, wrong cardinality and void feature model [3]. Among these feature model errors, dead features and false variable features have been recognized as the most critical errors [3-6]. A variable feature is dead if it cannot appear in any software product line (SPL) member and a variable feature is false variable if it must be included into all SPL members [5]. In this paper, we concentrate on the identification of dead features and false variable features.

The most promising approach of identifying feature model errors is proposed by *Trinidad* [5] and *Czarnecki* [7]. This approach is based on constraint satisfaction problem (*CSP*) which transforms features into variables and feature dependencies into constraints over the variables. To check whether a feature is dead, this approach assigns the checked feature with value "true" and then uses *CSP* solvers to find valid solutions over the other variables based on the constraints. If no valid solutions can be found, the checked feature is dead. The false variable features are identified in a similar way except that the checked feature is assigned with value "false". Although this approach identifies feature model errors automatically, it is quite inefficient, because *CSP* is a NP-complete problem and the number of solutions that need to be examined is 2^n where n is the number of features in a feature model. If the number of features in a feature model is large, the *CSP*-based approach is inefficient to identify feature model errors.

In this paper, we propose a recursive algorithm to identify dead features and false variable features based on constraint propagation, which is the process of propagating the inclusion or removal of a variable feature to a set of other features through different feature relationships. A set of constraint propagation rules are proposed. The time spent for identifying feature model errors is significantly reduced from $O(2^n)$ in [5] to $O(n^2)$.

The rest of this paper is organized as follows: section 2 will introduce some background knowledge on feature models and feature model errors. In section 3, we will introduce constraint propagation and propose the formalized constraint propagation rules. In section 4, we develop a recursive algorithm to identify feature model errors based on constraint propagation rules. In section 5, we evaluate the correctness and efficiency of our approach by comparing with a well known feature model validation tool, called *FAMA*, based on a

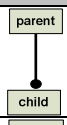
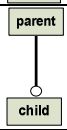
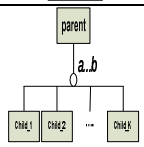
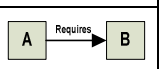
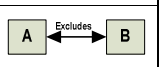
number of randomly generated feature models. Finally we conclude this paper and identify the future work in Section 6.

2 Background

2.1 Feature Models

A feature model represents the commonalities and variabilities of member products in a software product line (SPL) in terms of features. A feature model is mostly represented as a feature tree where nodes represent features and edges represent relationships among features. Since Kang et al., [8] first introduced the basic feature model in FODA several extensions have been proposed, such as feature attributes, feature cardinality and group cardinality [9-11]. Cardinality-based feature model (*CBFM*) which extends the basic feature model with group cardinality, is most widely used because of its conceptual completeness. There are two kinds of feature relationships in a *CBFM*: hierarchical relationship and cross-tree feature dependency. A hierarchical relationship describes the selection relationship between a parent feature and its child features, including mandatory, optional and feature group with group cardinality. A feature dependency describes feature selection constraint between two cross-tree features, including requires and excludes. In this paper, we adapt cardinality-based notations to represent feature models. The semantics and notations of feature relationships in a *CBFM* are shown in table 1.

Table 1 The semantics and notations of feature relationships in *CBFM*

Feature Relationship	Semantics	Notation
Mandatory	if the parent feature is selected, the child feature which has a mandatory relationship with its parent must be selected.	
Optional	if the parent feature is selected, the child feature which has an optional relationship with its parent may or may not be selected.	
Feature Group	if the parent feature is selected, at least *a* features must be selected and at most *b* features can be selected from the feature group based on the group cardinality [*a*…*b*].	
Requires	"A requires B" means that if feature A is selected, feature B must be selected.	
Excludes	"A excludes B" means that A and B cannot be selected into the same member product.	

A feature model can represent all potential member products of a SPL in terms of features and feature relationships among features. A specific member product can be derived from a SPL in feature-based product configuration during which the desired features are selected based on customers' requirements. The feature selection is also constrained by all the feature relationships specified in a feature model. Fig. 1 shows a *CBFM* of a simplified tourist guide SPL.

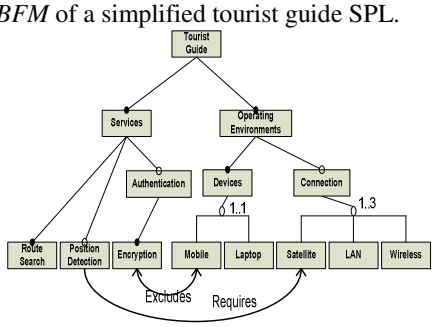


Figure 1 A cardinality based feature model of tourist guide SPL

2.2 Feature Model Errors

A feature model is designed to represent a software product line (SPL) and should include all potential member products of the SPL. Feature modelers use feature relationships to constrain the set of products that can be derived from a feature model. However, there may exist contradictory feature relationships which will cause feature model errors, such as dead features, false variable features and wrong cardinalities. These feature model errors prevent a feature model from representing the right set of products desired by the SPL domain. *Trinidad* et al. [5] has defined the two most critical feature model errors as follows:

- Dead features. A dead feature is a non-instantiable feature, i.e. a feature that despite of being defined in a feature model, it appears in no product in software product line.

- False variable features. A child feature in a non-mandatory relationship is a false variable feature if it has to be instantiated whenever its parent feature is selected.

Based on the above definition of dead features and false variable features, to check whether a feature *f* is a dead feature, we assume that *f* is not dead and include *f* into a product. When *f* is included, its neighbor features, such as parent, child, dependent features, will be included or excluded based on their specific relationship with *f*. Similarly, the inclusion or exclusion of the neighbor features will result in inclusions or exclusions of their neighbor features as well, and so forth. This transmission process through different feature relationships is called feature relationship propagation. If the inclusion of *f* will lead to a contradictory conclusion of exclusion of *f* through feature relationship propagation, the original assumption that *f* is not a dead feature is wrong. Then we can conclude that *f* is a dead feature. In a similar way, to check whether a feature *f* is a false variable feature, we assume that *f* is not false variable and remove *f* from products. If the exclusion of *f* will lead to a contradictory conclusion of inclusion of *f* through feature relationship propagation, the

original assumption that *f* is not a false variable feature is wrong. We can then conclude that *f* is a false variable feature.

In this paper, we concentrate on identifying dead features and false variable features through the feature relationship propagation. As feature relationships constrain the selection of features in product configuration, feature relationship propagation can also be called constraint propagation. In this paper, an approach, called constraint propagation method, is developed to identify feature model errors, including dead features and false variable features. Details of this approach will be described in the next two sections.

3 Constraint Propagation

Constraint propagation is the process of propagating the decision on a variable feature, either removal or inclusion, to a set of other features in the feature model through different feature relationships, including hierarchical relationships and cross-tree feature dependencies. The constraint propagation is an iterative process. At the first propagation step, the inclusion or removal of a variable feature will be propagated to its neighbor features which are closely connected with the variable feature by one single feature relationship. Then each newly removed or included feature in the last propagation step will propagate the decision on it to its neighbor features iteratively. The propagation process will stop until no more features can be removed or included. The key issue of constraint propagation is the propagation rules about how to propagate the decision on a feature to its neighbor features. To propose the propagation rules, we first give some definitions that are used to explain the propagation rules.

Definition 1: (feature attribute: status): Every feature has an attribute named as "status" that ranges over the domain {*-1*, *0*, *1*}. The attribute "status" gets the value "*1*" if the feature is included to be a part of the product, gets the value "*-1*" if the feature is removed from the product and otherwise "*0*".

Definition 2: (action: include or remove): In a product configuration, the action of changing the status of a feature *f* from "*0*" to "*1*" is named as "include" and represented as *INclude (f)* while the action of changing the status of a feature *f* from "*0*" to "*-1*" is named as "remove" and represented as *REmove (f)*.

Definition 3: In a feature group $FG = \{f_1, f_2 \ldots f_n\}$, the number of features whose statuses are "*1*" is represented as *NumberofINcluded (FG)* while the number of features whose statuses are "*-1*" is represented as *NumberofREmoved (FG)*. The number of features in *FG* is represented as *NumberofFeatures (FG).*

Definition 4: (neighbor features): The neighbor features of a certain feature *f* include all the features that connect with *f* by a single feature relationship, including mandatory, optional, feature group with cardinality, requires or excludes. We represent the neighbor features of a certain feature *f* using a set of notations as follows:

- Parent Feature*: f* has a parent feature if *f* is not root feature of a feature model. We use *f. parent* to represent the parent feature of *f*. If *f* is a root feature, *f. parent* has the value "*null*". We use *f. pr* to represent the hierarchical relationship the connects feature *f* with its parent feature:
  - *f. pr = m*, if a mandatory relationship connects feature *f* with its parent feature.
  - *f. pr = o*, if an optional relationship connects feature *f* with its parent feature.
  - *f. pr = c*, if a feature group with cardinality connects feature *f* with its parent feature.
- Child Feature: *f* has a number of child features if *f* is not leaf feature of a feature model. We use *f. set (child)* to represent the set of child features of feature *f*. The feature set *f. set (child)* can be decomposed into several subsets based on the hierarchical relationships that connect *f* with its child features: $f.\ set\ (child) = f.\ set\ (m) \cup f.\ set\ (o) \cup f.\ set\ (c_1) \ldots \cup f.\ set\ (c_n)$.
  - The notation *f. set (m)* illustrates the set of child features which connect with *f* by mandatory relationships.
  - The notation *f. set (o)* illustrates the set of child features which connect with *f* by optional relationships.
  - The notation $f.\ set\ (c_i)$ illustrates a feature group under *f*. Feature *f* may correspond to a number of feature groups and we use $\{f.\ set\ (c_1), f.\ set\ (c_2) \ldots f.\ set\ (c_n)\}$ $(n>=0)$ to represent the feature groups under feature *f*.
- Brother Features: *f* has a set of brother features if *f* belongs to a feature group. We use the notation *f. set (brother)* to represent the set of features which belong to the same feature group with *f*.
- Friend Features: *f* has a set of friend features if *f* connects with other features by requires or excludes. We use the notation *f. set (friend)* to represent the set of features which connect with *f* by cross-tree feature dependencies. The feature set *f. set (friend)* can be decomposed into three subsets based on different feature dependency types that connect *f* with its friend features: $f.\ set\ (friend) = f.\ set\ (reqed) \cup f.\ set\ (reqs) \cup f.\ set\ (exc)$.
  - The notation *f. set (exc)* represents the set of features that *f* excludes.
  - The notation *f. set (reqed)* represents the set of features that require *f*.
  - The notation *f. set (reqs)* represents the set of features that *f* requires.

Based on the above notations, we can represent the neighbor features for any feature in a feature model. In table 2, we take two features "devices" and "satellite" in the tourist guide feature model of Fig. 1 to illustrate how to represent the neighbor features of a feature in a feature model.

Table 2 The neighbour features of "devices" and "satellite" in tourist guide SPL feature model of Fig. 1.

feature *f*	devices	satellite
f. parent	operating environment	connection
f. pr	m	c
f. set (m)	∅	∅
f. set (o)	∅	∅
f. set (c_1)	mobile, laptop	∅
f. set (brother)	∅	LAN, wireless
f. set (exc)	∅	∅
f. set (reqed)	∅	position detection
f. set (reqs)	∅	∅

The action on a certain feature has different impacts to its neighbor features based on the semantics of different feature relationships connecting the certain feature and its neighbor features. Based on the semantics of feature relationships in table 1, the detailed propagation rules from a variable feature *f* to its different kinds of neighbor features are described as follows:

INclude (f): when *f* is included into products, the neighbor features of *f* will be affected as follows:

- Parent feature: when *f* is not the root feature, if *f* is included, its parent feature will be included. The propagation rule is:

1. $(f.parent != null) \Rightarrow INclude(f.parent)$

- Child feature: when *f* is not leaf feature and there are a set of child features *f. set (m)* that connect with *f* by mandatory relationships, if *f* is included into products, all the features in *f. set (m)* will be included. When there is a feature group *f. set (c_i)* with group cardinality [*a…b*] under *f*, if *f* is included and the number of removed features in *f. set (c_i)* reaches its maximum value (*NumberofFeatures (f. set (c_i))* – *a*), all features that are not determined in *f. set (ci)* will be included. The two propagation rules are:

2. $(f.set(m) != \varnothing) \Rightarrow \forall f_i \in f.set(m) \mid INclude(f_i)$

3. $NumberofFeatures(f.set(ci)) - a = NumberofREmoved(f.set(ci)) \Rightarrow \forall f_i \in f.set(ci) \wedge f_i.status = 0 \mid INclude(f_i)$

- Brother feature: when *f* is in a feature group with group cardinality [*a…b*] and has a set of brother features *f. set (brother)*, if *f* is included and the number of included features in *f. set (brother)* reaches its maximum value (*b-1*), all the features in *f. set (brother)* that are not determined will be removed. The propagation rule is:

4. $NumberofINcluded(f.set(brother)) = b-1 \Rightarrow \forall f_i \in f.set(brother) \wedge f_i.status = 0 \mid REmove(f_i)$

- Friend feature: when there exists a set of features that *f* requires and a set of features that *f* excludes, if *f* is included, all features in *f. set (reqs)* will be included and all the features in *f. set (exc)* will be removed. The propagation rules are:

5. $f.set(reqs) != \varnothing \Rightarrow \forall f_i \in f.set(reqs) \mid INclude(f_i)$

6. $f.set(exc) != \varnothing \Rightarrow \forall f_i \in f.set(exc) \mid REmove(f_i)$

REmove (f): when *f* is removed from products, the neighbor features of *f* will be affected as follows:

- Parent feature: when *f* is not the root feature and *f* connects with its parent feature by mandatory relationship, if *f* is removed, its parent feature will be removed. When *f* is in a feature group with group cardinality [*a…b*] and has a set of brother features *f. set (brother)*, if *f* is removed and the number of removed features in *f. set (brother)* exceeds its maximum value *NumberofFeatures* (*f. set (brother)*)-*a*, the parent feature of *f* will be removed. The propagation rules are:

7. $(f.parent != null) \wedge (f.pr = m) \Rightarrow REmove(f.parent)$

8. $(f.parent.status = 0) \wedge (NumberofREmoved(f.set(brother)) > NumberofFeatures(f.set(brother)) - a) \Rightarrow REmove(f.parent)$

- Child feature: when *f* is not leaf feature and it has a set of child features *f. set (child)*, if *f* is removed, all the features in *f. set (child)* will be removed. The propagation rule is:

9. $f.set(child) != \varnothing \Rightarrow \forall f_i \in f.set(child) \mid REmove(f_i)$

- Brother feature: when *f* is in a feature group with group cardinality [*a…b*] and has a set of brother features *f. set (brother)*, if the parent feature of *f* is included and the number of removed features in *f. set (brother)* reaches its maximum value, all the features in *f. set (brother)* that are not determined will be included. The propagation rules is:

10. $(f.parent.status = 1) \wedge (NumberofREmoved(f.set(brother)) = NumberofFeatures(f.set(brother)) - a) \Rightarrow \forall f_i \in f.set(brother) \wedge f_i.status = 0 \mid INclude(f_i)$

- Friend feature: when there exists a set of features *f. set (reqed)* that requires *f*, if *f* is removed, all the features in *f. set (reqed)* will be removed. The propagation rule is:

$$11.\quad f.set(reqed) != \varnothing \Rightarrow \forall f_i \in f.set(reqed) \mid REmove(f_i)$$

4 Identifying Feature Model Errors

In this section, we develop a recursive algorithm to identify dead features and false variable features for a given feature model *FM* that is received as input in Algorithm 1. We traverse all the variable features (*FM. vars*) of feature model *FM*. For each variable feature *var*, we check whether the inclusion of *var* will result in contradictions through constraint propagation (line 3) and check whether the removal of *var* will result in contradictions through constraint propagation (line 9). Before identifying each error, the feature model will be initialized by function *REset* (*FM*) which sets the status of all full-mandatory features [12] as "1" and the status of all other features as "0". It should be noted that we set the status of the parent feature of *f* as "1" when we check whether *f* is a false variable feature based on the definition on false variable features in section 2.2.

Algorithm 1: Feature Model Error Identification

Parameters:

FM: the given feature model

Function **identifyErrors** (*FM*)

```
1.  for (each var in FM. vars) {
2.      REset (FM);
        //check whether var is a dead feature;
3.      if (propagate (var, 1)= cfs)
4.      printout "var is dead feature";
5.  end for
6.  for (each var in vars) {
7.      Reset (FM);
8.      var. parent. status = 1;
        //check whether var is a false variable feature;
9.      if (propagate (var, -1) = cfs)
10.     printout "var is false variable feature";
11. end for
```

The constraint propagation from a variable feature to other features is achieved through a recursive function "*propagate*" which takes two parameters: a feature *f* where the propagation starts in one propagation step and value *v* that is assigned to *f*. The "*v = 1*" illustrates the inclusion of *f* while "*v = -1*" illustrates the removal of *f*. The propagation process returns to the last recursive step in three conditions: if the status of *f* is different with *v*, a contradiction arises and we return to the last recursive step with a sign "*cfs*" which represents contradictory assignments (line 13); if the status of feature *f* is same with *v*, a overlapped assignment arises and we return to the last recursive step with a sign "*ols*" which represents overlapped assignments (line 16); if no contradictions arise after propagating value *v* of *f* to other features, we return to the last recursive step with a sign "*no propagation*" (line 33). The constraint propagation from *f* to its neighbor features is realized from line 19 to line 32. The function "*rule (fn, f, v)*" returns the value assigned to *fn* which is a neighbor feature of *f*, when *v* is assigned to *f* based on the propagation rules proposed in last section.

Function: Constraint Propagation

Parameters:

f: the feature where the propagation starts in one recursive step.

v: the value assigned to *f* based on the action taken on *f*: v = 1 when selecting *f* and v = -1 when removing *f*

Function **propagate** (*f, v*)

```
12. if (f. status = -v)          //contradictory assignments
13.     return cfs;
14. end if
15. if (f. status = v)
16.     return ols;              //overlapped assignments
17. end if
18. f. status = v                // assigning f with v
19. if (v = = 1)  // propagation from f when f is included
20.     for each neighbor feature fn of f
21.         if (propagate (fn, rule (fn, f, 1)) = = cfs)
22.             return cfs;
23.         end if
24.     end for
25. end if
26. if (v = = -1)     // propagation from f when f is removed
27.     for each neighbor feature fn of f
28.         if (propagate (fn, rule (fn, f, -1)) = = cfs)
29.             return cfs;
30.         end if
31.     end for
32. end if
33. return no-propagation;
```

5 Evaluation

In this section, we aim to evaluate our approach from two aspects: the correctness of the identified feature model errors and the efficiency for identifying feature model errors. We propose an evaluation process illustrated in Fig. 2. *FAMA* [13] which uses *CSP* solvers to identify and explain feature model errors is chosen as the contrast to evaluate our approach, as *FAMA* has been widely used in validating feature models and evaluated by many researchers. For a specific feature model, we use *FAMA* tool as well as our algorithm to identify feature model errors. The correctness of our approach can be verified by comparing the errors identified by our algorithm with the errors identified by *FAMA*. The efficiency of our approach can be assessed by comparing the time spent by *FAMA* with the time spent by our algorithm.

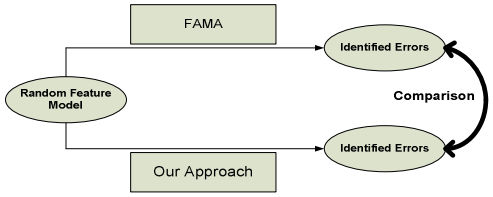


Figure 2 An evaluation process for correctness and efficiency of our approach

5.1 Experimental Platform

To perform our experiments, we develop our algorithm in Java, as *FAMA* is an Eclipse Project. The experiments were performed on a computer with an AMD 1.99 GHz CPU, 2 G of memory, Windows XP and 1.6 Java Virtual Machine (*JVM*). The maximum memory size of JVM is 512m (-Xmx512m) while the minimum memory size of JVM is 256m (-Xmx256m).

5.2 Random Feature Model

Feature model errors always exist in large-size feature models which have hundreds of thousands of features and feature relationships. Also, the advantage of our approach relies on the algorithm efficiency for identifying errors for large-size feature models. Therefore, the evaluation of our approach should be based on large-size feature models. It is difficult to obtain large-size feature model in real world SPL. To deal with this problem, we develop an algorithm which can generate random feature models. When generating the random feature models, we can set several parameters:

- the number of features in the feature model (*num_f*)
- the number of "requires" dependency in the feature model (*num_req*)
- the number of "excludes" dependency in the feature model (*num_exc*)
- the maximum number of mandatory child features of a certain feature (*max_mc*)
- the maximum number of optional child features of a certain feature (*max_oc*)
- the maximum number of feature groups under a feature (*max_gc*)
- the maximum number of features in a feature group (*max_var*)

Among these parameters, *num_f*, *num_req* and *num_exc* determine the size and complexity of a feature model while *max_mc*, *max_oc*, *max_gc* and *max_var* affect the depth and width of a fixed-size feature model. The values of *max_mc*, *max_oc*, *max_gc* and *max_var* should be set based on *num_f* to avoid that the feature model is too wide or too deep.

5.3 Correctness

The correctness of our approach heavily relies on whether the identified feature model errors by our algorithm are complete and correct. To evaluate the correctness of our approach, we generate 10 random feature models where the number of features ranges from 100 to 1000 and the number of dependencies ranges from 10 to 100. The parameters of the generation algorithm are set as follows: *max_mc*=3, *max_oc*=3, *max_gc*=3, *max_var*=4. Based on our experiments, when the number of features ranges from 100 to 1000, these parameters can satisfy that the feature model is neither too wide nor too deep.

Table 3 Dead features and false variable features detected by FAMA and our algorithm

Feature model	num_f	num_ req	num_ exc	FAMA		Our Algorithm	
				dead	false variable	dead	false variable
1	100	5	5	8	0	8	0
2	200	15	15	8	1	8	1
3	300	20	20	27	3	27	3
4	400	25	25	29	0	29	0
5	500	30	30	24	24	24	24
6	600	35	35	13	0	13	0
7	700	40	40	31	5	31	5
8	800	45	45	no result		56	3
9	900	50	50	no result		21	27
10	1000	50	50	no result		76	3

The number of dead features and false variable features identified by FAMA as well as our algorithm is shown in table 3. By comparison, the errors identified by our algorithm are the same as the errors detected by *FAMA* except for feature model 8, 9 and 10. For these three feature models, *FAMA* meets "out of heap memory" error and cannot identify feature model errors. However, the correctness of our approach can be verified by the other seven feature models in table 3.

5.4 Efficiency

The efficiency of our algorithm is reflected in identifying feature model errors for large-size feature models. In this experiment, we record the time spent by *FAMA* and our algorithm in identifying and explaining feature model errors for the ten feature models in table 3. The results are shown in table 4.

Table 4 The time spent for identifying feature model errors by FAMA and our approach

Feature model	num_f	num_req	num_exc	Our Algorithm (ms)	FAMA (ms)
1	100	10	10	15	1875
2	200	15	15	15	2203
3	300	20	20	143	19854
4	400	25	25	129	20345
5	500	30	30	986	125743
6	600	35	35	45	6534
7	700	40	40	1572	153983
8	800	45	45	1856	out_of_memory
9	900	50	50	1678	out_of_memory
10	1000	50	50	2145	out_of_memory

In table 4, the time unit is millisecond and from the result we can find that *FAMA* reveals a weak time performance when identifying feature model errors on medium and large size feature models and even meets "out of memory" error if the

number of features is larger. Compared with *FAMA*, our approach reveals a strong time performance. This is because *FAMA* adapts constraint programming to identify feature model errors and the number of solutions that need to be checked is 2^n where n is the number of features. Our approach significantly improves the efficiency because at most n^2 (n is the number of features) steps need to be traversed.

6 Conclusion and Future Work

In the area of feature model validation, a set of approaches have been proposed to identify feature model errors [2-7]. However, there is a critical limitation in the existing approaches. These approaches are inefficient because an exponential number of solutions need to be checked when identifying a feature model error for large-size feature models. To overcome this limitation, we develop a constraint propagation based algorithm to identify dead features and false variable features. The correctness and efficiency of our algorithm are evaluated based on a set of large size feature models which are randomly generated in our approach. In the future, we aim to detect the explanations for the identified errors in the constraint-propagation process.

7 Reference

[1] Kyo Chul Kang, "FODA: Twenty Years of Perspective on Feature Models," the keynote of SPLC 2009, San Francisco, CA, USA, 2009.

[2] D.Batory, D.Benavides and A.Ruiz-Cortes, "Automated Analysis of Feature Models: Challenges Ahead," in Communications of the ACM Vol. 49, No. 12. (2006) 45-47.

[3] David Benavides, Sergio Segura, Antonio Ruiz-Cortes, "Automated analysis of feature modes 20 years later: a literature review," in the journal of informaiton systems, 2010.

[4] Von der Massen, T., Lichter, H., " Deficiencies in feature models," in workshop on software variability management for product derivation-towards tool support.

[5] P. Trinidad *, D. Benavides, A. Duran, A. Ruiz-Cortes, M. Toro, "Automated error analysis for the agilization of feature modelling," in Journal of Systems and Software, 2007.

[6] A. Hemakumar, "Finding contradictions in feature models," in proceedings of the first international workshop on analysis of software product lines, 2008, pp. 183-190

[7] K. Czarnecki, S. Helsen and U. Eisenecker, "Formalizing cardinality-based feature mdoels and their specialization," in the journal of software process: improvement and practice 10 (1) (2005) 7-29.

[8] Kyo C. Kang, G. C. S., J. A. Hess, W.E. Novak and A. S. Petersem (1990). Feature-Oriented Domain Anaylisis (FODA) Feasibility Study. Technical Report CMU/SEI 90-TR-21, 1990.

[9] Czarnecki, K. Kim, P. , "Cardinality-based feature modelling and constraints: a progress report," in the proceeding of the International Workshop on Software Factories at OOPSLA 2005.

[10] David Benavides, Sergio Segura, Antonio Ruiz-Cortes, "Automated analysis of feature modes 20 years later: a literature review," in the journal of informaiton systems, 2010.

[11] M. Griss, J. Favaro, and M. d'Alessandro, "Integrating feature modeling with the RSEB," in Proceedings of the Fifth International Conference on Software Reuse, pages 76–85, Canada, 1998.

[12] H. Ye, Y. Lin, and W. Zhang, "Streamlined Feature Dependency Representation in Software Product Lines,"in international conference on software engineering research& practtice, Las Vegas, Nevada (2010)

[13] D, Benavides, S Segura, P Trinidad, A Ruiz-Cortés, "FAMA: Tooling a Framework for the Automated Analysis of Feature Models," in proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS), 2007.