Automated Analysis of Feature Models using Atomic Sets *

Sergio Segura
Department of Computer Languages and Systems
University of Seville
Av Reina Mercedes S/N, 41012 Seville, Spain
sergiosegura AT us.es

Abstract

Scalability is recognized as a key challenge in the automated analysis of Feature Models (FMs). Current solutions in this context mainly propose using different logic paradigms as a way to improve the performance at the solution level while the problem remains the same. Atomic Sets (ASs) were proposed as a promising solution for the simplification of FMs (i.e. reduction of the number of variables) in the context of automated analysis. However, years after their introduction, the lack of specific algorithms and performance results still hinder its integration into current proposals and tools. In this paper, we set the basis for the usage of ASs as a generic technique for the automated analysis of FMs. In particular, we first propose a specific algorithm to construct the ASs of an FM. Then, we present a performance test measuring the degree of improvement (in time and memory) when implementing ASs into CSP, BDD and SAT-based solutions.

1. Introduction

The automated analysis of FMs enables the extraction of information from the models through the implementation of a number of analysis operations. The problems of feature combinatorics related to these operations are accepted to be NP-hard and can take a long time to solve [2]. Current solutions in this context mainly focus on the usage of different logic paradigms and solvers as a way to improve the performance at the solution level [2, 5, 6]. However, few efforts have been made to study how improving the performance through the treatment of FMs in the domain of the problem.

In [10], Zhang *et al.* proposed a propositional logic-based method for the verification of FMs at different binding times. In their work, the authors proposed the usage of so-called *atomic sets* as a way to improve the efficiency of the process at the problem level. An atomic set represents a group of features (at least one) that can be treated as a unit during the analysis of an FM. According to Zhang *et al.* working with atomic sets instead of features may often reduce the size of the problem to solve improving efficiency. However, even when atomic sets seems a promising technique, the benefits of using it are still unknown by the research community and the lack of specific algorithms and empirical results difficults its integration into current proposals.

In this paper, we set the basis for the usage of atomic sets as a generic technique for the automated analysis of FMs. To this aim, we report the lessons learned from integrating atomic sets into our framework for the automated analysis of FMs, FAMA[1] [5, 9]. In particular, we first propose an algorithm to construct the atomic sets of a given FM. Then, we detail the results of a performance test measuring the degree of improvement (in time and memory) when implementing atomic sets into CSP, BDD and SAT-based solutions integrated in FAMA.

The remainder of the paper is structured as follows: in Section 2, the automated analysis of FMs and atomic sets are introduced. Our algorithm for the computation of the atomic set of an FM and the experimental results are presented in Section 3. Finally, we summary our conclusions in Section 4.

2. Preliminaries

2.1. Automated Analysis of Feature Models

The analysis of an FM consists on the observation of its properties. Typical operations of analysis allow finding

*This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and the Andalusian Government project ISABEL (TIC-2533)

[1] http://www.isa.us.es/fama/

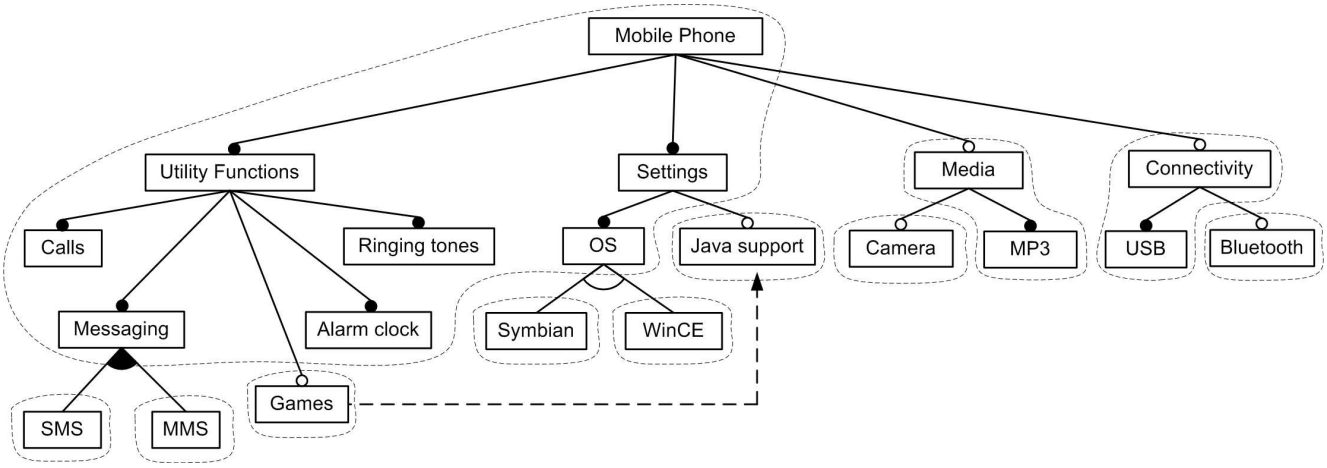


Figure 1. Atomic sets

out whether an FM is void (i.e. it represents no products), whether it contains errors (e.g. feature that can not be part of any products) or what is the number of product of the software product line represented by the model.

In order to enable the automated analysis of FMs specific logic paradigms and solvers have been proposed. In particular, the analysis is generally performed in two steps: i) First, the model is translated into an specific logic representation such as a Constraint Satisfaction Problem (CSP) [3], a SATisfiability problem (SAT) [1] or a Binary Decision Diagram (BDD) [8], ii) Then, off-the-shelf solvers are used to automatically perform a variety of operations on the logic representation of the model.

The available empirical results [5, 6] and surveys [4] suggests that there is neither an optimum logic paradigm nor solver to perform all the operations identified on FMs. As a result of this, current research efforts as the FAMA framework integrate different paradigms and solver in order to combine the best of all of them in terms of performance [7].

2.2. Atomic Sets

The usage of *atomic sets* for the automated analysis of FMs was proposed by Zhang *et al.* back in 2004 [10]. An atomic set represent a group of features (at least one) that can be treated as a unit during the analysis of an FM. The intuitive idea of atomic sets is that mandatory features and their parent features can be treated as a whole in certain analysis operation without altering the result. This is because those features can never appear in a product separately. Figure 1 depicts an FM inspired by the mobile phone industry. The areas delimited by dashed lines illustrate the atomic sets. As an example, feature *'USB'* can only be part of a product if its parent feature, *'Connectivity'*, is also part of the product. Thus, both features can be treated as a unit to perform some operation such as finding out if the model is void (i.e. it represents at least one product) or what is the number of product represented by the model. The step to use atomic sets as the basic unit in the analysis can be achieved by replacing each feature in the model by the atomic set which contains it.

The benefit of using atomic sets lies in efficiency. Working with atomic sets instead of features may reduce the number of variables to consider and consequently the size of the problem to solve. As an example, in the FM of Figure 1 the number of variables after constructing atomic sets is reduced from 20 (features) to 11 (atomic sets).

3. Our Contribution

3.1. Atomic Sets Computation

In this section, we present an algorithm for the construction of the atomic sets of a given FM. Figure 2 depicts the pseudocode of the function implementing our algorithm. The function receives an FM as input and returns a set of atomic sets as output. Each atomic set is composed by a name (e.g. "AS-2") and the collection of features in the atomic set. The main part of the work is made by a recursive procedure also presented as part of the figure. This procedure checks all the subfeatures of the feature received as input. For each subfeature, if it is mandatory, it is added to the current atomic set under construction. Otherwise, a new atomic set including the subfeature is created and added to the collection of atomic sets. After repeating this process with all the features in the model, the set of atomic set is returned.

```
1  function buildAS(FeatureModel fm)::Collection<AtomicSet>
2    Collection<AtomicSet> atomic_sets = new Set<AtomicSet>();
3    Feature root = fm.getRoot();
4    AtomicSet as = new AtomicSet("AS-0");
5    as.addFeature(root);
6    atomic_sets.add(as);
7    computeAS(atomic_sets, root, as, 0);
8    return atomic_sets;
9  endfunction

10 procedure computeAS(Collection<AtomicSet> atomic_sets, Feature f, AtomicSet current_set, int set )
11   foreach Feature g in f.getSubfeatures()
12     if (g.getRelationType() == Feature.MANDATORY)
13       current_set.addFeature(g);
14       computeAS(atomic_sets,g,current_set, set);
15     else
16       String setname = "AS-" + (set+1);
17       AtomicSet new_as = new AtomicSet(setname);
18       new_as.addFeature(g);
19       atomic_sets.add(new_as);
20       computeAS(atomic_sets,g,new_as,set+1);
21     endif
22   endforeach
23 endprocedure
```

Figure 2. Algorithm for the computation of atomic sets

3.2. Experimental Results

In order to measure the benefits of implementing atomic sets we carried out a performance comparison using the FAMA framework. In particular, we first generated a set of random FMs with a different number of features and cross-tree constraints. Then, we performed some operations using CSP, BDD and SAT-based solvers with and without the usage of atomic sets. Next, we gave the details about the experiment and the results.

3.2.1 The Experiments

For the experiments, we generated a number of random FMs using FAMA. The algorithm for the automated generation of those models is presented in Appendix A as one of the contributions of the paper. In particular, we used four groups of 50 randomly generated FMs. Each group included FMs with a number of features in a specific range ([50-100),[100-150),[150-200) and [200-300)) in order to test the performance with different sizes of the problem. Once all the FMs were generated, we proceeded with the execution using the FAMA framework. For the execution, we used three of the solvers integrated in FAMA: JaCoP[2] (CSP), JavaBDD[3] (BDD) and Sat4j[4] (SAT). The set of experiments was executed twice with each solver, with and without the usage of atomic sets, in order to compare the improvement in the performance. Each FM was executed several times increasing the number of cross–tree constraints from 0 to 5, 10, 15, 20 and 25% of the number of the features in the FM. Cross-tree constraints were added randomly as well, but checking that the same feature can not appear in more than one cross–tree constraint and that a feature can not have a cross–tree constraint with any of its ancestors. Once the results were obtained, we worked out averages from the results in order to avoid as much exogenous interferences as possible. Averages were obtained from all the FMs in each range with the same percentage of cross–tree constraints. Table 1 summarizes the characteristics of the experiments.

N. of Features	N. of instances	CT constraints
[50-100)	50	[0%-25%]
[100-150)	50	[0%-25%]
[150-200)	50	[0%-25%]
[200-300]	50	[0%-25%]

Table 1. Experiments

For our experiments we performed two operations: $i)$ finding out if an FM is valid, that is, if it represents at last one product and $ii)$ finding out the total number of products represented by a given FM. The first one is one of simplest

[2] http://jacop.cs.lth.se/
[3] http://javabdd.sourceforge.net/
[4] http://www.sat4j.org/

operation while the second is the hardest one in terms of performance because it is necessary to work out the total number of possible combinations. The data extracted from the tests were:

- Average memory used by the logic representation of the FM (measured in Kilobytes).
- Average execution time to find one product (measured in milliseconds).
- Average execution time to obtain the number of products (measured in milliseconds).
- Time to compute the atomic sets (measured in milliseconds).

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java 1.6.0_04. We ran our tests on a WINDOWS VISTA BUSINESS EDITION machine equipped with a 2.4Ghz Intel Core 2 microprocessor and 2048 MB of RAM memory.

3.2.2 The Results

The experimental results revealed a noticeable improvement in the performance of solvers when using atomic sets. The first evidence was a reduction in the average memory usage of 4% in JaCoP, 15% in Sat4j and 19% in JavaBDD. The operation to find one product was performed on average between 10% (JaCoP and Sat4j) and 20% (JavaBDD) faster using atomic sets. Similarly, the number of products of the FMs was computed on average between 5% and 20% faster using this technique.

The improvement in the performance was better observable with large FMs. As an example, Figure 3 illustrates the average memory and time used by JavaBDD in the range of 200-300 features. Improvements were especially considerable in the experiments with a higher number of cross-tree constraints. In these cases, experiments using atomic sets revealed an improvement of up to 25 seconds (28% of improvement) on average when finding the number of solutions.

The benefits of using atomic sets were especially noticeable when we studied the hardest cases in terms of performance. An example of this situation is illustrated in Figure 4. This figure presents the time and memory consumed by JavaBDD in the worst cases of the range of experiments between 200 and 300 features. In these cases, we found an improvement of up to 119 MB (in a total of 476) in memory and 7 minutes (in a total of 27) in the time to find the number of solutions.

Finally, we found that the time to compute the atomic sets of FMs was insignificant (0-10 ms) in all the cases.

3.2.3 Discussion

The improvement in time when adopting atomic sets was not easy to measure in part of the experiments. This was because many of these were not complex enough and the time to perform the operations was very low. This is the reason because we provided range of improvements for the time and we do not give accurate data.

The available empirical results shows that the memory usage of BDD solvers can be huge [5]. In fact, it seems to increases exponentially with the number of cross-tree constraints. In this context, our experimental results suggest that atomic sets could be used as a suitable strategy to ease the effects of this trend.

Our performance test showed that the usage of atomic-set may provide a great improvement of the performance specially when dealing with large FMs. However, we remark that the cost of implementing atomic sets is practically insignificant and can bring very positive results even in small specific cases (especially when these are hard to compute).

Finally, we remark that in this paper we focus on the performance provided by different solvers when implementing atomic sets. For an empirical comparison of the performance provided by the solvers presented in this paper we refer the reader to [5].

4. Conclusions

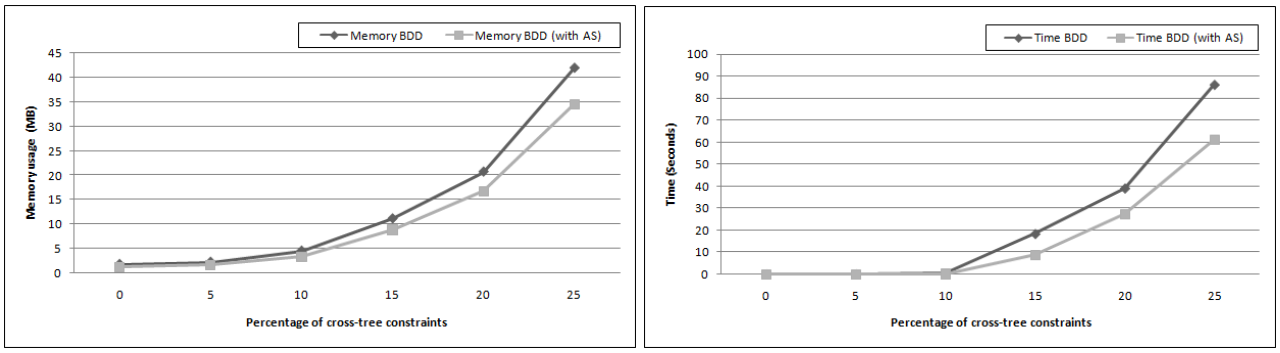
In this paper, we set the basis for the usage of atomic sets as a generic technique for the automated analysis of FMs. In particular, we first presented an algorithm to construct the atomic set of a given FMs. Then, we detailed the results of a performance test measuring the degree of improvement when using atomic sets with CSP, BDD, and SAT-based solutions currently integrated in the FAMA framework.

The experimental results revealed that the improvement when using atomic sets can be considerable in both, time and memory. This improvement was especially important in large FMs (in the order of hundreds of MB and minutes). However, we remark that the payoff for implementing this technique is insignificant and can bring noticeable results even in small cases, especially if these are hard to compute.

As a result of our performance test, we conclude that the techniques for the treatment of FMs at the problem level could help to improve the efficiency notably. Additionally, we consider that this kind of techniques could be applicable to the analysis of other kind of variability models.

References

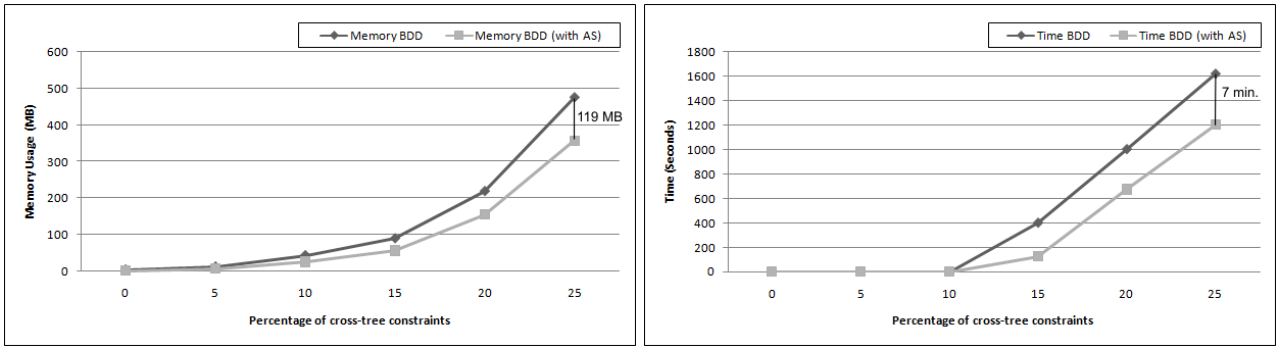
[1] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Confer-*



(a) Average memory usage

(b) Average time to find the number of products

Figure 3. Average time and memory usage of JavaBDD (range 200-300 features)



(a) Memory usage

(b) Time to find the number of products

Figure 4. Time and memory usage of JavaBDD in the worst cases (range 200-300 features)

ence, LNCS 3714, pages 7–20, 2005.

[2] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December, 2006.

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[4] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[5] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *LNCS*, 4143:389–398, 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and 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)*, pages 129–134, 2007.

[8] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*, 2005.

[9] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez. Fama framework. In *Proceedings of the 12th International Software Product Line Conference (Tool demonstration)*, 2008.

[10] W. Zhang, H. Zhao, and H. Mei. A propositional logic-based method for verification of feature models. In J. Davies, editor, *ICFEM 2004*, volume 3308, pages 115–130. Springer–Verlag, 2004.

A Random Feature Model Generation

The random generation of feature models is one of the capabilities of the FAMA framework. Figure 5 depicts the pseudocode of the algorithm used for the generation of those models. In particular, the algorithm takes the following parameters as inputs:

w : is the maximum number of child relationships of the features in the model.

h : is the maximum height of the model. We consider the height of a feature model as the maximum distance between the root and any feature without considering cross–tree constraints.

e : is the maximum number of elements in a set relationship.

d : is the number of cross–tree constraints to be generated.

```
1  function GenerateFM(int w,h,e,d)::FeatureModel
2     FeatureModel fm = new FeatureModel();
3     Feature root = new Feature();
4     fm.setRoot(root);
5     fm.generateTree(w,h,e,root);
6     fm.generateCrossTreeConstraints(d);
7     return fm;
8  endfunction

9  procedure GenerateTree(int w,h,e, Feature parent)
10    if (h ≥ 1)
11       int nChildren = Random.getInt(w);
12       forach int i in {0..nChildren}
13          RelationType type = Random.getRelationType();
14          Feature child = new Feature();
15          switch(type)
16             case mandatory:
17                createMandatory(parent,child);
18                generateTree(w,h-1,e,child);
19             case optinal:
20                createOptional(parent,child);
21                generateTree(w,h-1,e,child);
22             case cardinality:
23                int card = Random.getInt();
24                createCardinality(parent,child,1,card);
25                generateTree(w,h-1,e,child);
26             case set:
27                int nChildrenSet = Random.getInt(e);
28                int setCard = Random.getInt(nChildrenSet);
29                FeatureGroup group = new FeatureGroup(nChildrenSet);
30                createSet(parent,group,1,setCard);
31             foreach Feature node in group
32                GenerateTree(w,h-1,e,node)
33             endforeach
34          endswitch
35       endforeach
36    endif
37 endprocedure

38 procedure generateCrossTreeConstraints(int d)
39    if (h ≥ 1)
40       int i = 0;
41       while (i < d)
42          Feature f = getFeatureRandomly();
43          Feature g = getFeatureRandomly();
44          if (valid(f,g))
45             if (Random.getBool == true)
46                generateDepends(f,g);
47             else
48                generateExcludes(f,g);
49             endif
50             i++;
51          endif
52       endwhile
53    endif
54 endprocedure
```

Figure 5. Algorithm for the generation of random feature models