View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by idUS. Depósito de Investigación Universidad de Sevilla



# A first step towards a framework for the automated analysis of feature models

David Benavides, Sergio Segura, Pablo Trinidad, Antonio Ruiz-Cortés
Dpto. of Computer Languages and Systems
University of Seville
{*benavides, sergio, trinidad, aruiz*}@*tdg.lsi.us.es*

***Abstract*— Feature modelling is a common mechanism for variability management in the context of software product lines. After years of progress, the number of proposals to automatically analyse feature models is still modest and the data about the performance of the different solvers and logic representations used in such area are practically non–existent. Three of the most promising proposals for the automated analysis of feature models are based on the mapping of feature models into CSP, SAT and BDD solvers. In this paper we present a performance test between three off-the-shelf Java CSP, SAT and BDD solvers to analyse feature models which is a novel contribution. In addition, we conclude that the integration of such proposals in a framework will be a key challenge in the future.**

***Index Terms*— Software Product Lines, Variability Management, Feature Models.**

## I. Introduction

Feature Models (FMs) are one of the most common variability mechanisms. Good tool support is needed to debug, extract information and in summary analyze FMs in order to select them as a variability mechanism in a Software Product Line (SPL) approach. A FM represents all possible products of a SPL in a single model using features. FMs can be used in different stages of development such as requirements engineering [10], [11] , architecture definition or code generation [1], [3]. A FM is a tree–like structure and consists of: $i$)relations between a parent feature and its child features. $ii$) cross–tree constraints that are typically inclusion or exclusion statements of the form "if feature F is included, then feature X must also be included (or excluded)".

Automated analysis of FMs is an important challenge in SPL research [1], [2]. It can be performed using off–the–shelf solvers to automatically extract useful information of the SPL such as the number of possible combinations of features, all the configurations following a criteria, finding the minimum cost configuration, etc. Although there have been some promising proposals based in the representation of FMs as a Constraint Satisfaction Problem (CSP), boolean SATisfiability problem (SAT) and Binary Decision Diagrams (BDD) the performance of the solvers working with such representations is unknown for the SPL community.

In a previous work, we presented a performance comparison of two CSP java solvers analysing FMs [8]. In this paper we go further integrating different solvers and logic representations. First we give a complete mapping for the three solvers (BDD, SAT and CSP) and then we present a performance comparison of them. To the best of our knowledge, this is the first test that measures the performance of solvers dealing with different logic representations of FMs.

The remainder of the paper is structured as follows: in Section II the automated analysis of FMs is outlined and details on how to translate a FM into a CSP, BDD and SAT are presented. Section III focuses on the results of the experiment. Finally we summarize our conclusions and describe our future work in Section IV.

## II. Automated analysis of Feature Models

Once a FM is translated into a suitable representation it is possible to use off–the–shelf solvers to automatically perform a great variety of operations such as calculating the number of possible combinations of features, retrieving configurations following a criteria, finding the minimum cost configuration, etc [6].

There is a great variety of techniques and tools that can be used in the automated analysis of FMs. This paper focus on three well known problems in the area of automated reasoning: Constraint Satisfaction Problems (CSP), Boolean Satisfiability Problems (SAT) and Binary Decision Diagrams (BDD). All those representations have not been yet fully adopted in the automated analysis of FMs. In the next sections we will give a brief overview of each of them and finally we will introduce how translating a FM into a CSP, SAT and BDD.

### A. Constraint Satisfaction Problem

Constraint Programming can be defined as the set of techniques such as algorithms or heuristics that deal with CSPs. A CSP consists on a set of variables, finite domains for those variables and a set of constraints restricting the values of the variables. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. CSP solvers can deal with numerical values such as integer domains. The main ideas concerning the use of constraint programming on FM analysis were stated in [6], [7].

### B. Boolean Satisfiability Problem (SAT)

A propositional formula is an expression consisting on a set of boolean variables (literals) connected by logic operators ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$). The propositional satisfiability problem (SAT) consists on deciding whether a given propositional

formula is satisfiable, that is, if logical values can be assigned to its variables in a way that makes the formula true.

The problem is restricted by using the propositional formulas in conjunctive normal form (CNF), that is, propositional formulas composed by a conjunction of clauses in which each clause is a disjunction of literals $(e.g.((L1 \vee L2) \wedge (L3 \vee L4) \wedge (L5 \vee L6)))$. Every propositional formula can be converted into an equivalent formula in CNF by using logical equivalences. The basic concepts about the using of SAT in the automated analysis of FMs were introduced in [1].

### C. Binary Decision Diagrams (BDD)

A Binary Decision Diagram (BDD) is a data structure used to represent a boolean function. A BDD is a rooted, directed, acyclic graph composed by a group of decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each node in the graph represents a variable in a boolean function and has two child nodes representing an assignment of the variable to 0 and 1. All paths from the root to the 1-terminal represents the variable assignments for which the represented boolean function is true meanwhile all paths to the 0-terminal represents the variable assignments for which the represented boolean function is false.

Although the size of BDDs can be reduced according to some established rules, the weakness of this kind of representation is the size of the data structure which may vary between a linear to an exponential range depending upon the ordering of the variables. Calculating the best variable ordering is an NP-hard problem. In the context of the automated analysis of FMs there are some tools that claim the internal use of BDDs [9]

### D. Mapping

Rules for translating FMs to constraints are listed in Figure 1. In all cases the notation more common in the bibliography has been used. The final representation of the FM is the conjunction of the translated relations following the rules of Figure 1 plus an additional constraint selecting the root which is included in all products.

## III. Experimental Results

The experiments focused on a performance comparison of three off–the–shelf Java solvers working with CSP, SAT and BDD in order to test how these representations can influence in the automatic analysis of FMs. The comparison results were obtained from the execution of a number of FMs mapped as CSP, BDD and SAT in three of the most popular Java solvers within the research community: JaCoP[1] (CSP), JavaBDD[2] (BDD) and Sat4j[3] (SAT).

[1]http://www.cs.lth.se/home/Radoslaw_zymanek/
[2]http://javabdd.sourceforge.net
[3]http://www.sat4j.org

| N. of Features | N. of instances | Dependencies |
|---|---|---|
| **[50-100)** | 50 | [0%-25%] |
| **[100-150)** | 50 | [0%-25%] |
| **[150-200)** | 50 | [0%-25%] |
| **[200-300]** | 50 | [0%-25%] |

TABLE I
Experiments

### A. The Experiment

we used four group of 50 randomly generated FMs. Each group included FMs with a number of features in an specific range ([50-100),[100-150),[150-200) and [200-300)) with a double aim: test the performance of small, medium and large instances and working out averages from the results in order to avoid as much exogenous interferences as possible. After formulating each one as a CSP, BDD and SAT, we proceeded with the execution. Each FM was executed several times increasing the number of cross–tree constraints from one until the 25% of the number of the features in the FM in order to find out how dependencies influence in the performance. The dependencies were added randomly as well, but cheking 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. Averages were obtained from all the FMs in each range with the same percentage of cross–tree constraints. Table I summarizes the characteristics of the experiments.

As exposed in [6], [7], there are some operations that can be performed. For our experiments we performed two operations: $i)$ finding out if a model is satisfiable, that is, if it has at last one solution and $ii)$ finding the total number of configurations of a given FM. The first one is the simplest 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 solution (measured in milliseconds).
- Total number of solutions, that is, the potential number of products represented in the FM.
- Average execution time to obtain the number of solutions (measured in milliseconds).

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java 1.5.0_04. We ran our tests on a WINDOWS XP PROFESSIONAL SP2 machine equipped with a 3Ghz Intel Pentium IV microprocessor and 512 MB of DDR RAM memory.

### B. The Results

The experimental comparison revealed some interesting results. The first evidence was that JavaBDD is on average 96% faster than JaCoP and 75% faster than Sat4j finding one solution. However, JavaBDD revealed a memory usage on average 928% higher than JaCoP and 1672% higher than Sat4j. On the other hand, although JaCoP and Sat4j showed

| RELATION | | CSP | BDD | SAT |
|---|---|---|---|---|
| MANDATORY | P – C | $P=C$ | $P \leftrightarrow C$ | $(\neg P \vee C) \wedge (\neg C \vee P)$ |
| OPTIONAL | P – C | $if\,(P=0)$<br>$C=0$ | $C \rightarrow P$ | $\neg C \vee P$ |
| OR | P – $C_1$ $C_2$ $C_n$ | $if\,(P>0)$<br>$sum(C_1, C_2, ..., C_n)\, in \{1..n\}$<br>$else$<br>$C_1=0, C_2=0, ..., C_n=0$ | $P \leftrightarrow (C_1 \vee C_2 \vee ... \vee C_n)$ | $(\neg P \vee C_1 \vee C_2 \vee ... \vee C_n) \wedge (\neg C_1 \vee P) \wedge$<br>$(\neg C_2 \vee P) \wedge ... \wedge (\neg C_n \vee P)$ |
| ALTERNATIVE | P – $C_1$ $C_2$ $C_n$ | $if\,(P>0)$<br>$sum(C_1, C_2, ..., C_n)\, in \{1..1\}$<br>$else$<br>$C_1=0, C_2=0, ..., C_n=0$ | $(C_1 \leftrightarrow (\neg C_2 \wedge ... \wedge \neg C_n \wedge P)) \wedge$<br>$(C_2 \leftrightarrow (\neg C_1 \wedge ... \wedge \neg C_n \wedge P)) \wedge$<br>$(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{n-1} \wedge P))$ | $(C_1 \vee C_2 \vee ... \vee C_n \vee \neg P) \wedge$<br>$(\neg C_1 \vee \neg C_2) \wedge ... \wedge (\neg C_1 \vee \neg C_n) \wedge (\neg C_1 \vee P) \wedge$<br>$(\neg C_2 \vee \neg C_3) \wedge ... \wedge (\neg C_2 \vee \neg C_n) \wedge (\neg C_2 \vee P) \wedge$<br>$(\neg C_{n-1} \vee \neg C_n) \wedge (\neg C_{n-1} \vee P) \wedge (\neg C_n \vee P)$ |
| IMPLIES | A → B | $if\,(A>0)$<br>$B>0$ | $A \rightarrow B$ | $\neg A \vee B$ |
| EXCLUDES | A ↔ B | $if\,(A>0)$<br>$B=0$ | $\neg(A \wedge B)$ | $\neg A \vee \neg B$ |

Fig. 1. Mapping

a similar memory usage, SAT representation showed better results in both aspects, memory and especially in time. The performance of the solvers was similar in the four groups of experiments. Figures 2 and 3 presents the results for the group of FMs with a number of features between 100 and 150.

In fact, Figure 2 can be confusing in the sense that in the worst case the memory usage is insignificant (in the order of 2 Mb) but this behavior seems to be exponential with the number of features and dependencies. For instance, in the range of (200-300) features, we found some cases where the memory used by the solver was around 300 Mb. We think that in bigger FMs (e.g. 1000 features) this can be even a bigger problem. What Figure 2 try to stress is the difference in the use of memory of the three solvers.

The results obtained from finding the total number of configurations of a given FM showed a great superiority of JavaBDD. While JaCoP and Sat4j were computationally incapable of performing that operation in a reasonable time in most of the cases, JavaBDD lasted 5312 ms to work out the $7.77 \times 10^{34}$ solutions of the worst case.

Finally, we found some unexpected results or outliers in the data obtained from the experiments with JaCoP and JavaBDD. On the one hand, JaCoP showed in a few consecutive executions a huge number of backtracks and consequently a great time penalty. On the other hand, JavaBDD revealed in a few experiments a huge memory usage which seemed to increase exponentially with the number of dependencies. We are investigating the possible causes of these behaviors [5].

### C. Discussion

The great superiority of JavaBDD on finding the total number of solutions is because for calculating the number of solutions, in general, CSP and SAT solvers have to retrieve all the solutions (which is a #P-complete problem [12]) meanwhile BDD solvers use efficient graph algorithms to calculate the total number of solutions without the need of calculating all the solutions. The huge memory usage of BDD solvers depends on the variable ordering for representing the BDD.

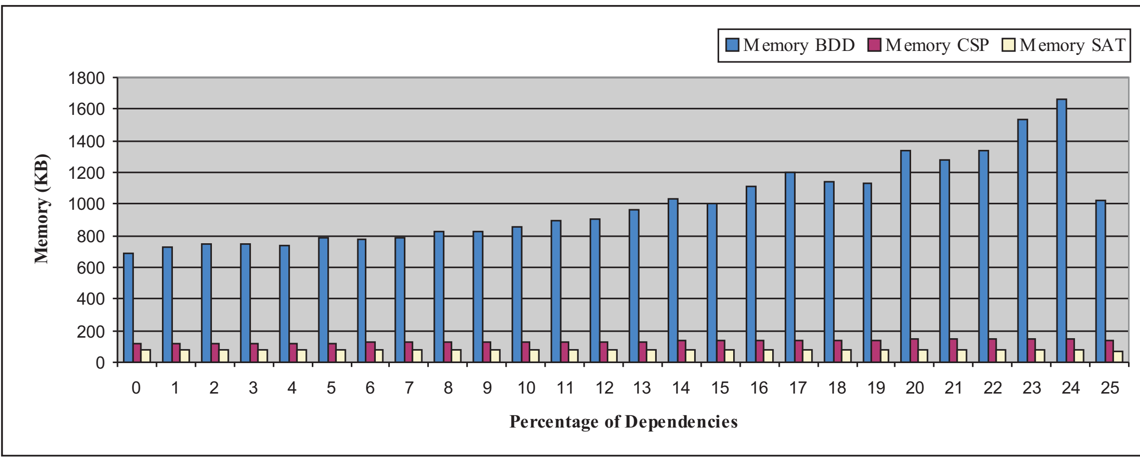


Fig. 2. Memory Usage

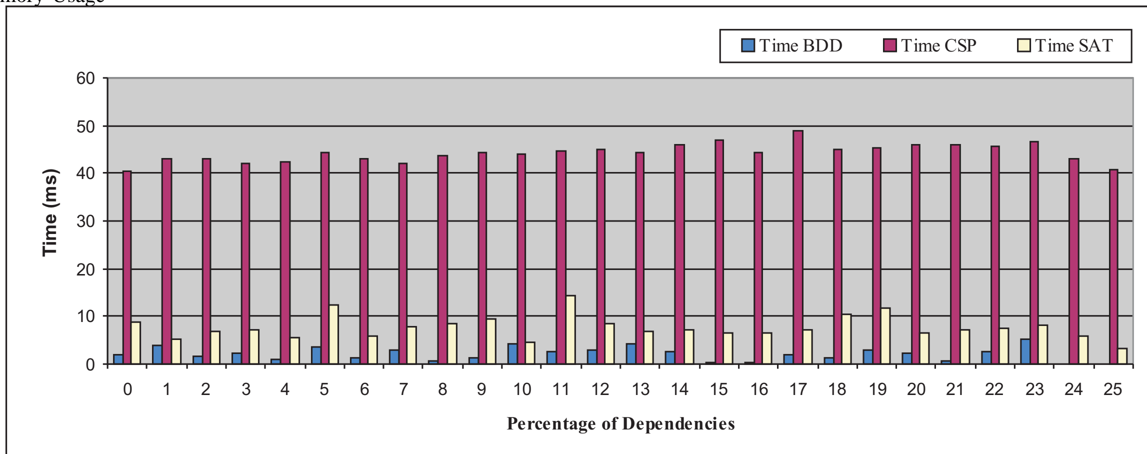


Fig. 3. Average time to get one solution

As stated earlier, the size of BDDs can be reduced with a good variable ordering, however, calculating the best variable ordering is a NP-hard problem.

To the best of our knowledge, there is only a proposal to include feature attributes in the automated analysis of feature models [6]. This proposal uses CSP solvers for that aim and we are not aware of any result where BDD or SAT solvers could be used to deal with feature attributes in order to maximize or minimize values.

As a result of the test, we claim that there is not an optimum representation for all the possible operations that can be performed on FMs. Therefore, we think that a framework for the automated analysis of feature models is needed. The framework will be designed to be open to other solvers where formal semantics of feature models will play a fundamental role [13]. The development of such a framework is the seed of our ongoing research [4].

## IV. Conclusion and Future Work

In this paper we integrated the use of different solvers in the automated analysis of FMs. We presented how to translate a feature model into a CSP, SAT and BDD and we performed a comparative test between three off–the–shelf Java solvers managing with each representation. The results showed that using BDDs for determining satisfiability in a FM is much faster than using SAT or CSP. On the other hand, FMs mapped as BDDs required a bigger memory usage in comparison with CSP and SAT. The test also revealed that while FMs mapped as SAT and CSP are computationally incapable of finding the total number of configurations in most of medium and large size FMs, FMs mapped as BDDs can get it in a very low time.

We think that there is not an optimum representation for all the possible operations that can be performed on FMs, therefore a framework for the automated analysiss of feature models is needed.

## Acknowledgements

We thank Don Batory and Jean-Christophe Trigaux for their helpful comments on an earlier draft of this paper.

## References

[1] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, 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*, Conditionally accepted, 2006.
[3] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
[4] D. Benavides, A. Durán, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A framework for the automated manipulation of software product lines. *In preparation*.

[5] D. Benavides, A. Ruiz-Cortés, B. Smith, Barry O'Sullivan, and P. Trinidad. Computational issues on the automated analyses of feature models using constraint programming. *International Journal of Software Engineering and Knowledge Engineering*, in preparation, 2006.

[6] 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.

[7] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005*, 2005.

[8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. *LNCS*, to be asigned:to be asigned, 2006.

[9] 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.

[10] S. Jarzabek, Wai Chun Ong, and Hongyu Zhang. Handling variant requirements in domain modeling. *The Journal of Systems and Software*, 68(3):171–182, 2003.

[11] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, 2002. Springer.

[12] G. Pesant. Counting solutions of csps: A structural approach. In *IJCAI*, pages 260–265, 2005.

[13] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006.