Detection of Dead Features in Feature Models

Hong Mei, Wei Zhang\*, Li Yi, Haiyan Zhao, Zhi Jin

Key Laboratory of High Confidence Software Technology (Peking University), Ministry of Education, China.

Institute of Software, School of EECS, Peking University, Beijing, 100871, China.

meih@pku.edu.cn, zhangw@sei.pku.edu.cn, zhhy@sei.pku.edu.cn, yili07@sei.pku.edu.cn, zhijin@sei.pku.edu.cn

\* Corresponding author. Tel.: +86 10 62759074; fax: +86 10 62751792

**Abstract**: **Context:** This paper focuses on the detection of dead features in feature models. **Objective:** The objective of this paper is to find an efficient method to detect dead features. In particular, we focus on how to detect all the dead features in a feature model by invoking a *SAT-Solver* as few times as possible. **Method:** To achieve this objective, we distinguish two kinds of dead features: trivial dead features and nontrivial dead features, and prove that in order to detect all the dead features in a feature model, we only need to detect those nontrivial dead features. Based on that, we proposed an abstract algorithm to detect dead features, and identified fifteen specializations of the abstract algorithm. After that we carried out a set of experiments to evaluate the efficiency of the fifteen specialized algorithms. **Results:** The experiment results show that in small feature models, the three kinds of leaf-first binary search algorithm and the three kinds of bottom-up linear search algorithm have the nearly same efficiency; while in large feature models the leaf-first binary search algorithms perform better. **Conclusion:** Our recommendation for detecting dead features in a feature model is a leaf-first binary search algorithm.

**1. Introduction**

Feature models provide an effective approach to manage and reuse requirements in software product lines (SPLs) [2,5]. One important issue related to feature models is called the verification problem. The purpose of feature model verification is to detect deficiencies in feature models, so as to avoid the transmission of these deficiencies into subsequent core-asset and product development activities in SPL engineering. Although many researchers have observed that the verification problem of feature models can be transformed into *SAT* problems and proposed to resolve these transformed *SAT* problems by invoking third-party’s *SAT-solver* or *model-checker* tools [1,4,8], few of them point out how to use these third-party tools efficiently.

In this paper, we focus on a specific kind of feature model verification problem: the detection of *dead features*. A dead feature is a feature that cannot be selected in any valid feature model configuration. The problem of checking the deadness of a certain feature is equivalent to the SAT problem corresponding to the original feature model except that the feature must be set to True. In order to detect all dead features, a traditional yet trivial method is to check the features one by one. This method would take too much time to be practical for large feature models in real world that contain up to ten thousands features and nearly the same amount of constraints. However, there still lacks an exploration of efficient methods other than the trivial one. In this paper, we explore how to decrease the number of features needed to be checked during the detection of all dead features in a feature model, and thus improve the detection efficiency. In our approach, we distinguish two kinds of dead features: trivial and nontrivial dead features, and prove that in order to detect all the dead features in a feature model, we only need to detect those nontrivial dead features. Based on that, we propose an abstract algorithm to detect dead features, and identify fifteen specializations of the abstract algorithm. After that we carry out a set of experiments to evaluate the efficiency of the fifteen specialized algorithms. The experiment results show that in small feature models, the three kinds of leaf-first binary search algorithm and the three kinds of bottom-up linear search algorithm have the nearly same efficiency; while in large feature models the leaf-first binary search algorithms perform better.

The rest of this paper is organized as follows. Section 2 gives some preliminaries of feature models and the verification of feature models. Section 3 introduces some basic concepts and theorems about dead features, and proposed an abstract algorithm to detect dead features and fifteen specializations of the abstract algorithm. Experiments and analysis are shown in Section 4. Related works are discussed in Section 5. Finally, Section 6 concludes this paper and discusses future work.

**2. Preliminaries**

In this section, we introduce some preliminaries of our approach, with the purpose of building a clear understanding of feature models and dead features in feature models.

**2.1 Feature Models**

The motivation behind feature models is to find a practical technique for the modeling and reusing of reusable requirements in a software product line. This motivation is derived from a paradigm for domain-oriented software reuse [3, 15], which consists of two basic activities: domain engineering (*a.k.a.* core asset development), and application engineering (*a.k.a.* product development). In the former activity, reusable assets in a software domain are produced or modeled based on existing applications in this domain. In the latter one, these assets are consumed (i.e. reused) to produce new applications/products in this domain. Directed by such a paradigm, feature models are proposed to support the modeling and reusing of reusable assets at the requirements level, that is, to support the modeling and reusing of reusable requirements.

说明: Metamodel%20of%20FM

**Fig. 1.** An abstract metamodel of feature models.

The structure of feature models can be generally summarized as *a set of features* with *a set of relations among features* (see Fig. **1**). Each feature encapsulates a cohesive set of individual requirements, and serves as a basic unit in requirements reuse. There are two common kinds of relation among features: *refinement*, and *constraint*. The purpose of the refinement relation is to organize the usually large number of features in a software product line into a tree structure, in which high-level abstract features are gradually refined into low-level concrete features. The purpose of the constraints relation is to model the dependencies among features that must be satisfied when doing customization on feature models.

A formal definition of feature models is described in the following definition.

**Definition 1.** (Feature Models)

A *feature model* is a 6-tuplet (*F*, *root*, *Refine*, *BS*, *RDC, EAC*)[[1]](#footnote-1), where:

* *F* is a set of *n* features {*f1*, *f2*, …, *fn*}.
* *root* is a feature: *root* ∈ *F*.
* *Refine* is a partial function *F* →*F*, which satisfies

(1) ∀ *f* ∈ *F* ⋅ (*f*, *f*) ∉ *Refine* and (*root*, *f*) ∉ *Refine*.

(2) ∀ *f1, f2* ∈ *F* ⋅ (*f1, f2*) ∈ *Refine* implies (*f2*, *f1*) ∉ *Refine.*

(3) ∀ *f* ∈ *F* ⋅ Exists a sequence of features (*f1, f2, …, fk*) that (*f, f1*) ∈ *Refine* and (*fk, root*) ∈ *Refine,* and (*fi, fi+*1) ∈ *Refine* for 1≤ *i* < *k.*

Through the *Refine* function, features in *F* are formed as a tree structure with *root* as its root node. The physical meaning of a pair (*fi*, *fj*) ∈ *Refine* is that *fi* is refined from *fj*. For convenience, we call *fj* the parent of *fi* and *fi* one of the children of *fj*. We use *f*.*children* to denote the feature set that contains all of *f*’s children, and *f*.*parent* the parent feature of *f*.

* *BS* is a three-valued predicate *F* →{*true*, *false*, *unknown*}. For a feature *fi*, *BS*(*fi*) denotes whether *fi* is selected (*true*), removed (*false*) from the feature model, or still undecided (*unknown*). Initially, each feature’s *binding-state* is undecided (and will be changed to selected or removed in later customization activities), except the root feature, whose binding-state is always *selected.* *BS* is used to record the customization result of a feature model. The customization of a feature model means to make customization decisions to those undecided features. A customization decision for an undecided feature decides whether to select this feature or to remove it from the current feature model.
* *RDC* =*def* { *BS*(*fi*) *BS*(*fj*) | (*fi*, *fj*) ∈ *Refine*} is a set of *n-1* constraints. Since these constraints are derived from the *Refine* relation, we call them *refine-derived constraints*. The physical meaning of constraints in *RDC* is that any child feature cannot be selected unless its parent feature has been selected. A further derivation is that if a feature is removed, than all its descendants must also to be removed; otherwise, at least one constraint in *RDC* will be violated.
* *EAC* is a set of *m* constraints {*c1*, *c2*, …, *cm*} explicitly added by the feature model’s constructors, and we call them *explicitly-added constraints*. For simplicity, we suppose each constraint in *EAC* is a *CNF* clause (i.e. a disjunction of literals), in which, a literal either has a form of *BS*(*fi*) or its negation. A basic property of *EAC* is that *BS*(*root*) ∈ *EAC* (that is, the root feature should be selected in any valid feature model configuration). This property is to avoid the situation of a valid feature model configuration with no feature: if the root feature is removed, in order to satisfy all constraints in *RDC*, all the other features must be removed as well, then we will get a configuration with zero features, a configuration that has nonsense in practice.

The reusing of requirements in feature models is usually carried out through a customization-based approach (see Fig. 2). In the core-asset development activity, after a feature model is constructed, the *model-level verification* is carried out to detect possible deficiencies in the feature model. If deficiencies are detected, then the feature model should be reconstructed to eliminate these deficiencies; if no deficiency is detected, then the feature model is allowed to be further customized. In the product development activity, after customization, the *customization-level verification* is carried out to detect the violation of constraints. If any constraint is violated, a further customization will be carried out to adjust the current result of customization, and thereby resolve the violation of constraints.

说明: Reusing%20Requirements%20in%20Feature%20Models

**Fig. 2.** Reusing requirements in feature models.

**2.2 Dead Features**

In this paper, we focus on the *model-level verification* of feature models. In particular, we only focus on a special kind of deficiency in feature models: *the dead features*.

**Definition 2.** (Dead Features)

Given a feature model *FM* = (*F*, *root*, *Refine*, *BS*, *RDC, EAC*), and a feature *f* ∈ *F*, *f* is a *dead feature*, iff

* is unsatisfiable*.

Informally speaking, a dead feature is a feature that can not be selected in any valid feature model configuration, because that if a dead feature is selected, at least one constraint in *RDC* or *EAC* will be violated.

Obviously, deciding whether a feature is dead or not is a SAT problem. Therefore, we can resolve this SAT problem by invoking a SAT-Solver (a program that checks whether a CNF propositional formula is satisfiable or not).

It is easy to detect all dead features in a feature model in a traversal-based way: we can traverse all features in the feature model, and for each feature, invoke a SAT-Solver to decide whether it is dead or not. In such a way a SAT-Solver is invoked exactly once for each feature.

However, not all the invocations of a SAT-Solver are necessary, due to *Refine* and *RDC* of a feature model. In the following section, we will explain why some of the invocations are unnecessary and present a set of algorithms to reduce the invocations as few as possible, and in turn, improve the efficiency of dead feature detection.

**3. Detection of Dead Features**

In this section, we first introduce some basic concepts and theorems about dead features. Then we propose an abstract algorithm for dead feature detection. After that, we present a set of possible specializations to this abstract algorithm and their implementations.

**3.1 Basic Concepts and Theorems**

**Definition 3.** (Trivial Dead Features)

Given a feature model *FM* = (*F*, *root*, *Refine*, *BS*, *RDC, EAC*), and a dead feature *f* ∈ *F*, *f* is a *trivial dead feature*, iff

* is unsatisfiable*.

**Definition 4.** (Nontrivial Dead Features)

Given a feature model *FM* = (*F*, *root*, *Refine*, *BS*, *RDC, EAC*), and a dead feature *f* ∈ *F*, *f* is a *nontrivial dead feature*, iff

* is satisfiable*.

**Definition 5.** (Refinement Paths)

Given a feature model *FM* = (*F*, *root*, *Refine*, *BS*, *RDC, EAC*) and a feature sequence *P* = < *p0*, *p1*, …, *pi*, *pi+1*, …, *pl* >, *P* is called a *refinement path* in *FM*, iff it satisfies the following two properties:

* Any node in *P* is a feature in *F*.
* For any adjacent two features *pi* and *pi+1* in P, *pi* is *pi+*1’s parent feature.

**Lemma 1.** Any ancestor of a non-dead feature is also a non-dead feature.

**Proof:** Suppose *f* and *a* are two features in a feature model *FM* = (*F*, *Root*, *Refine*, *BS*, *RDC, EAC*), *f* is a non-dead feature, and *a* is an ancestor of *f*. The definition of dead features implies that ** is satisfiable. According to *RDC*, we can deduce that **. As a result, we can conclude that **. That is, *a* is a non-dead feature. Therefore, this lemma is proven.

**Lemma 2.** Any descendant of a dead feature is also a dead feature.

**Proof:** Similar to the proof of Lemma 1.

**Theorem 1.** Any ancestor of a nontrivial dead feature is a non-dead feature, and any descendant of a nontrivial dead feature is a trivial dead feature.

**Proof:** From the definition of (non)trivial dead features, **Lemma 1**, and **Lemma 2**.

**Corollary 1.** Any refinement path in a feature model contains at most one nontrivial dead feature.

**Proof:** From **Theorem 1**.

**Corollary 2.** The number of nontrivial dead features in a feature model is not larger than the number of leaf features in the feature model.

**Proof:** We first prove that the corollary is true in the case that a feature model contains only one leaf feature. In such a case the feature model is reduced to a single refinement path. According to **Corollary 1**, this feature model contains at most one nontrivial dead feature. Therefore, the corollary is true in this case.

Suppose the corollary is true in the case that a feature model contains *n* () leaf features. Given a feature model *FM* that contains  leaf features, suppose *fl* is a leaf feature of *FM*, then there exists at least one ancestor of *fl* that contains two or more children features (otherwise, *FM* contains only oneleaf feature). Suppose *fa* is an ancestor of *fl* and *fa*.*parent* is the nearest ancestor of *fl* that contains two or more children features, then *FM* contains a refinement path *P* that starts with *fa* and ends with *fl*. By removing all the features in *P* from *FM*, we get a new feature model *FM’* that contains *n* leaf features. According to the assumption, *FM’* contains at most *n* nontrivial dead features; according to **Corollary 1**, the refinement path P contains at most one nontrivial dead feature. Then, we can deduce that *FM* contains at most ** nontrivial dead features. Therefore, this corollary is proven.

**3.2 An Abstract Algorithm to Detect Dead Features**

According to **Theorem 1**, in order to detect all the dead features in a feature model, we only need to locate those nontrivial dead features, from which trivial dead features can be deduced.

According to **Corollary 1**, in order to locate all the nontrivial dead features in a feature model, we can select a refinement path from the feature model and then locate the nontrivial dead feature in this path (if it exists), and we repeat such a procedure until all the nontrivial dead features are located.

To make the detection as efficient as possible, the selection of refinement paths should satisfy three obvious properties:

1. Different selected paths should not contain common features. If a feature appears in a selected path, then whether this feature is dead or not is determined after the path is checked, and therefore it is unnecessary to include this feature again in another selected path.
2. If the deadness of a feature has been determined, it should not be included in any refinement path to be selected later. The reason is similar to that for the first property.
3. The number of selected paths should be as few as possible. It means that if two selected paths can be merged into one refinement path, we should select the merged path instead.

An algorithm based on the above idea is presented as **Algorithm 1**.

**Algorithm 1.** (Detecting Dead Features in a Feature Model)

**Input**: A feature model *FM* = (*F*, *root*, *Refine*, *BS*, *RDC, EAC*) that satisfies the following property: ** is satisfiable; that is, there is no conflict among constraints in *FM* (if this property is violated, then all features are dead).

**Output**: A feature set containing all dead features in *FM*.

1. **detect\_dead\_features(*FM*: FeatureModel): Set<Feature>** {
2. Set<Feature> *dead\_features* = {};
3. Mark *root* as *determined*;
4. **while** (there is a feature in *F* that hasn’t been marked as *determined*) {
5. Select a refinement path *P* = <*p0*, …, *pl*> from *FM* that satisfies:
6. A. No feature in *P* has been marked as *determined*,
7. B. *p0*.*parent* has been marked as *determined*, and
8. C. *pl* is a leaf feature;
9. Locate the nontrivial dead feature in *P*;
10. Mark all features in *P* as *determined*;
11. **if** (*P* contains a nontrivial dead feature *f*){
12. Add *f* and all of *f*’s descendant features to *dead\_features*;
13. Mark *f* and all of *f*’s descendant features as *determined*;
14. }
15. }
16. **return** *dead\_features*;
17. }

**Fig. 3** gives an illustrative example of the execution of **Algorithm 1**. In this example, we assume that there are three nontrivial dead features: *h*, *n* and *c*.

In the first iteration, the refinement path <*a*, *d*> is selected. After trying to locate the nontrivial dead feature in this refinement path, feature *a* and *d* are marked as *determined*, and since no nontrivial dead feature is found, no feature is added to the *dead\_features* set.

In the second iteration, the refinement path <*e*, *h*, *l*> is selected; after trying to locate the nontrivial dead feature in this path, feature *e*, *h* and *l* are marked as *determined*, and since *h* is found as the nontrivial dead feature in this path, all of *h*’s descendants (i.e. feature *l* and *m*) are also marked as *determined*, and they are added to *dead\_features*.

****

**Fig. 3.** An illustrative example of **Algorithm 1**.

The similar iteration repeats six times until all features are marked as *determined*. It should be noticed that **Algorithm 1** is an abstract algorithm. It can be specialized into a set of different algorithms from two perspectives:

1. The policy to select a refinement path from a feature model. This algorithm has imposed three restrictions (see line 6, 7 and 8 in **Algorithm 1**) on the selected refinement paths. However, in each iteration, there might be more than one refinement path satisfying the restrictions, but the abstract algorithm doesn’t specify any policy to decide which path should be selected from the possible candidates.
2. The method to locate the nontrivial dead feature (if any) in a refinement path (see line 9 in **Algorithm 1**). For each selected refinement path, this algorithm does not specify any concrete method about how to locate the nontrivial dead feature in this path.

There are many possible specializations of the abstract algorithm. Here the key problem is to find which one gives the highest efficiency. In other words, which combination of *nontrivial-dead-feature-locating* method and *refinement-path-selecting* policy gives best efficiency of **Algorithm 1**. In the following two sub-sections, we will introduce a set of locating methods and selecting policies.

**3.3 Methods to locate the nontrivial dead feature in a refinement path**

Table 1 lists five different methods to locate the nontrivial dead feature in a refinement path. The concrete algorithms to implement the standard binary search methods are given in **Algorithm 2**.

Table 1. Five methods to locate the nontrivial dead feature in a refinement path

|  |  |  |
| --- | --- | --- |
| ID | Name | Description |
| 1 | Standard Binary Search | A method that uses a standard binary search to locate the nontrivial dead feature in a refinement path. |
| 2 | Leaf-First Binary Search | A method that first checks whether the leaf feature in a refinement path is dead or not, and if necessary, uses a standard binary search method to locate the nontrivial dead feature in the rest of the refinement path. |
| 3 | Root-First Binary Search | A method that first checks whether the root feature in a refinement path is dead or not, and if necessary, uses a standard binary search method to locate the nontrivial dead feature in the rest of the refinement path. |
| 4 | Top-Down Linear Search | A method that follows a top-down (from the root feature to the leaf feature) linear sequence to locate the nontrivial dead feature in a refinement path. |
| 5 | Bottom-Up Linear Search | A method that follows a bottom-up (from the leaf feature to the root feature) linear sequence to locate the nontrivial dead feature in a refinement path. |

**Algorithm 2.** (Detecting the Nontrivial Dead Feature in a Refinement Path: Standard Binary Search Method)

**Input**: A refinement path *P* = < *p0*, *p1*, …, *pl-1* > in a feature model: the first element (located by index *0*) denotes the root feature in *P*, the last element (located by index **) denotes the leaf feature in *P*, and the feature *P*[*0*].parent has been determined as a non-dead feature.

**Output**: An integer that indicates the index of the first dead feature (i.e. the dead feature that has a minimum index among all dead features) in the input; if *P* contains no dead feature, then the output is a negative integer.

1. **standard\_binary\_search(*P*: Array<Feature>): int** {
2. **int** *location* = *-1*;
3. **int** *mid* = *P*.*length*/*2*;
4. **bool** *is\_dead* = Check whether *P*[*mid*] is dead or not;
5. **if** (*is\_dead*){
6. **if** (*mid* == *0*) *location* = *0*;
7. **else** {
8. **int** *innerloc* = standard\_binary\_search(*P*.subArray(*0*, *mid-1*));
9. **if** (*innerloc* < *0*)
10. *location* = *mid*;
11. **else**
12. *location* = *innerloc*;
13. }
14. } **else**{
15. **if** (*mid* == *P*.*length-1*) *location* = *-1*;
16. **else**{
17. **int** *innerloc* = standard\_binary\_search(*P*.subArray(*mid*+*1*, *P*.*length-1*));
18. **if** (*innerloc* < *0*)
19. *location* = -*1*;
20. **else**
21. *location* = *mid*+*1*+*innerloc*;
22. }
23. }
24. **return** *location*;
25. }

**3.4 Policies to select a refinement path**

Table 2lists three policies to select a refinement path from a feature model. The concrete algorithms to implement the three policies are given in **Algorithm 3**.

Table 2. Three policies to select a refinement path from a feature model

|  |  |  |
| --- | --- | --- |
| ID | Name | Description |
| 1 | Shortest Path First | A policy that select a shortest path from the current feature model to locate the possible nontrivial dead feature in this shortest path. |
| 2 | Longest Path First | A policy that select a longest path from the current feature model to locate the possible nontrivial dead feature in this longest path. |
| 3 | Middle-Length Path First | A policy that select a middle-length path from the current feature model to locate the possible nontrivial dead feature in this middle-length path. |

**Algorithm 3.** (Selecting a Refinement Path: Three Policies)

**Input**: A set containing all of the determinedfeatures, and the name of policy.

**Output**: A refinement path *P = <p0, …, pl>* that satisfies three conditions:

A. all features in *P* have not been *determined*,

B. *p0.parent* has been *determined*, and

C. *pl* is a leaf feature.

1. **select\_path(*S:* Set<Feature>*, Policy:* String): Array<Feature>** {
2. // Find all possible path roots (*p0*) according to Condition *B*
3. Array<Feature> *path\_roots* = find\_possible\_path\_roots(*S*);
4. Array<Feature> *path\_leaves* = Find leaves under all possible path roots;
5. Compute depth of each *path\_leaf* (i.e. the length of its corresponding path);
6. Sort *path\_leaves* by their depth;
7. Feature *selected\_path\_leaf =* Select a feature from *path\_leaves* by *Policy*;
8. **return** construct\_path\_from\_leaf(*selected\_path\_leaf*);
9. }
10. **find\_possible\_path\_roots(*S*: Set<Feature>): Array<Feature> {**
11. Array<Feature> *result* = [];
12. **for** (**each** Feature *F* **in** *S*) {
13. **for** (**each** Feature *C* **in** *F.children*) {
14. **if** (*C* is non-determined) *result*.append(*C*);
15. }
16. }
17. **return** *result*;
18. }
19. **construct\_path\_from\_leaf(*L*: Feature): Array<Feature> {**
20. Array<Feature> *path* = [];
21. **for** (Feature *f* = *L*; *f* is not determined; *f* = *f.parent*) {
22. *path*.prepend(*f*);
23. }
24. **return** *path*;
25. }

Based on the three *refinement-path-selecting* policies and five *nontrivial-dead-feature-locating* methods, we get fifteen specialized algorithms of the original abstract algorithm. Then, a key problem is to find which one is the most efficient in detecting dead features. A series of experiments are conducted find the answer, as described in the next section.

**4. Experiments and Analysis**

In this section, we carry out a series of experiments with the purpose to find out the algorithm with the highest efficiency for dead feature detection and then give explanations to the results.

**4.1 Experiments**

We compare each algorithm’s cost of invoking SAT-Solvers (it depends on how many feature are actually *checked*) in order to *determine* the deadness of all features in a feature model. There are three variables depicting structural properties of feature models in our experiments.

The first variable is the number of nontrivial dead features. Section 3.1 shows that once a nontrivial dead feature is found, the deadness of its descendants can be directly determined without being checked. Therefore, more nontrivial dead features give more chances to save cost by skipping the check of their descendant features.

The second variable is the average length of refinement paths (i.e. the average height of the feature tree). It may affect the performance of different nontrivial dead feature locating methods. For example, the advantage of binary searches may not be obvious for short paths.

The third variable is the number of refinement paths (i.e. the number of leaf features). It may affect the performance of different path selecting policies.

The last two variables together affect the size of feature models (i.e. the number of features), which in turn, may affect the performance of different algorithms.

For the first variable, we conduct a simulation to theoretically check *how many invocations* of SAT-Solversare needed in average, for each algorithm. The reason is that it is not easy to obtain a feature model with exact numbers of nontrivial dead features. To do the simulation, we first select seven real feature models from the SPLOT website (http://www.splot-research.org) with the number of features ranging from 31 to 290 (see Table 3), and then assume that they contain zero to four nontrivial dead features (indicated by a Boolean “deadness” attribute for each feature). Then an “invocation of a SAT-Solver” is just a checking of the Boolean value. Since the *position* of nontrivial dead features is biased against different dead feature locating methods (e.g. root-first methods are always better if a nontrivial dead feature positions near the root), we simulate all possible cases that nontrivial dead features may locate to avoid any bias. Fig. 4 illustrates the possible cases of two nontrivial dead features. We run the algorithms on each case and compare the average numbers of deadness checking. The results are shown in Table 4.



**Fig. 4.** An example of positioning two nontrivial dead features.

Table 3. The 7 feature models used in the simulation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Number of Features | Number of Leaves | Tree Depth |
| 1 | Reference Management Software | 31 | 23 | 4 |
| 2 | DELL Laptop/Notebook Computers | 46 | 37 | 3 |
| 3 | PFTest1 | 56 | 40 | 5 |
| 4 | OW2-FraSCAti-1.4 | 63 | 48 | 5 |
| 5 | Billing | 88 | 74 | 4 |
| 6 | Xtext | 137 | 102 | 8 |
| 7 | Electronic Shopping | 290 | 194 | 12 |

For the second and the third variable, we utilized SAT4J [2] as the SAT-Solver and perform actual detection of dead features in two groups of randomly generated feature models. One group of feature models has a fixed number (100) of leaf features, and various average heights of feature trees, ranging from 3 to 21. By contrast, another group has a fixed average height (11) and various numbers of leaf features, ranging from 20 to 800. The running time of the algorithms is shown in Fig. 5.

Table 4. The average numbers of actually checked features in the simulation

(DF# = Dead Feature Number; L/M/S = Select Longest/Middle/Shortest Path)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FM  ID | DF# | Standard  Binary Search | | | Leaf-First  Binary Search | | | Bottom-Up  Linear Search | | | Root-First  Binary Search | | | Top-Down  Linear Search | | |
| L | M | S | L | M | S | L | M | S | L | M | S | L | M | S |
| 1 | 0 | 30.00 | 30.00 | 31.00 | **23.00** | **23.00** | **23.00** | **23.00** | **23.00** | **23.00** | 30.00 | 30.00 | 31.00 | 31.00 | 31.00 | 31.00 |
| 1 | 28.73 | 28.73 | 29.70 | 22.60 | 22.60 | **22.57** | 22.60 | 22.60 | **22.57** | 28.80 | 28.80 | 29.70 | 29.70 | 29.70 | 29.70 |
| 2 | 28.48 | 28.48 | 29.46 | 22.92 | 22.92 | 22.91 | **22.87** | **22.87** | 22.91 | 28.57 | 28.57 | 29.46 | 29.46 | 29.46 | 29.46 |
| 3 | 28.17 | 28.17 | 29.16 | 23.18 | 23.17 | 23.18 | **23.09** | **23.09** | 23.18 | 28.29 | 28.28 | 29.16 | 29.16 | 29.16 | 29.16 |
| 4 | 27.81 | 27.81 | 28.80 | 23.38 | 23.37 | 23.39 | **23.26** | **23.26** | 23.39 | 27.96 | 27.95 | 28.80 | 28.80 | 28.80 | 28.80 |
| 2 | 0 | 45.00 | 45.00 | 45.00 | **37.00** | **37.00** | **37.00** | **37.00** | **37.00** | **37.00** | 46.00 | 46.00 | 46.00 | 46.00 | 46.00 | 46.00 |
| 1 | 44.20 | 44.20 | 44.20 | **36.73** | **36.73** | **36.73** | **36.73** | **36.73** | **36.73** | 45.18 | 45.18 | 45.18 | 45.18 | 45.18 | 45.18 |
| 2 | 43.52 | 43.52 | 43.52 | **36.55** | **36.55** | **36.55** | **36.55** | **36.55** | **36.55** | 44.48 | 44.48 | 44.48 | 44.48 | 44.48 | 44.48 |
| 3 | 42.84 | 42.84 | 42.84 | **36.34** | **36.34** | **36.34** | **36.34** | **36.34** | **36.34** | 43.78 | 43.78 | 43.78 | 43.78 | 43.78 | 43.78 |
| 4 | 42.13 | 42.12 | 42.13 | 36.10 | **36.09** | 36.10 | 36.10 | **36.09** | 36.10 | 43.04 | 43.04 | 43.04 | 43.04 | 43.04 | 43.04 |
| 3 | 0 | 52.00 | 53.00 | 53.00 | **40.00** | **40.00** | **40.00** | **40.00** | **40.00** | **40.00** | 54.00 | 56.00 | 56.00 | 56.00 | 56.00 | 56.00 |
| 1 | 50.91 | 51.87 | 51.87 | 39.84 | **39.78** | **39.78** | 39.87 | **39.78** | **39.78** | 52.84 | 54.71 | 54.71 | 54.71 | 54.71 | 54.71 |
| 2 | 50.03 | 50.98 | 50.98 | 39.79 | **39.71** | **39.71** | 39.85 | **39.71** | **39.71** | 51.91 | 53.68 | 53.68 | 53.68 | 53.68 | 53.68 |
| 3 | 49.18 | 50.09 | 50.10 | 39.73 | **39.61** | **39.61** | 39.80 | **39.61** | **39.61** | 51.00 | 52.68 | 52.68 | 52.68 | 52.68 | 52.68 |
| 4 | 48.32 | 49.21 | 49.22 | 39.63 | **39.48** | **39.48** | 39.71 | **39.48** | **39.48** | 50.09 | 51.69 | 51.69 | 51.69 | 51.69 | 51.69 |
| 4 | 0 | 59.00 | 59.00 | 59.00 | **48.00** | **48.00** | **48.00** | **48.00** | **48.00** | **48.00** | 61.00 | 62.00 | 63.00 | 63.00 | 63.00 | 63.00 |
| 1 | 57.32 | 57.32 | 57.35 | 47.16 | **47.15** | **47.15** | 47.19 | **47.15** | **47.15** | 59.26 | 60.19 | 61.15 | 61.15 | 61.15 | 61.15 |
| 2 | 56.56 | 56.58 | 56.63 | 47.02 | **47.00** | 47.01 | 47.05 | 46.98 | 47.01 | 58.44 | 59.32 | 60.27 | 60.27 | 60.27 | 60.27 |
| 3 | 55.87 | 55.92 | 55.98 | 46.93 | **46.92** | **46.92** | 46.95 | 46.88 | **46.92** | 57.69 | 58.53 | 59.46 | 59.46 | 59.46 | 59.46 |
| 4 | 55.14 | 55.22 | 55.30 | 46.78 | 46.78 | 46.79 | 46.80 | **46.73** | 46.79 | 56.89 | 57.69 | 58.61 | 58.61 | 58.61 | 58.61 |
| 5 | 0 | 86.00 | 85.00 | 86.00 | **74.00** | **74.00** | **74.00** | **74.00** | **74.00** | **74.00** | 87.00 | 88.00 | 88.00 | 88.00 | 88.00 | 88.00 |
| 1 | 84.91 | 83.94 | 84.93 | **73.38** | **73.38** | **73.38** | **73.38** | **73.38** | **73.38** | 85.9 | 86.86 | 86.86 | 86.86 | 86.86 | 86.86 |
| 2 | 84.20 | 83.25 | 84.24 | 73.09 | **73.08** | 73.09 | **73.08** | **73.08** | 73.09 | 85.17 | 86.11 | 86.11 | 86.11 | 86.11 | 86.11 |
| 3 | 83.51 | 82.58 | 83.56 | 72.81 | **72.79** | 72.81 | **72.79** | **72.79** | 72.81 | 84.46 | 85.39 | 85.39 | 85.39 | 85.39 | 85.39 |
| 4 | 82.77 | 81.85 | 82.82 | 72.46 | 72.45 | 72.46 | **72.44** | 72.45 | 72.46 | 83.7 | 84.62 | 84.62 | 84.62 | 84.62 | 84.62 |
| 6 | 0 | 123.00 | 132.00 | 135.00 | **102.00** | **102.00** | **102.00** | **102.00** | **102.00** | **102.00** | 130.00 | 135.00 | 136.00 | 137.00 | 137.00 | 137.00 |
| 1 | 121.57 | 130.35 | 133.29 | 101.46 | 101.35 | **101.32** | 101.53 | 101.35 | 101.33 | 128.49 | 133.30 | 134.28 | 135.23 | 135.23 | 135.23 |
| 2 | 120.45 | 129.04 | 131.93 | 101.13 | 100.92 | **100.89** | 101.24 | 100.94 | 100.90 | 127.27 | 131.94 | 132.9 | 133.81 | 133.81 | 133.81 |
| 3 | 119.36 | 127.78 | 130.62 | 100.81 | 100.52 | **100.47** | 100.97 | 100.54 | 100.49 | 126.08 | 130.62 | 131.57 | 132.44 | 132.44 | 132.44 |
| 7 | 0 | 259.00 | 267.00 | 273.00 | **194.00** | **194.00** | **194.00** | **194.00** | **194.00** | **194.00** | 270.00 | 284.00 | 287.00 | 290.00 | 290.00 | 290.00 |
| 1 | 255.86 | 263.69 | 269.59 | 192.44 | 192.33 | **192.30** | 192.60 | 192.35 | 192.31 | 266.71 | 280.37 | 283.35 | 286.30 | 286.30 | 286.30 |
| 2 | 253.96 | 261.66 | 267.51 | 191.79 | 191.59 | **191.54** | 192.04 | 191.62 | 191.55 | 264.71 | 278.12 | 281.09 | 284.01 | 284.01 | 284.01 |

****

**Fig. 5.** The impact of average height and number of leaves. Left: The number of leaves is fixed to 100, while the average height ranges from 3 to 21. Right: The average height is fixed to 11, while the number of leaves ranges from 20 to 800.

**4.2 Analysis**

From the experiment results shown in Table 4, we can observe that our algorithms reduce the average times of invoking third party solvers by nearly 30%, at most, comparing to a traditional traversal-based approach (in which solvers are invoked once for each feature). The leaf-first binary search algorithms and the bottom-up linear search algorithms give the best yet nearly the same efficiency, while the top-down linear search algorithms are the worst. The small size of the feature models affects benefits brought by binary search methods. In addition, there are no significant differences between the refinement path selecting policies.

From the experiment results shown in Fig. 5, we can observe that the leaf-first binary search algorithms are more stable and perform better in most situations, especially for large feature models. For example, in the case (2a), the bottom-up algorithms fluctuate a lot when there are 500 or more leaf features, while the running time of the leaf-first binary search algorithm increases slowly and smoothly. It should be noticed that for some algorithms, sometimes the running time decreases despite of the increment of tree height or leaf features. The reason is that the generated feature models happen to have a considerable number of nontrivial dead features in such cases (in one case there are 33 nontrivial dead features in a feature model containing about 3000 features). Also the decrements happen at different points for different algorithms, depending on the position of nontrivial dead features. In addition, the curves’ shapes are similar despite of different refinement path selecting policies. It again indicates that there is no significant difference between these policies.

Based on the above analysis, our recommendation for detecting dead features is that: in a relatively small feature model, either a leaf-first binary search algorithm or a bottom-up linear search algorithm can be selected; if the feature models become large (e.g. contains thousands of features), a leaf-first binary search algorithm is preferred.

**5. Related Work**

The approach proposed in this paper is a succeeded work of our previous research on the optimization of feature model verification. In our previous research, we have proposed two techniques to reduce the number of features and constraints to be checked during the verification of a feature model. One is the *atomic-set* technique that treats a set of features as a single feature [10], and the other one is an optimization strategy that removes verification-irrelevant features and constraints from a feature model [9]. It should be noticed that the approach proposed in this paper is orthogonal to the above two techniques, and thus can be safely integrated together in feature models’ verification.

Existing approaches to feature models’ verification can be generally classified into two categories. One category consists of those approaches based on third-party tools. In these approaches, the verification problem of feature models are firstly transformed into *SAT*, *CSP*, or other kinds of well-resolved formal problems, and then a third-party tool is invoked to find solutions of these transformed problems. The approach proposed in this paper can be classified into this category. However, as far as our knowledge, we do not observe any of these approaches that focus on how use third-party tools in efficient ways.

Another category consists of approaches that develop specific algorithms for feature models’ verification. Unfortunately, there exist few approaches in this category. One distinctive approach in this category is a simplified LTMS (Logic Truth Maintenance Systems) algorithm proposed by Batory [1], considering the characteristics of feature models’ verification. This algorithm aims to find those features that must be removed or selected in the customization to feature models, through constraints propagation. This algorithm focuses on how to detect customization-level deficiencies, and thus can not detect all kinds of model-level deficiencies. For example, considering the following two constraints: ¬*a*∨*b* and ¬*a*∨¬*b*, this algorithm cannot detect that *a* is a dead feature until a user tries to select *a*. Although a full version LTMS algorithm may detect such kind of model-level deficiencies, the direct invoking on a full version LTMS algorithm is just like invoking on a third-party tool, which still does not focus on the problem of how to invoke third-party tools in efficient ways, while this problem is particularly concentrated in this paper.

**6. Conclusions**

Dead feature detection in feature models is often performed by checking the deadness of each feature with the help of third party solvers such as SAT-Solvers. In this paper, we prove that the detection can be done by finding only a few nontrivial dead features. According to such facts, one can design algorithms to determine the deadness of all features by checking only some of them. Therefore the times of invoking third party solvers are reduced, as well as the total running time. We then present an abstract algorithm following such idea, and further provide fifteen concrete algorithms specializing the abstract algorithm in two dimensions: the policy of selecting a refinement path in a feature model, and the method of locating the nontrivial dead feature in the refinement path. We compare these fifteen algorithms by changing three variables in a set of experiments. The results show that in small feature models, both leaf-first binary search and bottom-up linear search algorithms give best results; however in large feature models, leaf-first binary search algorithms are preferred.

Our future work will mainly focus on applying our algorithms to real feature models with very large scales, such as the Linux feature model. In such a large feature model, checking features one by one takes too much time, and our algorithms may be a promising approach to detecting dead features.

**References**

1. D. Batory, “Feature Models, Grammars, and Propositional Formulas”, in Proceedings of 9th Software Product Line Conference, LNCS 3714, 2005, pp.7-20.
2. Daniel Le Berre, Anne Parrain. The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation, Volume 7 (2010), system description, pages 59-64.
3. Clements, P., and Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, 2002.
4. Czarnecki, K. and Kim, C.H.P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report. In OOPSLA’05 International Workshop on Software Factories, 2005.
5. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson, “Feature-Oriented Domain Analysis Feasibility Study”, Technical Reports, Software Engineering Institute, Carnegie Mellon University, 1990.
6. T. von der Maßen, H. Lichter, “Deficiencies in feature models”, in Workshop on Software Variability Management for Product Derivation, in Conjunction with the 3rd Software Product Line Conference, 2004.
7. M. Mannion, “Using First-Order Logic for Product Line Model Validation”, in Proceedings of 2nd Software Product Line Conference, LNCS 2379, 2002, pp.176-187.
8. Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, and Toro, M.: Automated error analysis for the agilization of feature modeling. Journal of Systems and Software, 81(6), 2008, pp.883-896.
9. Yan, H., Zhang, W., Zhao, H., and Mei, H.: An Optimization Strategy to Feature Models’ Verification by Eliminating Verification-Irrelevant Features and Constraints. In Proceeding of 11th International Conference on Software Reuse, pp.65-75, LNCS 5791, 2009.
10. Zhang, W., Zhao, H., and Mei, H.: A Propositional Logic-Based Method for Verification of Feature Models. In Proceedings of 6th International Conference on Formal Engineering Methods, pp.115-130, 2004.
11. Zhang, W., Mei, H., and Zhao, H.: Feature-Driven Requirements Dependency Analysis and High-Level Software Design. Requirements Engineering Journal, 2006, 11(3):205-220.

1. It should be pointed out that this definition is a simplified definition of feature models, and the purpose is to make this paper concentrate on the core aspects of feature models’ verification. The main simplification is that the *optionality* attribute of features is omitted, and every feature is treated as *optional*, except the root feature that is *mandatory* to every product. Such simplification will not change the essence of feature models’ verification, since that any feature model can be easily transformed into a unique feature model conforming to Definition 1, by using the *core-set* technique proposed in our previous research [9]. [↑](#footnote-ref-1)