

Using Java CSP Solvers in the Automated Analyses of Feature Models ^{*}

David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés

Dpto. de Lenguajes y Sistemas Informáticos
University of Seville
Av. de la Reina Mercedes S/N, 41012 Seville, Spain
{benavides, sergio, trinidad, aruiz}@tdg.lsi.us.es

Abstract Feature Models are used in different stages of software development and are recognized to be an important asset in model transformation techniques and software product line development. The automated analysis of feature models is being recognized as one of the key challenges for automated software development in the context of Software Product Lines. In our previous work we explained how a feature model can be transformed into a constraint satisfaction problem. However cardinalities were not considered. In this paper we present how a cardinality-based feature model can be also translated into a constraint satisfaction problem. In that connection, it is possible to use off-the-shelf tools to automatically accomplish several tasks such as calculating the number of possible feature configurations and detecting possible conflicts. In addition, we present a performance test between two off-the-shelf Java constraint solvers. To the best of our knowledge, this is the first time a performance test is presented using solvers for feature modelling proposes

1 Introduction

Throughout the years, software reuse and quality have been two constants aims in software development. Although significant progress has been made in programming languages, methodologies and so forth, the problem seems to remain. Software Product Line (SPL) development [8] is an approach to develop software systems in a systematic way that intends to solve these problems. Roughly speaking, an SPL can be defined as a set of software products that share a common set of features. Therefore, an SPL approach could be useful for organizations that are product-oriented rather than project-oriented [7]. That is, organizations that operate in a particular market segment.

SPL engineering consists of two main activities: domain engineering (also called core asset development) and application engineering (also called product development). These two activities are complementary and provide feedback to each other. Domain engineering deals with core assets production, that is, the pieces of the products to be shared by all SPL products. On the other hand, application engineering deals with individual system production.

^{*} This work was partially supported by the Spanish Science and Education Ministry (MEC) under contracts TIC2003-02737-C02-01 (AgilWeb)

Feature Analysis [17] is an important task of domain engineering and is expected to produce a Feature Model (FM) as its main output. A FM can be defined as a compact representation of all possible products of an SPL. Furthermore, it is commonly accepted that FMs can be used in different stages of an SPL effort in order to produce other assets such as requirements documents [15,16], portlets-based applications [11,12] or even pieces of code [3,9,20]. Hence, FM becomes an important focus of research in the field of model transformation.

Automated analyses of FMs are an important challenge in SPL [1,2]. In a previous work [4,5] we presented how to transform a FM (without considering cardinalities) into a Constraint Satisfaction Problem (CSP). In that way, it is possible to use off-the-shelf constraint satisfaction solvers to automatically accomplish several tasks such as calculating the number of possible configurations and detecting possible conflicts. The contribution of this paper is twofold: *i*) to explain how a FM with cardinalities can be translated into a CSP and *ii*) to show the result of a performance test between two off the shelf Java constraint solvers: JaCoP and Choco. To the best of our knowledge, this is the first test that measures the performance of constraint solvers in the context of feature analyses.

The remainder of the paper is structured as follows: in Section 2 we introduce feature models. In Section 3 constraint programming is outlined and details on how to translate a FM into a CSP are presented. Section 4 focuses on the results of the experiment. Finally we summarize our conclusions and describe our future work in Section 5.

2 Feature Models

A Feature Model (FM) is a compact representation of all possible products of an SPL. FMs are used to model a set of software systems in terms of features and relations among them. Designing a software system in terms of features is more natural than doing it in terms of objects or classes. Consequently, a software system will be composed of a set of features.

Since FMs were first presented in 1990 [17] there have been many publications and proposals to extend, improve and modify the original FM diagram. However, despite years of research, there is no consensus on a FM notation. Although it would be desirable to have a common notation, it is out of the scope of this paper to give yet another FM notation. Therefore, we use the one proposed by Czarnecki [10] that was formalized as a context free grammar and integrates some previous extensions.

A FM is basically a tree structure with dependencies between features. Figure 1 represents the general metamodel of a FM (this metamodel was presented in [6]). Likewise, Figure 2 represents a FM of the James Project [13]. James is a collaborative web based system that we modeled in terms of features and can be a clear example of an SPL. Some products can be derived from the FM on Figure 2. Having a web service interface (WSInterface) is optional while it is mandatory to have user management (UserManagement), at least one module (Modules) and the core of the system (Core).

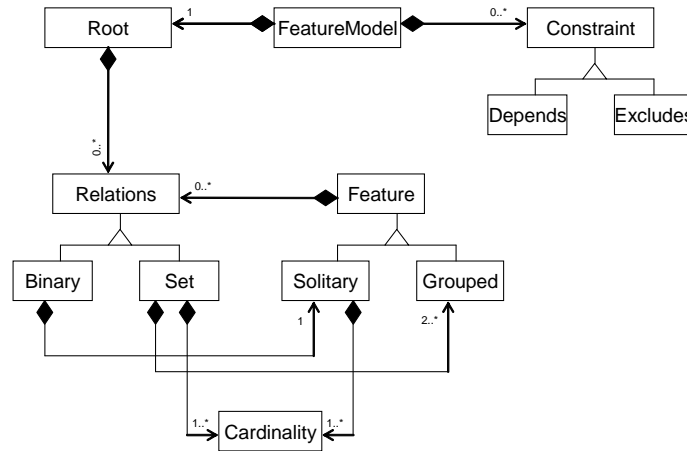


Figure 1. CFM meta model

A FM is composed of a **root** (*JAMES* in Figure 2) and an optional set of **constraints** (they refer to global constraints: depends and excludes; *R9* and *R10* in Figure 2).

A root is composed of an optional set of **relations**. Relations can be of two different types: **binary relations** which include mandatory (e.g. *R1*), optional (e.g. *R2*) and cardinality-based relations (e.g. *R4*) or **set** relations (e.g. *R7*).

A **feature** can be of two different types and is composed of zero or more relations. A binary relation is composed of one and only one **solitary feature** which is the child feature since the parent feature is the one that has this relation (*Core* or *UserManagement* are examples of solitary features); A set relation is composed of at least two **grouped** features (*Calendar, DB* or *PDA* are examples of grouped features). In addition, a solitary feature and set relations comprise one or more cardinalities. Note that in the graphical representation it is possible not to represent a cardinality in set relations although in fact that means that the cardinality is $\langle 1-1 \rangle$. Likewise, there are graphical representations for commonly used cardinalities of solitary features like $[1..1]$ and $[0..1]$ (see Figure 2 notes).

3 Constraint Programming

Constraint programming is a well established field of research and has been successfully applied in many engineering areas such as electronics or operational engineering. In the words of Prof. Freuder "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [14].

Constraint Programming can be defined as the set of techniques such as algorithms or heuristics that deal with Constraint Satisfaction Problems (CSP) to such an extent

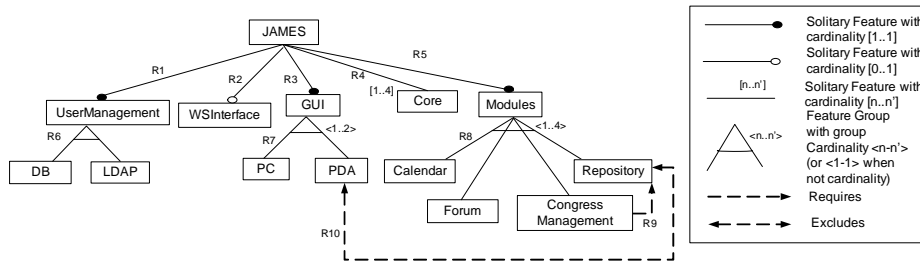


Figure 2. James System

that to solve a given problem by means of constraint programming, first the problem has to be formulated as a CSP.

A CSP consist of a set of variables, domains for those variables and a set of constraints restricting the values of the variables.

Definition 1 (CSP). A CSP is a three-tuple of the form (V, D, C) where $V \neq \emptyset$ is a finite set of variables, $D \neq \emptyset$ is a finite set of domains (one for each variable) and C is a constraint defined on V .

Once the problem is stated as a CSP, it is possible to use off-the-shelf CSP solvers that are able to provide the solutions to the problem. Internally the solvers will be implemented by using algorithms and heuristics that have been and are being investigated during several decades.

3.1 Mapping a FM into a CSP

We presented in [4,5] how a FM with dependencies was translated into a CSP. However we did not provide a way to do the same with cardinality-based FMs [10]. In this Section we give details on how to transform a FM with cardinalities into a CSP which is a novel contribution.

Rules for translating FMs to constraints are listed in Figure 3. First, there is a variable for each feature in the CSP. The domain of each variable depends on the cardinality associated to each variable. By default the domain is $\{0,1\}$ and if a feature is part of a cardinality relation, then the domain of the variable is added (e.g. $Core \in \{0,4\}$ in Figure 2). Then, a constraint selecting the root feature is added because all products have the root feature (e.g. $root = 1$). The final CSP for a FM is the conjunction of the constraints following the rules of Figure 3.

4 Experimental Results

Using CSP solvers, it is possible to automatically perform some operations on a FM such as calculating the number of possible combinations of features, retrieving configurations following a criteria, calculating the number of features in a given configuration,

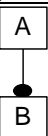
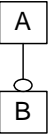
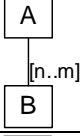
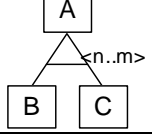
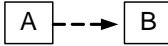
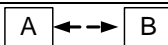
Relation	Diagram notation	Constraint
Mandatory		$B = A$
Optional		$\text{ifThen}(A=0;B=0)$
Cardinality		$\text{ifThenElse}(A=0;B=0;B \text{ in } \{n,m\})$
Set		$\text{ifThenElse}(A>0;\text{sum}(B,C) \text{ in } \{n,m\};B=0,C=0)$
Depends		$\text{ifThen}(A>0,B>0)$
Excludes		$\text{ifThen}(A>0,B=0)$

Figure 3. Feature Models and Related Constraints

validating a given FM to detect possible inconsistencies, finding an optimum product on the basis of a given criteria (the cheapest, the one with fewest features and so forth) or calculating the commonality factor for a given feature and the variability factor of a given FM.

The main ideas concerning the use of constraint programming on FM analyses were stated in [4,5] but some experimental results were left for our future work. In this Section we present an experimental comparison of two Java CSP solvers that were used to automatically analyse FMs.

4.1 The JaCoP and Choco Solvers

There are several commercial tools to work with CSPs. One of the major commercial vendors is ILOG that has two versions of CSP Solvers in C++ and Java. Because it is a commercial solution, we declined to use ILOG solvers' licenses in our empirical comparison.

To the best of our knowledge there is only one reliable and stable open source Java CSP Solver : Choco Constraint System [19]. We selected this solver because it seems to be one of the most popular within the research community and because it is the only

one we know of that is available for free directly from the Internet. We selected JaCoP solver [18] because it offers a free license for academic purposes. Both solvers have similar characteristic in terms of the variables and constraints allowed, therefore the implementation of our mapping was done in a straightforward manner. For JaCoP we used FDV variables (FDV stands for Finite Domain Variables) to represent the features while IntVar variables were used in the Choco implementation.

4.2 The Experiments

With the following experiments we intend to demonstrate which solver provides the best performance in the automated analyses of FMs. In addition, we studied the robustness and the areas of vulnerability of each solver. In order to evaluate both solvers we used five FMs. Three of them represent small and medium size real systems, meanwhile the larger two were generated randomly for this experiment. After formulating each one as a CSP in both platforms, we proceeded with the execution. Table 1 summarizes the characteristics of the experiments. Experiment 1 is the FM that was presented in [4]. It is a simple FM representing a Home Integration System. Experiment 2 is the FM of Figure 2 which represents a collaborative web based system. Experiment 3 is a medium size FM of a flight booking system based on the work done by [11,12]. Finally, we generated two larger FMs randomly (Experiments 4 and 5) with a double aim: representing more complex systems with a greater number of features and dependencies, and evaluating the solvers' performance in limit situations. We considered it was necessary to compare the performance with small, medium and large FMs in order to evaluate solver performance results in different situations.

Table 1. Experiments

Experiment	N. of Features	N. of Dep
1	15	0
2	14	2
3	26	0
4	40	14
5	52	28

The process to generate a FM randomly is based on a recursive method that has five input parameters: height levels, maximum number of children relations for a node, maximum cardinality number, maximum number of elements in a set relation and number of dependencies. Firstly, features and their relations are generated using random values. Secondly, the dependencies are created by taking pairs of features randomly and establishing a random dependency (includes or excludes) between them. We took care not to generate misconceptions (e.g. a child depends on a parent).

As exposed in [5], there are some operations that can be performed. For our experiments we performed two operations: *i*) finding one configuration that would satisfy all the constraints, that is, a product and *ii*) finding the total number of configurations of a

given FM. The first is the simplest operation while the second is the most difficult one in terms of performance because it is necessary to retrieve all possible combinations.

The comparison focused on the data obtained from several executions in order to avoid as much exogenous interferences as possible. The total number of executions to calculate the average time was ten. The data extracted from the tests was:

- Number of features in the first solution obtained by solver.
- Average execution time to obtain 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 machine equipped with a 3.2Ghz Intel Pentium IV microprocessor and 1024 MB of DDR 166Mhz RAM memory.

4.3 The Results

The experimental comparison revealed some interesting results (see Figures 4, 5 and 6). The first evidence we should mention is that JaCoP is on average 54% faster than Choco in finding a solution. It is important to observe that our approach is feasible because the necessary time to obtain a response is really low (35 milliseconds in the worst case).

However, while JaCoP is much faster than Choco in finding the total number of solutions in small CSPs, JaCoP seems to be noticeably slower than Choco in the big ones (see Figure 6). This curious result probably depends on how each solver is used to obtain the number of solutions. Choco has a simple method to know the number of solutions of a concrete problem (`Solver.getNbSolutions()`), while JaCoP implementation needs to find all the solutions first and count them afterwards. This simple variation implies a very important difference in performance. For instance, in test 5 JaCoP needs to create 61440 ArrayLists and fill all of them with all the solutions which produces a great time loss. On the other hand, Choco does not have this weakness as its method to find the number of solutions only returns five solutions to avoid memory deficit problems. If the user wants to obtain the other solutions he only has to make a simple iteration and take them one by one. In the three smaller experiments, JaCoP is faster than Choco so we presume that this trend would continue if JaCoP optimized this aspect. In test 5, we performed an experiment to find and return all the solutions in both solvers, that is, not only to find the number of solutions but the solutions themselves. The result was decisive: Choco required over a minute to perform this task, proving to be slower than JaCoP in this situation.

Although memory usage was not a relevant data in our experiments we noticed that in general Choco uses more memory than JaCoP; however there is not a remarkable difference between both solvers.

Finally, we identified some interesting characteristic in both solvers. Firstly, JaCoP allows the user to obtain easily from executions more interesting information than

JACOP / CHOCO						
Experiment	Features in Sol.	Time one Sol.		N° Solutions	Time all Sol.	
		JACOP	CHOCO		JACOP	CHOCO
1	7	9,9	18,8	32	37,5	45,5
2	8	9,4	22,7	68	64,4	81,3
3	13	12	24	512	225,6	265,3
4	19	20,2	34,9	34560	5619	2203,3
5	19	24,4	35,8	61440	15390,8	4817,6

Figure 4. Experimental Results of JaCoP and Choco Solvers

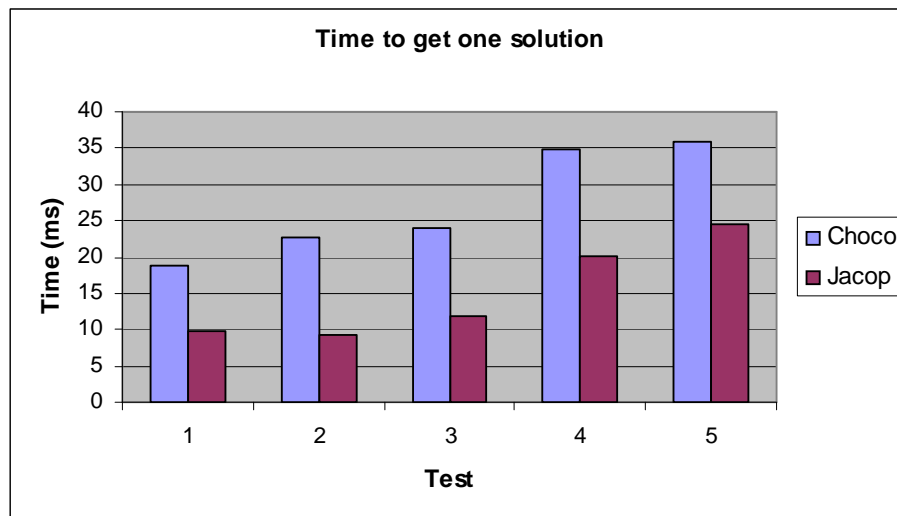


Figure 5. Comparing JaCoP and Choco getting one solution

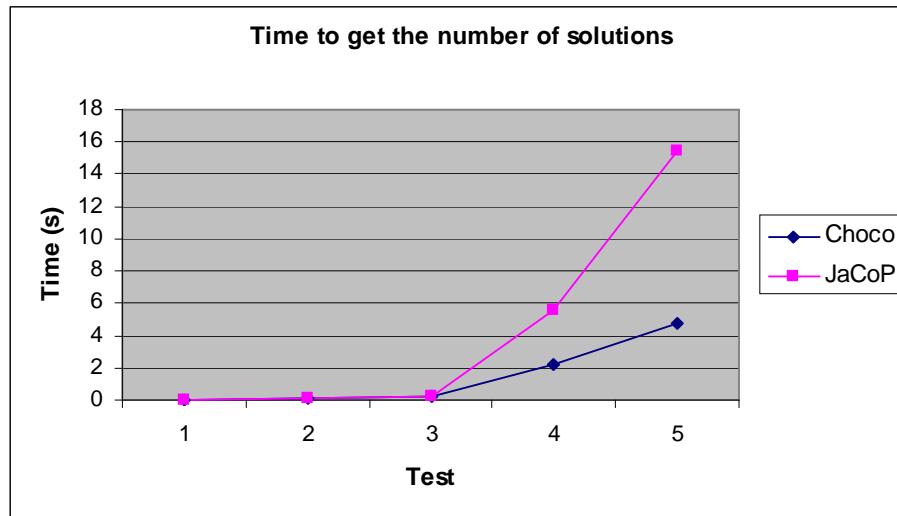


Figure 6. Comparing JaCoP and Choco getting the number of solutions

Choco such as the number of backtracks of a search or the number of decisions taken to find a solution. In second place, we found a worrying bug when working with big problems in Choco. In most cases, executions of CSPs representing big FMs generated an exception (`choco.bool.BinConjunction`) which imposes an important limitation to Choco.

5 Conclusion and Future Work

In this paper we presented how to translate a cardinality-based feature model into a constraint satisfaction problem. We performed a comparative test between two off-the-shelf CSP Java solvers and offered some interesting performance conclusions. The test showed that JaCoP is faster than Choco except in finding the number of solutions. JaCoP gives more details about executions than Choco such as the number of backtracks or the number of decisions. Choco has an important bug when working with big FMs while it is a good open source alternative especially for small and medium size problems. Both solvers have a similar memory usage. Nevertheless, both JaCoP and Choco are useful for the experiments presented in the paper as executions times are really low (milliseconds).

Several challenges remain for our future work. We plan to extend the experiments in order to scale our proposal and compare the results. Bigger experiments with more features and more dependencies are needed and we plan to perform those experiments in the future. Furthermore, we think that we should compare our proposal with others using different representations like SAT or BDDs to complement our results.

References

1. D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.
2. D. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *Summer school on Generative and Transformation Techniques in Software Engineering*, 2005.
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. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.
5. 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'05)*, July 2005.
6. D. Benavides, S. Trujillo, and P. Trinidad. On the modularization of feature models. In *First European Workshop on Model Transformation*, September 2005.
7. J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 1th edition, 2000.
8. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.
9. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, may 2000. ISBN 0–201–30977–7.
10. K. Czarnecki, S. Helsen, and U.W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
11. O. Díaz, S. Trujillo, and F.I. Anfurrutia. Supporting production strategies as refinements of the production process. In *to be published at Software Product Line Conference (SPLC 2005)*, 2005.
12. O. Díaz, S. Trujillo, and I. Azpeitia. User-Facing Web Service Development: A Case for a Product-Line Approach. In Boualem Benatallah and Ming-Chien Shan, editors, *Technologies for E-Services, 4th VLDB International Workshop (VLDB-TES 2003)*, volume 2819 of LNCS, pages 66–77. Springer-Verlag, 2003.
13. P. Fernandez and M. Resinas. James project. Available at <http://jamesproject.sourceforge.net/>, 2002-2005.
14. E. C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, April 1997.
15. G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Journal on Software and Systems Modeling*, 2(1):15–36, 2003.
16. 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.
17. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
18. K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, July 2003.
19. F. Laburthe and N. Jussien. Choco constraint programming system. Available at <http://choco.sourceforge.net/>, 2003-2005.
20. C. Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.