An Algorithm for Generating t-wise Covering Arrays from Large Feature Models

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

A scalable approach for software product line testing is required due to the size and complexity of industrial product lines. In this paper, we present a specialized algorithm (called ICPL) for generating covering arrays from feature models. ICPL makes it possible to apply combinatorial interaction testing to software product lines of the size and complexity found in industry. For example, ICPL allows pair-wise testing to be readily applied to projects of about 7,000 features and 200,000 constraints, the Linux Kernel, one of the largest product lines where the feature model is available. ICPL is compared to three of the leading algorithms for t-wise covering array generation. Based on a corpus of 19 feature models, data was collected for each algorithm and feature model when the algorithm could finish 100 runs within three days. These data are used for comparing the four algorithms. In addition to supporting large feature models, ICPL is quick, produces small covering arrays and, even though it is non-deterministic, produces a covering array of a similar size within approximately the same time each time it is run with the same feature model.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Algorithms, Verification

Keywords

Product Lines, Testing, Feature Models, Combinatorial Interaction Testing

1. INTRODUCTION

A software product line (SPL) 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 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 [5], but there are a number of suggested approaches. Each of the approaches still has problems of scalability; this is covered in Section 2.

Combinatorial interaction testing [3] is a promising approach for performing interaction testing between the features of a product line. Most of the difficulties of combinatorial interaction testing have been sorted out.

In [11], we resolved a previously considered bottleneck of combinatorial interaction testing and concluded that generating covering arrays from realistic feature models is tractable. We evaluated a basic algorithm for generating covering arrays. The performance of this basic algorithm was measured on a corpus of 19 realistic feature models. It was concluded that it is possible to make an algorithm that could generate covering arrays from the larger feature models, and that the basic algorithm has insufficient performance.

In this paper we present an algorithm, called ICPL[1] that is able to generate covering arrays from large feature models. Based on the basic algorithm presented in [11], the new algorithm contains optimizations that significantly increases the performance of it, described in detail in Section 4.

We then present the results from an experiment[2], Section 5, using the same corpus of 19 feature models from [11] on ICPL and on three of the leading algorithms for t-wise covering array generation. In this experiment, each of the five algorithms was given an opportunity to run 100 times on each of the 19 feature model for each strength of 1–3. The analysis of these data is presented in Section 5 and gives an extensive insight into how ICPL performs on its own and in comparison to the other algorithms. The experiment shows that ICPL successfully generates a pair-wise covering array for a feature models of about 7,000 features and 200,000 constraints, the Linux Kernel, one of the largest product lines where the feature model is available. In addition to supporting large feature models, ICPL produces small covering arrays overall and has the highest level of scalability. For ICPL, the standard deviation is low for both the time taken to generate the covering arrays and their sizes.

SPLC '12, September 02 - 07 2012, Salvador, Brazil

[1]ICPL is a recursive accronym that stands for "ICPL Covering array generation algorithm for Product Lines".
[2]An open source implementation of ICPL is available at `http://heim.ifi.uio.no/martifag/splc2012/` along with all the measurements, the feature models in the corpus and scripts used in the experiment.

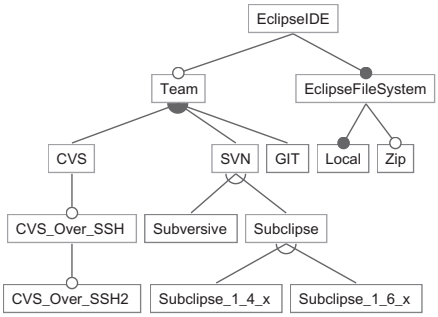

Figure 1: Feature Model for a Subset of the Eclipse IDE Product Line

2. BACKGROUND AND RELATED WORK

2.1 Software Product Lines

A software product line [19] is a collection of systems with a considerable amount of code in common. The primary motivation for structuring a system 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.

For example, the Eclipse IDEs [21] can be seen as a software product line. Today, the Eclipse project lists 11 products on their download page[3].

One way to model the commonalities and differences in a product line is using a feature model[4] [13]. 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. Figure 1 shows a part of the feature model for the Eclipse IDEs. The figure uses the common notation for feature models; for a detailed explanation of feature models, see Czarnecki and Eisenecker 2000 [4].

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 verify 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 64, but 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 [3], discussed below; reusable component testing, which we have seen in industry [12], 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 [20]; 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 [23]. Kim et al. 2011 [14] presented a technique where they can identify irrelevant features for a test case using static analysis. They use that information to reduce the combinatorial number of product to test.

[3] `http://eclipse.org/downloads/`, retrieved 2012-01-16

[4] Throughout this paper, the term feature models is used to mean basic feature models which have equivalent expressiveness as propositional logic formulas.

Table 1: 2-wise Covering array of the Feature Model in Figure 1.

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
Team	X	X	-	X	X	X	X	X	X	X	-	X
CVS	X	X	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH	X	-	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH2	-	-	-	X	X	-	X	-	X	-	-	-
SVN	X	X	-	X	X	X	-	X	X	X	-	-
Subversive	X	-	-	X	-	-	-	X	-	-	-	-
Subclipse	-	X	-	-	X	X	-	-	X	X	-	-
Subclipse_1_4_x	-	X	-	-	X	-	-	-	-	X	-	-
Subclipse_1_6_x	-	-	-	-	-	X	-	-	X	-	-	-
GIT	X	-	-	-	X	X	-	X	-	-	-	X
EclipseFileSystem	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
Zip	-	X	X	X	-	-	-	X	X	-	-	-

2.3 Combinatorial Interaction Testing for Product Lines

Combinatorial interaction testing [3] is one of the most promising approaches. The benefits of this approach are that it deals directly with the feature model to derive a small set 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 12 products that must be tested to ensure that every pairwise 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 excluded. Some features are included for every product because they are core features, and some pairs are not covered since they are invalid according to the feature model. Since every pair is present, this set of products is called a *pairwise covering array*, or *2-wise covering array*. (The latter terminology will be used in this paper.) Testing a product line based on a 2-wise covering array is called *2-wise testing* of the product line.

2-wise covering arrays are a special case of t-wise covering arrays where $t = 2$. 1-wise coverage means that every feature is at least included and excluded in at least one product. 3-wise coverage means that every combination of three features is present, etc. For our running example, 4, 12 and 27 products are sufficient to achieve 1-wise, 2-wise and 3-wise coverage, respectively.

The main motivation behind combinatorial interaction testing are the empirics collected by Kuhn et al. 2004 [15]. The empirics they collected indicate that most bugs are caused by simple interactions between features.

There are three main stages in the application of com-

binatorial interaction testing to a product line. First, the feature model of the system must be made (for example as in Figure 1). Second, the subset of products must be generated from the feature model for some coverage strength (for example as in Table 1). Last, after having built each product, a single system testing technique must be selected and applied to each product.

The first and last of these stages are well understood. For the second stage, however, no scalable algorithm is known[5], thereby rendering the approach less usable for larger industrial size software product lines. An algorithm for generating covering arrays from large feature models, Section 4, and an evaluation of it, Section 5, are the contributions of this paper.

3. THE BASIC ALGORITHM

We have previously presented Algorithm 1 in [11]. Using a greedy algorithm for generating a covering array in this way has certainly been known since the publication of Chvátal's algorithm [2], which presents a greedy heuristic for the set cover problem, of which covering array generation is a one type of. It has been previously discussed in the context of single system testing in Grieskamp et al. 2009 [8] and in Calvagna and Gargantini 2009 [1] among others.

3.1 Groundwork

To explain the algorithms in the following sections, some groundwork is needed.

An assignment a is a pair $(f, included)$, where f is a feature of a feature model FM, and *included* is a Boolean specifying whether f is included or not. A configuration, C, is a set of assignments where all features in the feature model, FM, have been given an assignment. A *t-set* is a set of t assignments, $\{a_1, a_2, ...\}$. The set of all t-sets of a feature model, FM, is denoted T_t, e.g. $T_1, T_2, T_3, ...$ The set of invalid t-sets is denoted I_t, e.g. $I_1, I_2, I_3, ...$ The set of valid t-sets, the universe of the set cover problem, is thus $U_t = T_t \setminus I_t$. A covering array of strength t is a set of configurations, C_t, in which all valid t-sets, all t-sets in U_t, are a subset of (or the same set as) at least one of the configurations.

The feature model is stored in an object and has some methods:

- $size()$. In this context, the size of the feature model is the number of features. $|FM|$ is an alternative notation to get the size of a feature model.
- $genT(t)$ returns the set of all combinations of t feature assignments (both valid and invalid), T_t.
- $satisfy(c)$ returns a valid configuration of the feature model given the assignments in c. Although not a requirement, this function gives a configuration that tends to have no more features included than necessary. This is the default behavior of SAT4J which is used in our concrete implementation of FM's satisfy method.
- $is\_satisfiable(c)$ returns true if the feature model is satisfiable given the assignments in c.

[5]Scalable algorithms are known for special cases, such as when there are no constraints.

3.2 Algorithm

Algorithm 1 requires that a feature model, FM, has been loaded. It also assumes a strength, t, of the wanted coverage strength has been given.

As long as there are more t-sets to cover, the algorithm fills up a new configuration, C, with many uncovered t-sets. The covered t-sets are first added to the set CO, line 9, before being removed from T, line 12.

At a certain point, line 17, all the invalid t-sets will be removed from T, leaving only valid t-sets in T. This point has been empirically determined to be when the order of magnitude of the covered t-sets is the same as the order of magnitude of the size of the feature model, when $(\lfloor log_{10}|CO| \rfloor \leq \lfloor log_{10}|FM| \rfloor)$. For example, with a feature model with 500 features, the invalid t-sets are removed when the number of t-sets covered is 999 or less. This condition could have been removed, meaning that all invalid t-sets would be removed in the first round. This is inefficient since the satisfiability solver would have to check a lot of valid t-sets, line 19. We found that it is better to wait until a lot of valid t-sets have been removed from T. It is also inefficient to wait until the algorithm has covered all valid t-sets, at which point CO would be empty. The reason is that the satisfiability solver will be burdened with a lot of invalid t-sets at line 7. The condition at line 17 was empirically found to be an efficient middle point between these two extremes.

Algorithm 1 Pseudo Code of Chvátal's Algorithm Adapted for Covering Array Generation

```
 1: T ← FM.genT(t)
 2: (invalidRemoved, C_t) ← (false, ∅)
 3: while T ≠ ∅ do
 4:    C ← ∅
 5:    CO ← ∅
 6:    for each t-set e in T do
 7:       if FM.is_satisfiable(C ∪ e) then
 8:          C ← C ∪ e
 9:          CO ← CO ∪ {e}
10:       end if
11:    end for
12:    T ← T \ CO
13:    if CO ≠ ∅ then
14:       C ← FM.satisfy(C)
15:       C_t ← C_t ∪ {C}
16:    end if
17:    if (¬invalidRemoved) ∧ (⌊log10|CO|⌋ ≤ ⌊log10|FM|⌋)
       then
18:       for each t-set e in T do
19:          if ¬FM.is_satisfiable(e) then
20:             T ← T \ {e}
21:          end if
22:       end for
23:       invalidRemoved ← true
24:    end if
25: end while
```

Algorithm 1 is guaranteed to finish with the complete t-wise covering array of FM in C_t. First, all combinations of t assignments are generated at line 1. Then, a t-set is either covered, line 8–9, and therefore removed from T, line 12, or found to be invalid and therefore removed from T at line 20. CO tends towards 0 as more and more valid t-sets are covered. This guarantees that the condition on line 17 will

pass eventually. Since all t-sets are either valid or invalid, the condition 3 is guaranteed to evaluate to false eventually, at which point all t-sets have either been covered or classified as invalid.

4. THE NEW ALGORITHM: ICPL

ICPL works essentially the same way and for the same reasons as Algorithm 1, but ICPL does several things to avoid redundant work, learn from intermediate results and allows doing some things in parallel.

Algorithm 2 shows the highest level of the ICPL algorithm. Just as Algorithm 1, ICPL takes a feature model, FM, and a strength t and generates the covering array of strength t, C_t. In addition, the complete set of invalid t-sets, I_t, is returned. Some parts of ICPL have been put into sub-algorithms that will each be explained in turn in this section.

Lines 10–20 is the same greedy main-loop at in Algorithm 1, except that generating a single configuration, line 11, and finding all covered t-sets, line 13, have been separated from each other and put into sub-algorithms. Finding all invalid t-sets, lines 16–17, has also been separated out into a sub-algorithm. The point at which the invalid t-sets are removed, line 15, is exactly as in Algorithm 1.

Lines 1–9 are different than in Algorithm 1. It has been previously treated and utilized in Fouche et al. 2009 [6], that a covering array of strength $t-1$ is a subset of (or the same set as) a covering array of strength t, that is that $CA(FM, t-1) \subseteq CA(FM, t)$. Thus, before starting to cover for strength t, the algorithm covers for all values of n in $1 \leq n < t$. Thus, before generating C_t, the algorithm knows C_{t-1} and I_{t-1}. This is done in lines 6–8.

Line 2–4 is the bottom of the recursion. Here, the complete set of invalid 1-sets, I_1, is identified, line 2, which gives us the complete U_1 (stored in T_t), line 3. Since the whole I_1 is found, *invalidRemoved* is set to true, line 4.

Algorithm 2 $ICPL(FM, t) : (C_t, I_t)$

```
 1: if t = 1 then
 2:    (C_t, I_t) ← genCompleteI_1(FM)
 3:    T_t ← FM.genT(1) \ I_t
 4:    invalidRemoved ← true
 5: else
 6:    (C_t, I_{t-1}) ← ICPL(FM, t − 1)
 7:    (T_t, I_t) ← generateTSets(FM.getT(t), I_{t-1}, C_t)
 8:    invalidRemoved ← false
 9: end if
10: while T_t ≠ ∅ do
11:    C ← genConfiguration(FM, T_t)
12:    C_t ← C_t ∪ {C}
13:    CO ← getCovered(FM, C, T_t)
14:    T_t ← T_t \ CO
15:    if (¬invalidRemoved)∧(⌊log10|CO|⌋ ≤ ⌊log10|FM|⌋)
       then
16:       (T_t, TempI) ← genInvalid(FM, T_t)
17:       I_t ← I_t ∪ TempI
18:       invalidRemoved ← true
19:    end if
20: end while
21: return (C_t, I_t)
```

4.1 Finding Core and Dead Features Quickly

A step in the algorithm of generating a covering array is finding all core and dead features. Core features are features that must always be included, and dead features are features that can never be included. Therefore, if a feature is core, any assignment excluding it is invalid. Also, if a feature is dead, any assignment including it is invalid. (There should never to be dead features in a feature model, but it is possible to have them.)

The basic way to do this is simply trying to set each feature to included and excluded. Then, if the feature model is unsatisfiable, it implies that the feature is either dead or core, respectively. This is done in Algorithm 3. This algorithm takes the feature model, FM, and a set of 1-sets, T_1. It then checks each assignment to see if it is valid or not, line 3. Those that are invalid are added to the set of invalid 1-sets, I_1. This, hence, is guaranteed to give us all invalid 1-sets of T_1.

Algorithm 3 $getInvalidAssignments(FM, T_1) : I_1$

```
1: I_1 ← ∅
2: for each t-set e in T_1 do
3:    if ¬FM.is_satisfiable(e) then
4:       I_1 ← I_1 ∪ {e}
5:    end if
6: end for
7: return I_1
```

There is a way to speed this up. Algorithm 4 shows how this is done.

Algorithm 4 $genCompleteI_1(FM) : (C, I)$

```
 1: (I, NotCore, NotDead) ← (∅, ∅, ∅)
 2: T ← FM.genT(1)
 3: c_1 ← FM.satisfy(∅)
 4: for each assignment a in c_1 do
 5:    if ¬a.included then
 6:       NotCore ← NotCore ∪ {a}
 7:    end if
 8: end for
 9: IC ← { ∀e ∈ T : e is a not included feature} \ NotCore
10: I ← getInvalidAssignments(FM, IC)
11: c_2 ← FM.satisfyManyInclusions(∅)
12: for each assignment a in c_2 do
13:    if a.included then
14:       NotDead ← NotDead ∪ {a}
15:    end if
16: end for
17: IC ← { ∀e ∈ T : e is an included feature} \ NotDead
18: I ← I ∪ getInvalidAssignments(FM, IC)
19: return ({c_1, c_2}, I)
```

Algorithm 4 gives the same result as Algorithm 3. The difference is that Algorithm 4 learns from two concrete configurations that were constructed a special way in order to do the same thing, only faster. (In addition, Algorithm 4 returns these two configurations.) The first part of the learning experience is line 3–9. Given, for example, a feature model with 7,000 features, the call at line 3 will give a configuration with, for example, 500 features included. This informs us that the remaining 6,500 features are not core features. If they were, they must have been included in a

valid configuration. Thus, we do not have to check that 6,500 assignments with these features not included are invalid. Thus they are removed from the set of candidates for invalid assignments, IC.

The same things is repeated again, line 10–18. But now, a configuration with, for example, 5,000 features included is returned instead, at line 11. This informs us that these 5,000 features are not dead. If they were dead, they could not be included in a valid configuration. Thus, 5,000 assignments with included features can be removed from the set of candidate invalid assignments, line 17. ($satisfyManyInclusions$ inverts the feature model before calling $satisfy$ and then inverts the assignments in the configuration [10].)

4.2 Early Identification of Invalid t-sets

In Algorithm 1, line 1, the complete T_t was given. However, it is possible to be more clever in this step. How this is done is shown in Algorithm 5. In ICPL we use the knowledge from the covering array and the set of invalid t-sets from the $t-1$ step of the recursion, given as C_t and I_{t-1}.

Algorithm 5 $generateTSets(T, I_{t-1}, C_t) : (T, I_t)$

```
 1: (NewT, I_t) ← (∅, ∅)
 2: Loop:
 3: for each t-set e in T do
 4:   for each configuration C in C_t do
 5:     if e ⊆ C then
 6:       continue Loop
 7:     end if
 8:   end for
 9:   for each t-set i in I_{t-1} do
10:     if i ⊆ e then
11:       I_t ← I_t ∪ {e}
12:       continue Loop
13:     end if
14:   end for
15:   NewT ← NewT ∪ {e}
16: end for
17: return (NewT, I_t)
```

If we removed lines 4–14, what we would get is the set T_t returned as $NewT$. This would give us the same result as Algorithm 1, line 1. But, two things are done in ICPL:

On lines 4–8, if a t-set has already been covered by a configuration in C_t, we do not have to cover it again. Thus, the t-set is skipped.

On lines 9–14, if i is an invalid (t-1)-set of I_{t-1}, so are all t-sets $e \in T$, for which $i \subseteq e$. Thus, those t-sets can be added to the set of invalid t-sets, line 11.

4.3 Generating a Single Configuration

Algorithm 6 finds a single configuration, which assignments are collected in C, that covers many uncovered t-sets.

This is essentially the same as is done in Algorithm 1, line 6–8. These three lines appear again in Algorithm 6 as lines 8, 16 and 17, but ICPL learns from the outcome of line 16 to avoid redundant work. It does not record which t-sets have been covered, as that is done in Algorithm 7 instead.

First of all, the set CF maintains a list of features that have been given an assignment. This enables ICPL to skip a t-set without having to pass it to the satisfiability solver, lines 9–15. It also enables us to break the search early, lines

Algorithm 6 $genConfiguration(FM, T) : C$

```
 1: (C, CF, PF) ← (∅, ∅, ∅)
 2: for each t-set e in T do
 3:   for each assignment a in e do
 4:     PF ← PF ∪ {a.f}
 5:   end for
 6: end for
 7: Loop:
 8: for each t-set e in T do
 9:   F ← ∅
10:   for each assignment a in e do
11:     F ← F ∪ {a.f}
12:   end for
13:   if F ⊆ CF then
14:     continue
15:   end if
16:   if FM.is_satisfiable(C ∪ e) then
17:     C ← C ∪ e
18:     CF ← CF ∪ F
19:   else
20:     for each assignment a in e do
21:       if ((e \ {a}) ⊆ C) ∧ (a ∉ C) then
22:         b ← a with assignment inverted
23:         C ← C ∪ {b}
24:         CF ← CF ∪ {b.f}
25:       end if
26:     end for
27:   end if
28:   if |CF| = |PF| then
29:     break Loop
30:   end if
31: end for
32: return FM.satisfy(C)
```

28–30, if all features of T, found in lines 2–6 and stored in PF, have been given an assignment.

When a t-set leads to unsatisfiability, line 19, and all assignments in the t-set except one are already in the configuration, C, line 21, we know that the opposite assignment of the only exception must be valid. This allows us to update the configuration, lines 22–24.

Algorithm 6 is guaranteed to give us a valid configuration that covers many uncovered t-sets. In line 32, we return a complete, valid configuration by calling the satisfiability solver. The main loop, lines 8–31, is guaranteed to end since there are only a finite number of t-sets in T which is not modified inside the loop. If there are valid, uncovered t-sets, then at least one will get covered on line 17, since set CF is empty, guaranteeing the check at lines 13 and 28 to fail until at least one t-set has been covered.

4.4 Finding the Covered t-sets

It is necessary to find the covered t-sets after a configuration was generated by Algorithm 6 because of the optimizations applied to it. In Algorithm 1, the covered t-sets were collected as they were covered. The result is the same in both cases: A configuration, C, and the set of covered t-sets, CO; but the combined speed is faster since Algorithm 6 is so much more efficient. How the covered t-sets are found in ICPL is shown in Algorithm 7.

Since ICPL deals with large feature models, it is important that the outer iteration of Algorithm 7, line 2, is on the set

Algorithm 7 $genCovered(FM, C, T) : CO$

```
1: CO ← ∅
2: for each t-set e in T do
3:    if e ⊆ C then
4:       CO ← CO ∪ {e}
5:    end if
6: end for
7: return CO
```

of uncovered tuples, T. T shrinks drastically as our greedy algorithm covers more and more t-sets. By iterating over T, Algorithm 7 speeds up towards the end of generating the covering array.

4.5 Identifying the Invalid t-sets

In Algorithm 1, the invalid t-sets were identified by checking each remaining t-set in turn, lines 18–22. Algorithm 8 does the same thing, but faster.

Algorithm 8 takes the feature model, FM, and the set of uncovered t-sets, T_t at line 16 of Algorithm 2, and classifies all the remaining t-sets as either valid, V, or invalid, I, and returns them.

Algorithm 8 $genInvalid(FM, T) : (V, I)$

```
1: (I, V) ← (∅, ∅)
2: while T ≠ ∅ do
3:    e ← any t-set in T
4:    if ¬FM.is_satisfiable(e) then
5:       I ← I ∪ {e}
6:       T ← T \ {e}
7:    else
8:       V ← V ∪ {e}
9:       T ← T \ {e}
10:      C ← FM.satisfy(e)
11:      NewV ← ∅
12:      for each t-set g in T do
13:         if g ⊆ C then
14:            NewV ← NewV ∪ {g}
15:         end if
16:      end for
17:      V ← V ∪ NewV
18:      T ← T \ NewV
19:   end if
20: end while
21: return (V, I)
```

Each t-set is classified in turn, line 4, and added to and removed from the respective sets, lines 5–6 and 8–9. Lines 11–18 contain an optimization. If a configuration, found in line 10, contains one of the unclassified t-sets, we can classify it as valid.

Algorithm 8 is guaranteed to finish since no elements are added to T, and each element picked is guaranteed to be either classified as valid or invalid.

4.6 Parallelization

Modern computers now commonly support truly parallel execution of programs. Algorithm 1 runs in one process only. It is neither trivial nor certain that an algorithm or parts of it can be parallelized, and no automatic way of parallelizing an arbitrary algorithm exists.

In ICPL, Algorithm 3, 5, 7 and 8 are all *data parallel*. Data parallel algorithms are easy to parallelize [9]. For example, if we have an array of integers, the algorithm that increases each element by 1 is data parallel. If the array has 1000 integers, then, for example, 4 processors can in parallel increase 250 elements each, giving the same result as if all 1000 had been increased by the same processor.

A similar parallel execution is easily done with the four data parallel sub-algorithms of ICPL. Each get a set of t-sets to work with and works on one t-set in each iteration.

Although not shown in the pseudo-code in this paper, our implementation of ICPL used in the experiment in the next section uses data parallel execution of these four algorithms, supporting any number of processors. It is not shown in the pseudo code because it requires uninteresting boiler-plate code to set things up properly. Instead, we give a summary of the parameters that can be split:

Algorithm 3 takes two parameters, the feature model object and the set T_1. T_1 is split up into n chunks, and the algorithm is executed in parallel in n threads with each of the n chunks and a copy of the feature model object. When all executions are done, there are n results available. These can be combined into the final result, I_1.

Algorithms 5, 7 and 8 are data parallel on parameter T. Their return values can be combined on completion.

5. RESULTS AND COMPARISON

We have set up an experiment[6] in order to demonstrate the performance of ICPL and to compare it to the performance of existing algorithms. Since the scope of ICPL is realistic feature models, we have reused without modification a collection of 19 realistic feature models from academia and industry previously presented in Johansen et al. 2011 [11]. The three largest feature models in our collection were presented in and provided by She et al. 2011 [22][7].

We compared ICPL with Algorithm 1 [11], CASA [7], IPOG [16], as implemented in the tool NIST ACTS, and MoSo-PoLiTe [17]. Each algorithm was run 100 times for each supported coverage strength, 1–3, with 100% coverage. For each run we recorded the size of the covering array and the time taken to generate it[8].

5.1 Software Setup

We have implemented an open source tool[9] with ICPL supporting 1–3-wise covering array generation from feature models. The algorithm is implemented as described in Section 4, including parallelization where applicable. In addition, our tool supports exporting input to Algorithm 1, CASA, NIST ACTS (IPOG) and MoSo-PoLiTe, and then invoking the tools to generate covering arrays. The results were interpreted by our tool and exported as a Comma Separated Values (CSV) file.

[6]The experiment was performed on a machine with four Intel Xeon CPU L7555 @1.87GHz with 8 cores with hyper threading, 128 GiB RAM. The activity on the machine was limited to 6 threads at 100% activity and 32 GiB of RAM. All the measurements have been done with open and freely available software.

[7]The FreeBSD feature model is partial.

[8]All the measurements are available online at `http://heim.ifi.uio.no/martifag/splc2012/`.

[9]The tool is available at `http://heim.ifi.uio.no/martifag/splc2012/`, implemented in Java.

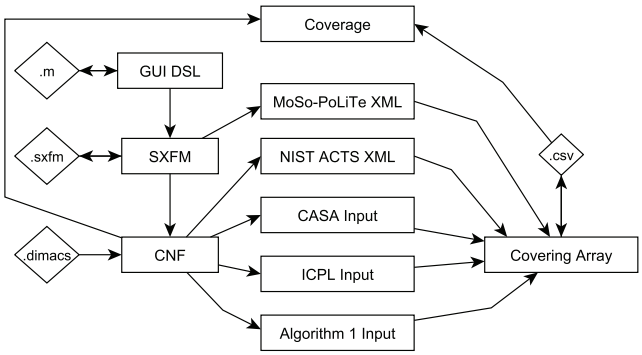

Figure 2: Transformations Used in the Experiment

Figure 2 shows the overview of the tool. The diagram 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.

Since our model collection consists of feature models stored in three different formats, the tool accepts feature models in these 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).

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[10] 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 Algorithm 1 and ICPL, which works with feature models in the CNF format.

The three other tools that implement the algorithms with which ICPL is compared are available from their respective providers[11].

The covering arrays are written to a Comma Separated Values (CSV) file. The covering arrays can then be used to configure products for which single system testing is applied.

5.2 Verification of Covering Arrays

Our tool can measure the coverage of the covering arrays. It was ensured that each tool generated a covering array with 100% coverage by using a separate algorithm independent of any of the compared algorithms. The configurations in the covering arrays were also separately verified to be valid configurations. These two features are implemented in the tool used in the experiment and are available as a part of it.

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

[11] MoSo-PoLiTe is a part of pure::variants provided by Pure Systems. CASA is available online at `http://cse.unl.edu/citportal/tools/casa/`. NIST ACTS is available online at `http://csrc.nist.gov/`

5.3 Generating Covering Arrays from Large Feature Models

Starting from the smallest feature model, we gave each algorithm 12 hours to complete the generation of a covering array[12].

ICPL is the only algorithm in our comparison that manages to generate a 1-wise covering array from the Linux kernel feature model (called `2.6.28.6-icse11.dimacs`), a 2-wise covering array from the three largest feature models, and a 3-wise covering array containing only t-sets with included features for two of the largest feature models. The results are shown in Table 2.

In Table 2, the measurements that are only available in the experiment from ICPL are shown with a gray background[13]. Notice that a 2-wise covering array can now be generated from a feature model with almost 7,000 features and 200,000 constraints. This is a big leap up from generating covering arrays from a feature model with close to 300 features and only 22 constraints, which is the largest feature model that one or more of the other algorithms managed to produce a 2-wise covering array from.

5.4 Comparison of the Five Algorithms

Using our experimental setup discussed earlier, all five algorithms were run through the same 19 feature models 100 times. If the time taken to generate 100 covering arrays was longer than three full days, the results were not included in the comparison. Figure 3 shows what part of the feature model corpus from which 100 covering arrays could be generated correctly[14] within three full days[15].

5.4.1 Comparison of Time Performances

Figure 4 shows the statistically estimated time performances. The numbers shown are the x'es in $O(f^x)$. Thus, for generating a covering array for a feature model with 1,000 features, a difference of 1 in the estimated value of x means that it takes 1,000 times longer to generate the covering array. To make the comparison fair, the basis for the statistical estimates were covering array sizes from the same 16, 14 and 12 feature models were compared for 1, 2 and 3-wise, respectively.

The fastest algorithm for all three strengths of covering ar-

[12] The available version of MoSo-PoLiTe does not handle feature models with propositional constraints above a certain level of complexity. Thus, we could not use it on five of the feature models in our corpus, including the four largest ones.

[13] In Perrouin et al. 2011 [18], it was reported that MoSo-PoLiTe generated a 3-wise covering array from the feature model with 287 features. That covering array was of size 841. In our experiments, we observed longer execution times for MoSo-PoLiTe than what was reported in Perrouin et al. 2011 [18], but it produced similar covering array sizes. ICPL produces a 3-wise covering array of size 108, an improvement in covering array size with a factor of 7.8. It was determined in personal correspondence with the authors that the available version of MoSo-PoLiTe might be different than the one used to generate the results in Perrouin et al. 2011 [18]. For our study, we used the only version available to us and provided by the authors on January 25. 2012.

[14] For five of the models, measurements are unavailable for MoSo-PoLiTe due to the complexity of the constraints in them. IPOG produced erroneous results for two of the features models. These results were therefore not included.

[15] IPOG and MoSo-PoLiTe does not support generating 1-wise covering arrays.

Table 2: Performance of ICPL on Large Feature Models: Feature Models, Satisfiability Times, Covering Array Sizes and Generation Times.
***Covering arrays for t-sets containing with only included features (1/8th of the t-sets), the memory usage was limited to 128 GiB instead of 32 GiB, and the algorithm was allowed to run in 64 processors at 100% activity instead of 6.**

Feature Model\keys	Features	Constraints (CNF clauses)	SAT time (ms)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	125	25	89	480	33,702	n/a	n/a
freebsd-icse11.dimacs	1,396	17,352	18	9	10	77	240	*78	*2,540
ecos-icse11.dimacs	1,244	2,768	12	6	2	63	185	*64	*973
Eshop-fm.xml	287	22	5	3	0.16	21	5	108	457

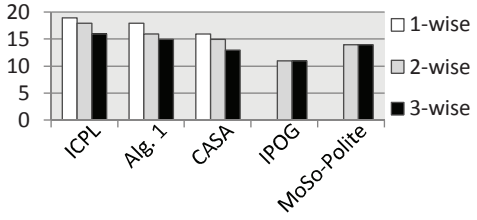

Figure 3: Out of the 19 feature models, how many of them the algorithms managed to correctly produce 100 covering arrays within three full days.

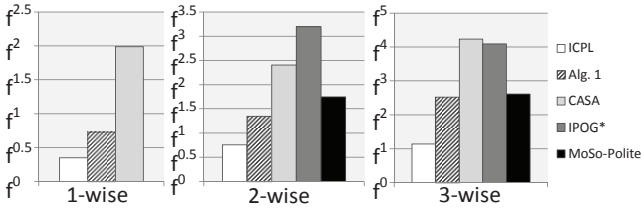

Figure 4: The statistically estimated values of x for the time taken for generating a covering array in the form $O(f^x)$ from a feature model with f Features. The times were for the same 16, 14 and 12 feature models for 1–3-wise covering arrays, respectively.
***The estimates for IPOG are extrapolated since it, as can be seen in Figure 3, managed considerably less feature models than the other algorithms.**

rays is clearly ICPL. Its statistically estimates performance is $O(f^{0.35})$, $O(f^{0.76})$ and $O(f^{1.14})$ for 1–3-wise covering array generation, respectively. Except for Algorithm 1 developed by us, the second fastest algorithm is MoSo-PoLiTe with a statistically estimated performance of $O(f^{1.75})$ and $O(f^{2.62})$ for 2 and 3-wise covering array generation.

5.4.2 Comparison of Size Performances

Figure 5 shows the sum of the sizes of covering arrays for the same 16, 14 and 12 feature models for 1, 2 and 3-wise, respectively. The sizes of 1–3-wise covering arrays are approximately the same for ICPL, Algorithm 1 and CASA. CASA is slightly better in this comparison. MoSo-PoLiTe produces larger covering arrays overall for both 2 and 3-wise covering arrays and IPOG for 3-wise covering arrays.

An interesting thing to notice here is that the size performance of ICPL and Algorithm 1 are almost identical. This is not a coincidence, as ICPL is Algorithm 1 with several speed optimizations.

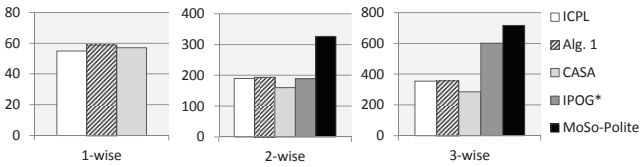

Figure 5: The sum of the sizes of the covering arrays for the same 16, 14 and 12 feature models for 1, 2 and 3-wise covering arrays, respectively.
***The estimates for IPOG are extrapolated since it, as can be seen in Figure 3, managed considerably less feature models than the other algorithms.**

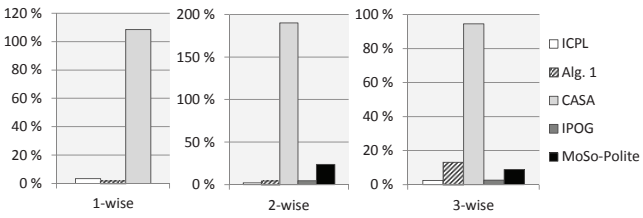

Figure 6: The sum of standard deviation of the 100 measurements of time over the sum of the average covering array generation times.

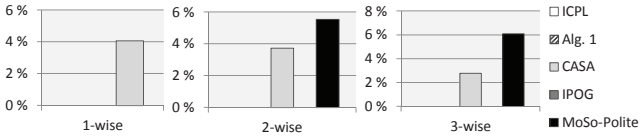

Figure 7: The sum of standard deviation of the 100 measurements of size over the sum of the average covering array sizes.

5.4.3 Effects of Non-Determinism

Each algorithm might have some degree of non-determinism. It was in order to identify such effects that 100 measurements were taken for each tool for each feature model. These measurements provide us with the standard deviation both in covering array sizes and in generation times. This is shown in Figure 6 and 7. Note that the standard deviations for ICPL and for IPOG are some places as low as 0, meaning that they run the same time each time they are run.

Only CASA shows a significant standard deviation for the time taken. A standard deviation of 100% means that the time taken to generate a covering array is often as long as

double that of the average time taken.

For the sizes of the covering arrays, the standard deviation is small for all algorithms.

6. THREATS TO VALIDITY

The authors of ICPL are also those who performed the comparison[16] with the other algorithms. Below we discuss things that might have influenced the result of the comparison.

Corpus Selection.

The feature model corpus consists of feature models not made by the authors of this paper. They are all used as originals except for some changes to the names due to differences in naming requirements in the different formats.

Corpus Completeness.

No feature models were excluded from the corpus except from their lack of realism. All feature models the authors were aware of and could verify the origin of were included in the corpus within the time available for conducting the experiment.

It is a possibility that the measurements are different given a more extensive corpus.

Adaptors.

Some of the tools came with limited documentation. The adaptors[17] that convert from the three formats of feature models in our corpus to input for each of the three other algorithms were written for the experiment in this paper and are available as open source. The results produced by each algorithm were confirmed to be complete and valid by the same, separate algorithms.

It is, however, possible that the adaptors are unfair to the other algorithms, and that the algorithms would have performed better with more suited converters. Had they been available from the implementations, they would have been used instead of the ones written by the authors of this paper.

Other Algorithms.

There are other algorithms which were not included in the comparison. The algorithm mAETG was neither available online nor by correspondence with the author. Some other algorithms which were not considered due to time constraints are the ATGT algorithm described in Calvagna and Gargantini 2009 [1], Microsoft's PICT Tool, and the algorithm in Grieskamp et al. 2009 [8] which is integrated into Microsoft's Spec Explorer for Visual Studio.

Other Strengths.

Empirics were not collected for $t >= 4$. One of the reasons for this is that the sizes of the covering arrays, using any algorithm, then becomes significantly larger than the number of features in the feature model from which it is generated. A second reason is that empirics indicate that 95% of bugs can be attributed to the interaction of 3 features [15]. If the quality requirements are above this level, it is not yet established that combinatorial interaction testing is the approach that should be taken in the first place.

[16] A complete, functional implementation of ICPL is provided as open source along with the scripts used to automatically accumulate the results presented. This makes it possible to completely reproduce all results reported in this paper. Also, all the measurements done in our experiment are available online. Implementations of the other algorithms are available from the respective sources.

[17] The adaptors are available as open source from `http://heim.ifi.uio.no/martifag/splc2012/`.

7. FUTURE WORK

Make a Specialized Feature Model Satisfier.

Since feature models are easy to configure but might be hard to convert to CNF [24], a specialized configurator for feature models that exploits the specific properties of feature models to satisfy it without converting it to a CNF formula would solve the problem of not converting the formula to CNF.

Find Additional Improvements.

We have not ruled out further improvements upon ICPL. ICPL is made available as open source so that it can be used for further research.

Finding a way to parallelize the part of ICPL which is not parallelized would increase the speed further.

Find a More Efficient t-set Storage.

A clear improvement to ICPL is more effectively storing the t-sets. Storing the t-sets is the cause of the high memory requirement of the algorithm.

8. CONCLUSION

In this paper, an algorithm for t-wise covering arrays, called ICPL, was presented. ICPL handles some of the largest feature models available, produces covering arrays of an acceptable size and has low standard deviation due to its non-determinism.

ICPL was compared to three other algorithms that handle the same problem. Within 12 hours, ICPL was the only one to handle the three largest feature models for 2-wise covering array generation. Within 3 whole days, each algorithm was run 100 times for each feature model. ICPL has the highest scalability overall for 1–3-wise covering array generation and generates covering array quickly. No variation in covering array size was observed, and the time taken varies insignificantly. Thus ICPL enables the application of combinatorial interaction testing to significantly larger software product lines than before.

9. ACKNOWLEDGMENTS

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

The authors would like to thank Ina Schaefer and the anonymous reviewers for their helpful feedback.

The authors would also like to thank Daniel Le Berre for his assistance on SAT4J, Brady J. Garvin for answering questions related to CASA and mAETG, and Sebastian Oster for answering questions related to MoSo-PoLiTe.

10. REFERENCES

[1] A. Calvagna and A. Gargantini. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In C. Dubois, editor, *Tests and Proofs*, volume 5668 of *Lecture Notes in Computer Science*, pages 27–42. Springer Berlin / Heidelberg, 2009.

[2] V. Chvátal. A greedy heuristic for the Set-Covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[3] M. B. Cohen, M. B. Dwyer, and J. Shi. 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.

[4] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[5] E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Information and Software Technology*, 53(1):2 – 13, 2011.

[6] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 177–188, New York, NY, USA, 2009. ACM.

[7] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Softw. Engg.*, 16:61–102, February 2011.

[8] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. Cohen. Interaction coverage meets path coverage by smt constraint solving. In M. Núñez, P. Baker, and M. Merayo, editors, *Testing of Software and Communication Systems*, volume 5826 of *Lecture Notes in Computer Science*, pages 97–112. Springer Berlin / Heidelberg, 2009.

[9] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, Dec. 1986.

[10] M. Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, 2010.

[11] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In J. Whittle, T. Clark, and T. Kuehne, editors, *Proceedings of Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011*, pages 638–652, Wellington, New Zealand, October 2011. Springer, Heidelberg.

[12] M. F. Johansen, Ø. Haugen, and F. Fleurey. A survey of empirics of strategies for software product line testing. In L. O'Conner, editor, *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11*, pages 266–269, Washington, DC, USA, 2011. IEEE Computer Society.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[14] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[16] Y. Lei, R. Kacker, D. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, pages 549–556. IEEE, 2007.

[17] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In J. Bosch and J. Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2010.

[18] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 2011.

[19] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[20] A. Reuys, S. Reis, E. Kamsties, and K. Pohl. The scented method for testing software product lines. In T. Käkölä and J. C. Dueñas, editors, *Software Product Lines*, pages 479–520. Springer, Berlin, Heidelberg, 2006.

[21] J. Rivieres and W. Beaton. Eclipse Platform Technical Overview, 2006.

[22] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 461–470. ACM, 2011.

[23] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309 –322, may-june 2010.

[24] T. Walsh. Sat v csp. In R. Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer Berlin / Heidelberg, 2000.