Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible

Martin Fagereng Johansen[1,2], Øystein Haugen[1], and Franck Fleurey[1]

[1] SINTEF ICT, Pb. 124 Blindern, 0314 Oslo, Norway
{Martin.Fagereng.Johansen,Oystein.Haugen,Franck.Fleurey}@sintef.no
[2] Institute for Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway

Abstract. Feature models and associated feature diagrams allow modeling and visualizing the constraints leading to the valid products of a product line. In terms of their expressiveness, feature diagrams are equivalent to propositional formulas which makes them theoretically expensive to process and analyze. For example, satisfying propositional formulas, which translates into finding a valid product for a given feature model, is an NP-hard problem, which has no fast, optimal solution. This theoretical complexity could prevent the use of powerful analysis techniques to assist in the development and testing of product lines. However, we have found that satisfying realistic feature models is quick. Thus, we show that combinatorial interaction testing of product lines is feasible in practice. Based on this, we investigate covering array generation time and results for realistic feature models and find where the algorithms can be improved.

Keywords: Software Product Lines, Testing, Feature Models, Practical, Realistic, Combinatorial Interaction Testing.

1 Introduction

A software product line is a collection of systems with a considerable amount of code in common. The commonality and differences between the systems are commonly modeled as a feature model. Testing of software product lines is a challenge since testing all possible products is intractable. Yet, one has to ensure that any valid product will function correctly. There is no consensus on how to efficiently test software product lines, but there are a number of suggested approaches. Each of the approaches still suffers from problems of scalability (Section 2).

Combinatorial interaction testing [4] is a promising approach for performing interaction testing between the features in a product line. Most of the difficulties of combinatorial interaction testing have been sorted out, but there is one part of it that is still considered intractable, namely finding a single valid configuration, an NP-hard problem. This is thus the bottleneck of the approach. In this paper we resolve this bottleneck such that combinatorial interaction testing should not be considered intractable any more (Section 3). We then investigate how a basic covering array generation algorithm performs on realistic feature models (Section 4), and suggest, based on the resolution of the bottleneck and on the empirics, how the algorithm can be improved (Section 5).

2 Background

2.1 Software Product Lines

A software product line (SPL) [19] is a collection of systems with a considerable amount of code in common. The primary motivation for structuring one's systems as a product line is to allow customers to have a system tailored for their purpose and needs, while still avoiding redundancy of code. It is common for customers to have conflicting requirements. In that case, it is not even possible to ship one system for all customers.

The Eclipse products [22] can be seen as a software product line. Today, Eclipse lists 12 products on their download page[1]. These products share many components, but all components are not offered together as one single product. The reason is that the download would be unnecessary large, since, for example, a C++ systems programmer usually does not need to use the PHP-related features. It would also bloat the system by giving the user many unnecessary alternatives when, for example, creating a new project. Some products contain early developer releases of some components, such as Eclipse for modeling. Including these would compromise the stability for the other products. Thus, it should be clear why offering specialized products for different use cases is good.

One way to model the commonalities and differences in a product line is using a feature model [10]. A feature model sets up the commonalities and differences of a product line in a tree such that configuring the product line proceeds from the root of the tree. Please refer to an example of a feature model for a subset of Eclipse in Figure 1. Proceeding from the root, configuring the product line consists of making a decision for each node in the tree. Each node represents a feature of the product line. The nature of this decision is modeled as a decoration on the edges going from a node to another. For example, in Figure 1, one has to choose one windowing system which one wants Eclipse to run under. This is modeled as an empty semi-circle on the outgoing edges. When choosing a team functionality provider, one or all can be chosen. This is modeled as a filled semi circle. The team functionality itself is marked with an empty circle. This means that that feature is optional. A filled circle means that the feature is mandatory. One has to configure the feature model from the root, and one can only include a feature when the preceding feature is selected. For example, supporting CVS over SSH requires that one has CVS.

The parts that can be different in the products of a product line are usually called its *variability*. One particular product in the product line is called a *variant* and is specified by a configuration of the feature model. Such a configuration consists of specifying whether each feature is included or not.

[1] `http://eclipse.org/downloads/`

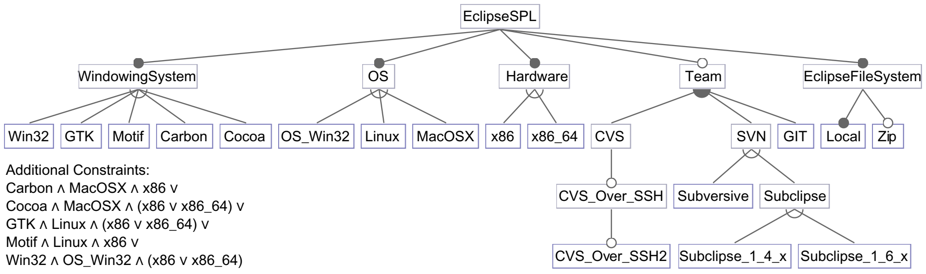

Fig. 1. Feature model for a subset of Eclipse

2.2 Software Product Line Testing

Testing a software product line poses a number of new challenges compared to testing single systems. It has to be ensured that each possible configuration of the product line functions correctly. One way to validate a product line is through testing, but testing is done on a running system. The software product line is simply a collection of many products. One cannot test each possible product, since the number of products in general grows exponentially with the number of features in the product line. For the feature model in Figure 1, the number of possible configurations is 512, and this is a relatively simple product line.

There is no single recommended approach available today for testing product lines efficiently [5], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [4], discussed below; reusable component testing, seen in industry [9], but which does not test for interaction faults in the product line; a technique called ScenTED, where the idea is to express the commonalities and differences on the UML model of the product line and then derive concrete test cases by analyzing it [21]; and incremental testing, where the idea is to automatically adapt a test case from one product to the next using the specification of similarities and differences between the products [25].

2.3 Combinatorial Interaction Testing for Product Lines

Combinatorial interaction testing [4] is one of the most promising approaches. The benefits of this approach is that it deals directly with the feature model to derive a small subset of products which can then be tested using single system testing techniques, of which there are many good ones. The idea is to select a small subset of products where the interaction faults are most likely to occur. For example, we can select the subset of all possible products where each pair of features is present. This includes the cases where both features are present, when one is present, and when none of the two are present. Table 1 shows the 22 products that must be tested to ensure that every pair-wise interaction between the features in the running example functions correctly. Each row represents one feature and every column one product. 'X' means that the feature is included for the product, '-' means that the feature is not included. Some features are included for every product because they are mandatory, and some pairs are not covered since they are invalid according to the feature model.

Table 1. Pair-wise coverage of the feature model in Figure 1 the test suites numbered

Feature\ Product	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
EclipseSPL	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
WindowingSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
GTK	-	X	-	-	-	-	X	-	-	X	-	-	-	X	-	-	X	-	-	-	-	-
Motif	-	-	-	-	X	-	-	X	-	-	-	X	-	-	-	-	-	-	X	-	-	-
Carbon	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	X	-	X	-	-	-	-
Cocoa	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	X	-
OS	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
OS_Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
Linux	-	X	-	-	X	-	X	X	-	X	-	X	-	X	-	-	X	-	X	-	-	-
MacOSX	X	-	-	X	-	-	-	-	-	-	X	-	X	-	-	X	-	X	-	X	X	-
Hardware	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
x86	X	-	-	X	X	-	-	X	-	X	X	X	-	-	X	X	-	X	X	-	-	-
x86_64	-	X	X	-	-	X	X	-	X	-	-	-	X	X	-	-	X	-	-	X	X	X
Team	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CVS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X
CVS_Over_SSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X
CVS_Over_SSH2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X
SVN	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X	X
Subversive	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	-	X
Subclipse	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	X	-	X	X	X	X	-
Subclipse_1_4_x	-	-	-	-	-	-	-	X	X	X	-	-	-	-	-	X	-	-	-	-	X	-
Subclipse_1_6_x	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	X	X	X	-	-
GIT	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	X	X	-	X	X	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	-	-	X	X	-	-	-	X	-	-	-	-	X	-	-	-	-	-	X	-	X

Testing every pair is called 2-wise testing, or pair-wise testing. This is a special case of t-wise testing where $t = 2$. 1-wise coverage means that every feature is at least included and excluded in one product, 3-wise coverage means that every combination of three features are present, etc. For our running example, 5, 64 and 150 products is sufficient to achieve 1-wise, 3-wise and 4-wise coverage, respectively.

An important motivation for combinatorial interaction testing is a paper by Kuhn et al. 2004 [11]. They indicated empirically that most bugs are found for 6-wise coverage, and that for 1-wise one is likely to find on average around 50%, for 2-wise on average around 70%, and for 3-wise around 95%, etc.

There are three main stages in the application of combinatorial interaction testing to a product line. First, the feature model of the system must be made. Second, the subset of products must be generated from the feature model for some coverage strength. Such a subset is called a t-wise covering array for a coverage strength t. Last, a single system testing technique must be selected and applied to each product in this covering array. The first and last of these stages are well understood. The second stage, however, is widely regarded as intractable, thereby rendering the approach useless for industrial size software product lines.

3 The Case for Tractable t-wise Covering Array Generation

3.1 Complexity Analysis of Covering Array Generation

The generation of t-wise covering arrays is equivalent to the minimum set cover problem, an NP-complete problem. Given a set of elements, for example $U = \{1, 2, 3, 4, 5\}$; we have a set of sets of elements from U, for example $S = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$. The set cover problem is to identify the minimum number of sets, C, from S such that the union contains all elements from U, which is for the example $C = \{\{1, 2, 3\}, \{4, 5\}\}$.

1-wise covering array generation is easily converted to a set cover problem by listing all valid configurations of the product line, and having that as S. Each element of U is a pair with the feature name and a Boolean specifying the inclusion or exclusion of the feature. Solving this set cover problem then yields a 1-wise covering array. This can be done similarly for $t > 1$ by having tuples of assignments in U.

The set cover problem has a known approximation algorithm. (An approximation in this context is not the degree of t-wise coverage, which is 100% for all the discussion in this paper; but how many more products are selected than absolutely necessary.) The approximation algorithm was presented in Chvátal 1979 [3]. It is a greedy algorithm with a defined upper bound for the degree of approximation which grows with the size of the problem, but the degree of approximation remains acceptable. The algorithm is quite simple; it selects the set in S which covers the most uncovered elements until all elements are covered. For t-wise testing, this means selecting the product which covers the most uncovered tuples.

The set cover problem assumes that the sets with which to cover are already available so that one can look at all of them. For feature models, the solution space grows exponentially with respect to the number of features. Thus, it is infeasible to iterate through all the valid configurations.

And it gets worse, even generating a single configuration of a feature model is equivalent to the Boolean satisfiability problem (SAT), an NP-hard problem. SAT is the problem of assigning values to the variables of a propositional formula such that the formula evaluates to true. Batory 2005 [1] showed that ordinary feature models are equivalent to propositional formulas with respect to expressiveness, and that a feature model can easily be converted to a propositional formula.

Approximating the SAT problem is not possible: either we have the solution or we do not. This is also why the literature on combinatorial interaction testing classifies the generation of covering arrays as intractable.

3.2 Quick Satisfiability of Realistic Feature Models

Nie and Leung 2011 [16] is a recent survey of combinatorial testing. They state that their survey is the first complete and systematic survey on this topic. They found 50 papers on the generation of covering arrays. Covering array generation is reported to be NP-hard, but no detailed analysis is given. Such an analysis is given in both Perrouin et al. 2010 [18] and Garvin et al. 2011 [6] which both classify covering array generation as intractable because finding a single configuration of a feature model is equivalent to the Boolean satisfiability problem.

And this is indeed the general case given an arbitrary, grammatically valid feature model, but is it so in practice? It was observed by Mendonca et al. 2009 [15] that SAT-based analysis of realistic feature models with constraints is easy, but they did not identify the theoretical reason for this nor whether it is necessarily so and suggested finding the theoretical explanation as future work.

We propose that the theoretical explanation simply is that realistic feature models must be easily configurable by customers in order for them to efficiently use them. Configuring a feature model is equivalent to solving the Boolean satisfiability problem for the feature model.

The primary role of feature models in software product line engineering is for a potential customer to be able to sit down and configure a product to fit his or her needs. Imagine the opposite case. A company has developed a product line, but finding a single product of the product line takes a million years since there is no tractable solution to NP-hard problems. This situation is absurd. If it is really that difficult to find even a single product in a product line, then the feature model is too difficult for customers to use. If the customers cannot configure a feature model by hand assisted by a computer, is not an important point of the product line approach lost?

The same argument also shows that finding the solution to a partially configured feature model remains quick. If not, a customer might come into the situation that he or she cannot manage to complete the product configuration.

The kind of complexity that gives rise to modern computers being unable to solve a Boolean satisfiability problem in a timely manner would start challenging what is understandable by an engineer maintaining the product line.

Therefore, for the class of feature models intended to be configured by humans assisted by computers, which we think at least is a very large part of the realistic feature models, quick satisfiability is also a property.

3.3 Configuration Space

Even if the satisfiability of a realistic feature model is quick, traversal of the configuration space is still an issue. The configuration space of a feature model grows exponentially with the number of features, so one cannot traverse this space looking for the configuration that covers the most uncovered tuples, as required by Chvátal's greedy approximation algorithm.

Even if one only manages to cover one tuple per iteration, the upper bound for both time and the numbers of products is polynomial, since the number of tuples is $\binom{f}{t}$ (where f is the number of features and t the coverage strength; for example, $\binom{f}{2}$ gives the number of ways we can select a pair out of the configured features where order does not matter.) It is highly likely, however, that one is able to quickly cover many tuples. For pair-wise coverage, finding the first product covers $\binom{f}{2}$ out of $4\binom{f}{2}$ pairs for the worst case scenario. This is at least 25% of the possible pairs.

Covering many tuples at each iteration is still a challenge, but the upper bound of the penalty is polynomial. Since it is not feasible to traverse the configuration space to find the product which covers the most tuples, neither is it possible to guarantee the upper bound for the approximation with Chvátal's greedy algorithm. As we will see in the section on empirics, this does not seem to be a problem as one is usually able to cover many tuples per iteration.

3.4 Tractable Approximation of Covering Arrays

Since finding a covering array consists of two parts, finding valid configurations and solving the set cover problem, and since the former was shown to be tractable and the second is approximable by Chvátal's algorithm, we conclude that finding an approximation of the covering array is also tractable for realistic feature models.

4 Performance of Chvátal's Algorithm for Covering Array Generation

Even if the generation of covering arrays can be shown to be tractable, some improvement of the algorithms still have to be done in order to generate covering arrays from some of the largest known feature models. Let us look at how a basic implementation of Chvátal's algorithm for generating covering arrays performs and then discuss how to improve it.

The following algorithm assumes a feature model, FM, has been loaded, and a strength, t, of the wanted coverage strength has been given. From the set of assignments, (f, i), where f is a feature of FM, and i is a Boolean specifying whether f is included, all combinations of t assignments are generated and placed in a set, U. This set then includes all valid and invalid tuples.

An Adaption of Chvátal's Algorithm for Covering Array Generation.

```
While U is not empty:
  c is a configuration of FM with no variables assigned.
  For each tuple e in U:
    Satisfy FM assuming the assignments in both c and e.
    If satisfiable: Fix the assignments of e in c. Remove e from U.
  Satisfy FM assuming c, add the solution to the covering array C.
  //At some point, decide to remove the invalid tuples from U.
  If the number of newly covered tuples < number of features:
    For each tuple e in U:
      If FM is not satisfiable assuming e, remove e from U.
//C now holds the covering array of FM of strength t.
```

4.1 Models

Sometimes in papers discussing combinatorial interaction testing, experiments are run on randomly generated feature models. The problem with that is that one is assuming things about feature models that might not be realistic. Here, performance measurements will be run on realistic feature models, so that no assumptions are made on the nature of realistic feature models.

Models[2] were gathered from some available sources within software product line engineering research where the models are open and available. All the feature models are either of actual product lines or related to publications. The models are listed in Table 2 together with the product line name and their source.

Table 2. Models and Sources

System name	Model File Name	Source
X86 Linux kernel 2.6.28.6	2.6.28.6-icse11.dimacs	[23]
Part of FreeBSD kernel 8.0.0	freebsd-icse11.dimacs	[23]
eCos 3.0 i386pc	ecos-icse11.dimacs	[23]
e-Shop	Eshop-fm.xml	[12]
Violet, graphical model editor	Violet.m	`http://sourceforge.net/projects/violet/`
Berkeley DB	Berkeley.m	`http://www.oracle.com/us/products/database/berkeley-db/index.html`
Arcade Game Maker Pedagogical Product Line	arcade_game_pl_fm.xml	`http://www.sei.cmu.edu/productlines/ppl/`
Graph Product Line	Graph-product-line-fm.xml	[13]
Graph Product Line Nr. 4	Gg4.m	an extended version of the Graph Product line from [13]
Smart home	smart_home_fm.xml	[27]
TightVNC Remote Desktop Software	TightVNC.m	`http://www.tightvnc.com/`
AHEAD Tool Suite (ATS) Product Line	Apl.m	[24]
Fame DBMS	fame_dbms_fm.xml	`http://fame-dbms.org/`
Connector	connector_fm.xml	a tutorial [26]
Simple stack data structure	stack_fm.xml	a tutorial [26]
Simple search engine	REAL-FM-12.xml	[14]
Simple movie system	movies_app_fm.xml	[17]
Simple aircraft	aircraft_fm.xml	a tutorial [26]
Simple automobile	car_fm.xml	[28]

4.2 Tool and Transformations

The models gathered were of many different formats. Software product line engineering is an active field of research, and there are many research tools for different purposes and with various strengths and weaknesses.

In order to measure the performance of covering array generation on the gathered models, integration and some modification of existing tools and libraries were needed to make them cooperate. Figure 2 shows the overview of the tool

[2] The models are available at the following URL: `http://heim.ifi.uio.no/martifag/models2011/fms/`

that was constructed for this purpose[3]. The figure is of no particular graphical modeling notation. The diamonds symbolize files with a certain suffix, the boxes symbolize internal data structures and the arrows symbolize transformations between the formats.

The tool accepts feature models in three different formats: GUI DSL (model names suffixed with '.m'), as shipped with earlier versions of Feature IDE; SXFM, the Simple XML Feature Model format (model names suffixed with '.xml') and dimacs (model names suffixed with '.dimacs'), a file format for storing propositional formulas in conjunctive normal form (CNF).

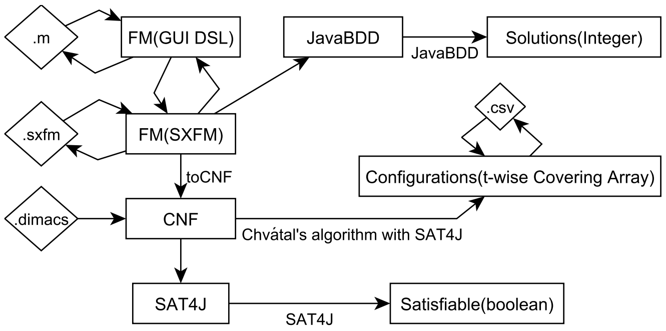

Fig. 2. Transformations in the tool

The GUI DSL files can be loaded using the Feature IDE library. This library allows writing and reading of SXFM files. Thus, they can be loaded into the SPLAR library[4] along with other SXFM files.

The SPLAR library provides an export to conjunctive normal form (CNF), a canonical way of representing general propositional constraints. Thus all the previously loaded models can be converted into CNF formulas, along with other formulas stored in dimacs files.

Once a model is in the form of a CNF formula, it can be given to SAT4J, an open source tool for solving the SAT problem. Thus, all the feature models can be input to the covering array algorithm discussed above. (SAT4J is also used to calculate satisfiability time for the feature models.)

The covering arrays are written to a comma separated values (CSV) file, which can be viewed in Microsoft Excel, Open Office Calc, etc. The covering arrays are then ready to be used to configure products for which single system testing is applied.

(Another interesting thing to know about a feature model is the number of possible configurations. The SPLAR library makes it possible to generate a

[3] The tool is available as open source at `http://heim.ifi.uio.no/martifag/models2011/spltool/`

[4] `http://splar.googlecode.com`

binary decision tree (BDD) which JavaBDD can work with. It then calculates the number of possible configurations of the feature model.)

4.3 Results

Table 3 shows the results from running[5] our tool on the feature models in Table 2. The feature models are ordered after the number of features. The next column shows the number of unique constraints in the model as the number of clauses of the conjunctive normal form of the constraints. (Constraints implied by the structure of the feature diagrams were not included in the count.) The number of valid products for each feature model is available for some of the smaller models, and as can be seen, quickly increases. The next column shows the time, in milliseconds, for running SAT4J on the feature model to find a single valid solution. The following columns show both the size and time for generating covering array of strengths 1–4. Some of the results are not available because the current implementation of the tools to not scale well to these sizes.

Boolean Satisfiability Times for Feature Models. Satisfiability in general has a worst case of about $O(2^n)$ according to Pătraşcu and Williams 2010 [20]. Table 3 shows the satisfiability times for the feature models. Empirically the satisfiability time of the feature models remains low. Thus, our conclusion regarding the quickness of satisfiability of realistic feature models is consistent with these few observations. Note that this is not meant as a validation, but merely as a demonstration of what we discussed in Section 3; that is, it follows from the fact that the feature models are meant to be configured manually.

Covering Array Generation. The following are the statistically significant relations[6] between the number of features and the sizes of the covering arrays. $CA(P, t)$ is the covering array with strength t for the propositional formula, P, representing a feature model with F features. The size function gives the size of the covering array.

$$log(size(CA(P, 2))) = 0.37 * log(F) + 1.30, \text{ adjusted } R^2\text{: } 0.59$$
$$log(size(CA(P, 3))) = 1.09 * log(F) + 0.00, \text{ adjusted } R^2\text{: } 0.63$$

Covering array sizes of strength 1 and 4 did not allow for a statistical model with a decent fit to be made. The fit for strengths 2 and 3 are poor. The reason is that covering array sizes are not really dependent on the number of features but on the structure of the feature model. For example, for 1-wise coverage, a covering array of size 2 might be sufficient: a certain assignment of optional features and the inverse.

[5] The computer on which we did the measurements had an Intel Q9300 CPU @2.53GHz and 8 GB, 400MHz ram. All executions ran in one thread.

[6] Adjusted R^2 is a measure, ranging from 0 to 1, of the goodness of fit of a statistical model. A value of 0.90 means that it is very unlikely a random sample would fit this approximation with the same significance, and a value of 0.20 means that it is very likely.

Table 3. Feature Models, satisfiability times, covering array sizes and generation times

Feature Model \ keys	Features	Constraints	Solutions	SAT time (ms)	1-way size	1-way time (ms)	2-way size	2-way time (ms)	3-way size	3-way times (ms)	4-way size	4-way time (ms)
2.6.28.6-icse11.dimacs	6,888	187,193	n/a	125	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
freebsd-icse11.dimacs	1,396	17,352	n/a	18	7	257,324	n/a	n/a	n/a	n/a	n/a	n/a
ecos-icse11.dimacs	1,244	2,768	n/a	12	6	16,178	n/a	n/a	n/a	n/a	n/a	n/a
Eshop-fm.xml	287	22	n/a	5	4	920	22	364,583	n/a	n/a	n/a	n/a
Violet.m	101	90	1.55E+26	1	4	280	28	21,278	121	3,865,245	n/a	n/a
Berkeley.m	78	47	4.03E+09	1	3	250	23	11,195	96	1,974,741	n/a	n/a
arcade_game_pl_fm.xml	61	35	3.30E+09	3	4	249	17	8,219	63	681,044	n/a	n/a
Gg4.m	38	23	960	1	6	171	22	1,903	63	88,355	156	11,915,393
smart_home_fm.xml	35	1	1,048,576	9	2	141	11	1,046	28	33,010	73	1,995,731
TightVNC.m	30	4	297,252	1	4	109	13	917	46	18,144	124	1,404,756
Apl.m	25	3	4,176	1	3	78	10	583	34	8,865	91	429,207
fame_dbms_fm.xml	21	1	320	3	3	109	9	515	24	5,138	49	121,989
connector_fm.xml	20	1	18	3	6	141	15	485	18	4,147	18	48,086
Graph-product-line-fm.xml	20	15	30	3	5	141	15	512	26	4,390	30	88,022
stack_fm.xml	17	1	432	7	3	109	12	409	41	2,471	96	56,086
REAL-FM-12.xml	14	3	126	7	4	94	13	340	33	1,261	66	18,807
movies_app_fm.xml	13	1	24	3	2	78	6	252	14	963	22	6,065
aircraft_fm.xml	13	1	315	7	3	78	10	286	23	1,131	54	9,597
car_fm.xml	9	3	13	7	3	63	7	200	12	425	13	1,141

The following are the estimated relations between the number of features and the time taken in milliseconds of generating the covering arrays.

$$\begin{aligned}
&log(time(CA(P,1))) = 1.46 * log(F) + 0.00, \text{ adjusted } R^2\text{: } 0.84\\
&log(time(CA(P,2))) = 2.13 * log(F) + 0.00, \text{ adjusted } R^2\text{: } 0.96\\
&log(time(CA(P,3))) = 4.03 * log(F) - 3.51, \text{ adjusted } R^2\text{: } 0.98\\
&log(time(CA(P,4))) = 6.02 * log(F) - 6.41, \text{ adjusted } R^2\text{: } 0.97.
\end{aligned}$$

5 Discussion

5.1 Memory Requirements

The way our tool deals with the constraints in a feature model is to calculate and store the valid, uncovered tuples in memory. The tuples need to be traversed in order to find the configurations which cover the most uncovered tuples at each iteration. Doing it this way, the number of constraints does not affect the memory requirement significantly, but memory might prove to be a bottle neck.

This effectively sets the memory requirement to $O(F^t)$, where F is the number of features in a feature model and t is the strength of the coverage. For a system with M bytes of memory and assuming each t-tuple requires $t * x$ bytes, the upper bound for t-wise coverage is $F^t = M/(t * x)$.

For pair-wise coverage on a system with 8GB of memory, and assuming that a structure holding the pairs take 20 bytes, the upper bound is $n = \sqrt{8,000,000,000/20}$, $n = 20,000$ features. This is the upper bound of a high-end laptop. More powerful computers are available which can be used for generating covering arrays which increases the upper bound such that even 3-wise coverage of the second largest feature model in our sample is within.

5.2 Accepted Covering Array Size

There is a correspondence between the number of features in a feature model and the size of the team working with it. Thus a team of developers and testers should be able to deal with a covering array of a size around the same size as the number of features. If we look at the data and statistical models for covering array size, we can see that the size of 1–3-wise covering arrays is below or close to the number of features since the coefficient of log(F), and thus the exponent of F, is less than or close to 1.

5.3 Suggested Improvements and Future Work

Given the evaluations up to this point, there are a number of source of improvement for generating covering arrays for software product lines.

Exploiting the Boolean Satisfiability Speed. Nie and Leung 2011 [16] classified handling constraints for covering array generation is an open problem for covering array generation in general. Using SAT-solvers is good way to handle constraints for covering array generation based on feature models. Also, since satisfiability of feature models has been assumed to be intractable up to this point, it might be an unexploited source for improvement of covering array generation speed.

Parallelization. The algorithm that was used to make the measurements in this paper ran in one thread. An algorithm which supports running on several threads will improve the execution time for generating the covering arrays. For example, the step for finding all invalid tuples in the adaption of Chvátal's algorithm above can be run in parallel by splitting the set of tuples in, for example, four and checking each fourth in a separate thread.

Heuristics. Another unexploited source of improvement for covering array generation is knowledge from the domain model. UML-models and annotations on feature models should be taken into account when generating a covering array to make it smaller and its generation time lower. CVL [7] is a variability language with tool support which, in addition to feature diagrams, models the variability of a system on the system model as well. Knowing what a feature refers to in a system model is an unexploited source of improvement for covering array generation.

In a recent publication [8], we show how to exploit one commonly occurring structure in product lines when doing combinatorial interaction testing. Often there are several implementations of the same basic functionality which is used by the other components in the product line through an abstraction layer. These implementations occur as mutual exclusive alternatives in the feature model. Mutual exclusive alternatives are detrimental to combinatorial interaction testing [2], causing a substantial increase in the number of products in the covering arrays. We show that if the increase of products is due to the abstraction layer implementations, then the number of test suites required can be reduced by reusing test suites for several of the products in the array without losing the bug detection capabilities.

6 Conclusion

In this paper we showed that although it is widely held that configuring feature models is intractable, in practice the role of feature models in software product line engineering implies that it is quick. Boolean satisfiability solvers thus provide an efficient way to handle constraints in feature models and should be exploited for doing covering array generation without the fear that the running time will be intractable.

Acknowledgments. We want to thank the anonymous reviewers for their helpful feedback.

The work presented here has been developed within the VERDE project ITEA 2 - ip8020. VERDE is a project within the ITEA 2 - Eureka framework.

References

1. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
2. Cabral, I., Cohen, M.B., Rothermel, G.: Improving the testing and testability of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 241–255. Springer, Heidelberg (2010)
3. Chvátal, V.: A greedy heuristic for the Set-Covering problem. Mathematics of Operations Research 4(3), 233–235 (1979)
4. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering 34, 633–650 (2008)
5. Engström, E., Runeson, P.: Software product line testing - a systematic mapping study. Information and Software Technology 53(1), 2–13 (2011)
6. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Softw. Engg. 16, 61–102 (2011)
7. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: SPLC 2008: Proceedings of the 2008 12th International Software Product Line Conference, pp. 139–148. IEEE Computer Society, Washington, DC (2008)
8. Johansen, M.F., Haugen, Ø., Fleurey, F.: Bow tie testing - a testing pattern for product lines. In: Proceedings of the 16th Annual European Conference on Pattern Languages of Programming (EuroPLoP 2011), Irsee, Germany, July 13-17 (2011)
9. Johansen, M.F., Haugen, Ø., Fleurey, F.: A survey of empirics of strategies for software product line testing. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW, pp. 266–269 (March 2011)
10. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
11. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30(6), 418–421 (2004)
12. Lau, S.Q.: Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, ECE Department, University of Waterloo, Canada (2006)
13. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: Dannenberg, R.B. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
14. Mendonca, M.: Efficient Reasoning Techniques for Large Scale Feature Models. Ph.D. thesis, School of Computer Science, University of Waterloo (January 2009)
15. Mendonca, M., Wasowski, A., Czarnecki, K.: Sat-based analysis of feature models is easy. In: Proceedings of the 13th International Software Product Line Conference, pp. 231–240. Carnegie Mellon University (2009)
16. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. 43, 11:1–11:29 (2011)
17. Parra, C., Blanc, X., Duchien, L.: Context awareness for dynamic service-oriented product lines. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 131–140. Carnegie Mellon University, Pittsburgh (2009)

18. Perrouin, G., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Automated and scalable t-wise test case generation strategies for software product lines. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST 2010, pp. 459–468. IEEE Computer Society, Washington, DC (2010)
19. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)
20. Pătraşcu, M., Williams, R.: On the possibility of faster sat algorithms. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 1065–1075. Society for Industrial and Applied Mathematics, Philadelphia (2010)
21. Reuys, A., Reis, S., Kamsties, E., Pohl, K.: The scented method for testing software product lines. In: Käkölä, T., Dueñas, J.C. (eds.) Software Product Lines, pp. 479–520. Springer, Heidelberg (2006)
22. Rivieres, J., Beaton, W.: Eclipse Platform Technical Overview (2006)
23. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 461–470. ACM, New York (2011)
24. Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 191–200. ACM, New York (2006)
25. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. IEEE Transactions on Software Engineering 36(3), 309–322 (2010)
26. Voelter, M.: Using domain specific languages for product line engineering. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 329–329. Carnegie Mellon University, Pittsburgh (2009)
27. Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 211–220. Carnegie Mellon University, Pittsburgh (2009)
28. White, J., Dougherty, B., Schmidt, D.C., Benavides, D.: Automated reasoning for multi-step feature model configuration problems. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 11–20. Carnegie Mellon University, Pittsburgh (2009)