Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays

Evelyn Nicole Haslinger
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
evelyn.haslinger@jku.at

Roberto E. Lopez-Herrejon
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
roberto.lopez@jku.at

Alexander Egyed
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Combinatorial Interaction Testing has shown great potential for effectively testing Software Product Lines (SPLs). An important part of this type of testing is determining a subset of SPL products in which interaction errors are more likely to occur. Such sets of products are obtained by computing a so called t-wise Covering Array (tCA), whose computation is known to be NP-complete. Recently, the ICPL algorithm has been proposed to compute these covering arrays. In this research-in-progress paper, we propose a set of rules that exploit basic feature model knowledge to reduce the number of elements (i.e. t-sets) required by ICPL without weakening the strength of the generated arrays. We carried out a comparison of runtime performance that shows a significant reduction of the needed execution time for the majority of our SPL case studies.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Performance, Theory

Keywords

product lines, combinatorial testing, pairwise testing, evaluation, feature model-based testing.

1. INTRODUCTION

In Software Product Line Engineering (SPLE) families of systems are designed, rather than developing the individual products separately [3, 6, 21]. The focus of SPLE lies on software reuse, meaning that the individual products of a Software Product Line (SPL) usually share a considerable amount of source code. These products are distinguished by the features they implement, where a feature is often defined as an increment in program functionality [25].

Testing is a vital part of the development cycle of any software system. However, because an SPL is a collection of systems (frequently with a large number of members) special considerations should be taken into account. Obviously each product of the SPL could be tested using single system testing techniques. This approach though, is usually not feasible or too expensive because the number of different products can be very large. Furthermore, such approach can result in redundant test effort as the products within an SPL usually share some of the functionalities they are offering. Different techniques have been proposed to test the products within an SPL more efficiently [8,17,19,22,24]. Incremental testing attempts to automatically adapt the available test cases using the knowledge about commonalities and differences among the member products of the SPL [24]. In combinatorial interaction testing a representative subset of the products within the SPL is selected for testing [5].

This research-in-progress paper proposes two reduction rules that can be used to enhance algorithms that extract such subsets of products (i.e. so called *t-wise Covering Arrays* (tCAs)) for combinatorial interaction testing. These rules reduce the number of feature combinations from which the representative product subset is selected without a negative impact of the test quality. We evaluated our approach in 146 feature models, of 9 to 172 number of features, and obtained a median execution time speedup of up to 88%.

2. BACKGROUND

This section outlines the motivation of combinatorial interaction testing, variability modeling with feature models in the context of combinatorial interaction testing and the existing related work.

2.1 Feature Models

Feature Models (FMs) are the de-facto standard to model variability in SPLE [6, 14], where they are used to express which feature combinations are considered valid products. An example of an FM of an SPL for printers is shown in Figure 1. While the number of possible product configurations is exponential in the number of features not all of these products are valid. For instance, in our running example the feature "print sepia" may only be available if also feature "support colored prints" is selected.

VaMoS '13, January 23–25 2013, Pisa, Italy

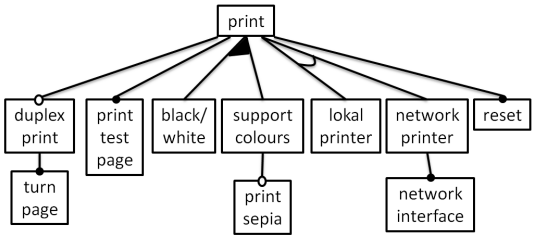

Figure 1: Feature Model of Printer SPL

FMs are tree-like structures in which features are depicted using labeled boxes. An FM has exactly one root feature, i.e. feature `print` in our running example, that is selected in every valid products configuration. Each feature may have a set of child features with which it can interrelate in four different types of relationships:

- *mandatory child features*: these features have to be selected whenever their parent is selected (e.g. feature `test page` in the printer SPL).
- *optional child features*: these features may or may not be selected if their parent is selected (e.g. feature `duplex print` in the printer SPL).
- *inclusive-or relations*: for these features holds that at least one feature of the group needs to be selected if their parent is selected (e.g. features `back-white` and `support colors` form an inclusive-or relation in the printer SPL).
- *exclusive-or relations*: for these features holds that exactly one feature of the group needs to be selected if their parent is selected (e.g. features `local printer` and `network printer` form an exclusive-or relation in the printer SPL).

Apart from these child-parent relations there are also so called *Cross Tree Constraints (CTCs)*. These constraints are denoted using propositional logic formulas. CTCs are used to capture constraints that are not covered by the FM tree.

2.2 Combinatorial Interaction Testing

Combinatorial interaction testing aims to tackle some of the problems imposed by testing SPLs, namely the large number of products that need to be tested and the resulting redundant test effort. The basic idea of combinatorial interaction testing is to select a representative subset of products where interaction errors are more likely to occur [5], rather than testing the complete product family. There are three steps to apply combinatorial interaction testing:

1. Choose a subset of the products within the SPL.
2. Configure and implement these products.
3. Apply single system testing on the set of selected products.

This research-in-progress paper focuses on step one of combinatorial interaction testing, meaning to select a proper subset of products on which single system testing is applied. An approach to select the products for combinatorial interaction testing are so called *t-wise covering arrays (tCAs)*. As mentioned earlier the products within an SPL are distinguished by the set of features they implement, formally they can be represented using feature sets.

Definition 1. *Feature List (FL) is the list of features in a software product line.*

For our running example the feature list is `FL=[print, print duplex, turn page, print test page, black/white, support colors, print sepia, local printer, network printer, network interface, reset]`.

Definition 2. *Feature Set (FS) is a 2-tuple [sel,$\overline{sel}$] where sel and $\overline{sel}$ are respectively the set of selected and not-selected features of a member product. Let FL be a feature list, thus sel, $\overline{sel} \subseteq FL$, $sel \cap \overline{sel} = \emptyset$, and $sel \cup \overline{sel} = FL$. The terms p.sel and p.$\overline{sel}$ respectively refer to the set of selected and not-selected features of product p*[1].

An example of a feature set for our printer SPL is `ts`$_1$`=[{print, print test page, black/white, local printer, reset}, {duplex print, turn page, support colors, print sepia, network printer, network interface}]`.

Definition 3. *A t-set ts is a is a 2-tuple [sel,$\overline{sel}$] representing a partially configured product, defining the selection of t features of feature list FL, i.e.* $ts.sel \cup ts.\overline{sel} \subseteq FL \wedge ts.sel \cap ts.\overline{sel} = \emptyset \wedge |ts.sel \cup ts.\overline{sel}| = t$. *We say t-set ts is covered by feature set fs iff* $ts.sel \subseteq fs.sel \wedge ts.\overline{sel} \subseteq fs.\overline{sel}$.

Definition 4. *We say a feature set `fs` is valid in feature model `fm`, i.e. `valid(fs, fm)` holds, iff `fs` does not contradict any of the constraints introduced by `fm`*[2]. *A t-set `ts` is valid if there exists a valid feature set `fs` that covers `ts`.*

The feature set `ts`$_1$ mentioned above is an example of a valid feature set of the printer FM. The 2-set [{local printer, network printer}, {}] is an example of an invalid t-set, because the two features `local printer` and `network printer` cannot be selected together in the same product.

Formally we can now define a t-wise covering array as:

Definition 5. *A t-wise covering array tCA for a feature model FM is a set of valid feature sets that covers all valid t-sets of the software product line.*[3]

If we use a 1-wise CA to select the subset of products to test we make sure to verify whether a product behaves faulty if a certain feature is included or not included. For 2-wise CAs we make sure to test whether all possible interactions among two features `A` and `B` behave erroneously. A study by Kuhn et. al [16] has shown that:

- When using 1-wise CAs it is likely to detect 50% of the defects.
- When using 2-wise CAs it is likely to detect 70% of the defects.

[1]Definition based on [4].
[2]Such constraints are introduced by child parent relationships and additional CTC that are expressed as propositional logic formulas and are obtained as described in [4].
[3]Definition inspired by [13].

- When using 3-wise CAs it is likely to detect 95% of the defects.

The generation of t-wise covering arrays is an instance of the set-cover problem, which has been proven to be NP complete [15], meaning that no efficient algorithms are known to generate minimal t-wise covering arrays.

3. REDUCTION RULES FOR T-SETS

This section describes a set of reduction rules that can be used to speed up the generation of t-wise CAs.

3.1 Generating t-wise CAs as an Instance of the Set-Cover Problem

The generation of t-wise CAs is an instance of the set-cover problem [13]. The set-cover problem states that there is some universe `U` that contains all the elements that need to be covered, then there is a set `S` whose elements are subsets of `U`. An optimal solution to the set-cover problem is a minimal subset `S'` of `S` for which holds $\bigcup$`S' = U` [2].

Let `FM` be a feature model describing the valid combinations among features in feature list `FL`, let `products` be the set of valid feature sets in `FM` and let `V` be the set of all valid t-sets in `FM`. Then `V` and `products` can be mapped as follows to the set-cover problem:

- The set of valid t-sets `V` is the universe `U` of the set-cover problem, i.e. the set of elements that need to be covered.
- The set of valid feature sets `products` corresponds to the set `S`, where each feature set in `products` covers a subset of t-sets in `V`.
- The minimal subset `S'` of `S` is then a minimal t-wise covering array.

Every algorithm that generates a t-wise CA for a given feature model `FM` needs to resolve two problems:

1. Determine the universe `U` of the set-cover problem, i.e. set of valid t-sets `V` in `FM`.
2. Select a subset `S'` of `S` that covers the universe `U`, i.e. select a subset of the set of valid feature sets `products`.

To determine the set of valid t-sets `V`, all t-sets that can be formed for features in feature list `FL` need to be checked whether they are valid in feature model `FM`. The number of possible t-sets is `A*B`, where $A = \frac{|FL|!}{t!*(|FL-t|!)}$ denotes the number of different subsets of size `t` that can be formed by the elements in `FL` and $B = 2^t$ represents the number of possible combinations among `t` features.

The universe `U` (i.e. the set of all valid t-sets `V`) of this specific instance of the set-cover problem tends to grow fast in the number of features in feature list `FL`. In this paper, we propose a set of reduction rules for `V` that compute a subset `core` $\subset$ `V`, for which holds that any t-wise CA for `core` will also cover `V`. In other words, rather than suggesting an improvement to the procedure used to select a subset of `products`, we propose to reduce the size of the problem input `V`, while still guaranteeing the t-wise array coverage.

The following sections describe the two reduction rules and analyze their rationale.

3.2 t-sets Containing the Root Feature

Our first reduction rule is concerned with the root feature of a feature model `FM`. By definition any valid feature set of `FM` needs to select the root feature, having as a result that each t-set `ts` that deselects the root feature `r`, in other words, `r` $\in$ `fs.` $\overline{\texttt{sel}}$, is invalid.

If the feature list `FL` of feature model `FM` contains more than `t` features (i.e. `|FL|>t`), then all t-sets containing the root feature `r` can be safely removed from the set of all valid t-sets `V` without weakening the generated tCA. The condition `|FL|>t` is needed because to generate a t-wise coverage at least `t` features are necessary. If removing the t-sets containing the root feature `r` reduces `V` to the empty set, then t-wise coverage could not be guaranteed.

Let us now more formally describe this rule. Let t_r be a valid t-set where root feature `r` is selected. If `|FL|>t` holds, then there exists a valid t-set `t'` that selects and deselects the same features as t_r, except that instead of root feature `r` an additional feature `f` is element of either the `sel` or $\overline{\texttt{sel}}$ set of `t'`, i.e. $t_r.\texttt{sel}\backslash\{r\} \subseteq t'.\texttt{sel} \wedge t_r.\overline{\texttt{sel}} \subseteq t'.\overline{\texttt{sel}}$ and $f \in t'.\texttt{sel} \vee f \in t'.\overline{\texttt{sel}}$.

Let `fs` be a valid feature set that covers `t'`, then `fs` also covers t_r:

- Because `fs` covers `t'` the condition $t'.\texttt{sel} \subseteq fs.\texttt{sel} \wedge t'.\overline{\texttt{sel}} \subseteq fs.\overline{\texttt{sel}}$ holds.
- Because the root feature `r` has to be included in every valid feature set the condition $r \in fs.\texttt{sel}$ holds.
- As we know that $t_r.\texttt{sel}\backslash\{r\} \subseteq t'.\texttt{sel} \subseteq fs.\texttt{sel} \wedge t_r.\overline{\texttt{sel}} \subseteq t'.\overline{\texttt{sel}} \subseteq fs.\overline{\texttt{sel}}$ and $r \in fs.\texttt{sel}$ we can derive that `fs` also covers t_r.

For example t_r could be t_r=[{print, black}, {support}] then `t'` could be `t'`=[{black, duplex}, {support}]. A valid feature set `fs` for the feature model shown in Figure 1 that covers `t'` is:
`fs=[{print, print test page, reset, duplex, turn page, black/white, local}, {support, network printer, network interface}]`

Please notice that `fs` does not only cover `t'` but also t_r. In other words, this reduction rule does not weaken the t-wise coverage.

3.3 t-sets Containing Mandatory Features

Our second reduction rule is concerned with mandatory child features. A mandatory child feature has to be selected whenever their parent is selected, and they must not be selected when their parent is not selected. Hence, for every valid feature set `fs` holds that a feature `f` and its mandatory child feature f_{mand} are either both selected or deselected, i.e. $\{f, f_{mand}\} \subseteq fs.\texttt{sel} \vee \{f, f_{mand}\} \subseteq fs.\overline{\texttt{sel}}$. If the feature list `FL` of feature model `FM` contains more than `t` features that are neither the root feature nor mandatory child features, then all t-sets containing a mandatory child feature f_{mand} can be safely removed from the set of all valid t-sets `V` without weakening the generated tCA.

Let t_f be a t-set containing a feature `f` that has a mandatory child f_{mand} feature in FM and let t_{mand} be a t-set that contains the same sets of selected and deselected features as t_f, except that feature `f` is replaced by f_{mand}, i.e.

- If $f \in t_f.\texttt{sel}$ then $t_{mand} = [t_f.\texttt{sel}\backslash\{f\} \cup \{f_{mand}\}, t_f.\overline{\texttt{sel}}]$.

- If $f \in t_f.\overline{sel}$ then $t_{mand} = [t_f.sel, t_f.\overline{sel} \setminus \{f\} \cup \{f_{mand}\}]$.

Let fs be a valid feature set that covers t_f, then fs also covers f_{mand}:

- Because fs covers t_f the condition $t_f.sel \subseteq fs.sel \wedge t_f.\overline{sel} \subseteq fs.\overline{sel}$ holds.
- If $f \in fs.sel$ holds then we know that $t_{mand}.\overline{sel}=t_f.\overline{sel} \subseteq fs.\overline{sel}$ and $t_{mand}.sel \setminus \{f_{mand}\} \subseteq fs.sel$, as each valid feature set that selects a feature f has also to include its mandatory child f_{mand} we now that fs covers t_{mand}.
- If $f \in fs.\overline{sel}$ holds then we know that $t_{mand}.sel=t_f.sel \subseteq fs.sel$ and $t_{mand}.\overline{sel} \setminus \{f_{mand}\} \subseteq fs.sel$, as each valid feature set that deselects a feature f has also to exclude its mandatory child f_{mand} we now that fs covers t_{mand}.

For example t_f and t_{mand} could be:
t_f=[{network printer, print sepia}, {duplex print}]
t_{mand}=[{network interface, print sepia}, {duplex print}]. Consider Figure 1 to determine a feature set fs that covers t_f. An example of a valid feature set fs covering t_f is:
fs = [{print, print testpage, reset, black/white, support colors, print sepia, network printer, network interface}, {local printer, duplex print, turn page}].
Please also notice that fs does not only cover t_f but also t_{mand}.

4. EVALUATION

To evaluate the reduction rules presented in this paper we adapted the ICPL algorithm. The following sections provide more details on this algorithm, the set up of our evaluation and present the results we obtained.

4.1 The ICPL Algorithm

ICPL is a tCA generation algorithm proposed by Johansen et al. that has been recently published in [13]. They performed a comparison with three other tCA generation algorithms (i.e. CASA [11], IPOG [18] and MoSo-PoLiTe [20]), showing that ICPL overall has a better runtime performance and also tends to generate smaller tCAs. This is the main reason why we chose to use ICPL for our evaluation.

ICPL is an adaption of Chvátal's greedy approach to solve the set-cover problem. They use the fact that a (t-1)-wise CA is always a subset of the t-wise CA [10] to recursively build up a tCA. Moreover they parallelize ICPL to improve its performance.

In very simplified terms ICPL performs three steps:

1. If the tail of the recursion is not reached (i.e. t=1) then call ICPL recursively with t-1.
2. Generate the set of all t-sets.
3. Remove t-sets that are invalid in FM and extract the t-wise CA using the results of call t-1.

We enhance ICPL by applying our reduction rule in step 2. Instead of returning all possible t-sets that can be formed by features in feature list FL, we return only the t-sets that can be formed by those features in FL that are neither a mandatory child feature nor the root node of the FM.

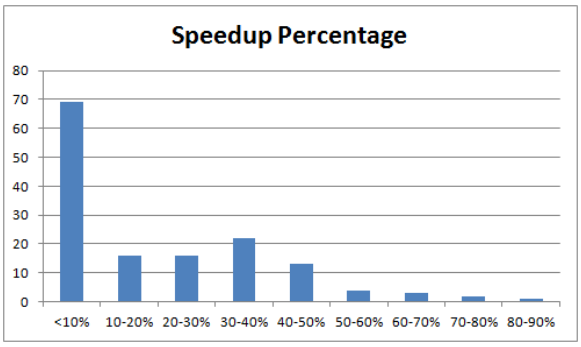

Figure 2: Histogram of obtained Speedup in %

4.2 Results Obtained

We performed our evaluation using a Windows 7 System running at 3.2Ghz on an Intel Core i5, and with RAM of 8GB. We used 146 publicly available FMs, where 15[4] are from the resources homepage of [13] and the others where obtained from the SPLOT website [1]. The number of features of these models range from 9 to 172. For each of these models we ran ICPL and our adapted version of ICPL a hundred times to generate 3-wise CAs. Where the 3-wise coverage is the highest one that is supported by the implementation provided by Johansen et. al.

The median execution time of ICPL is between 16ms and 9 minutes. We obtained a median speedup of 13% using our reduction rules and a maximal speedup of 88%. For 2 of the 146 feature models our adaptations lead to a slower median execution time. Figure 2 summarizes the speedups obtained in a histogram. For 69 models the recorded speedup is less than 10% but note that most of these models (i.e. 66) are rather small with only 20 or less features. ICPL needs time to read in the feature model, initialize variables, synchronize threads etc. We believe that ICPL spends for small models most of its execution time on these processing steps, therefore our reduction rules cannot gain more speedup.

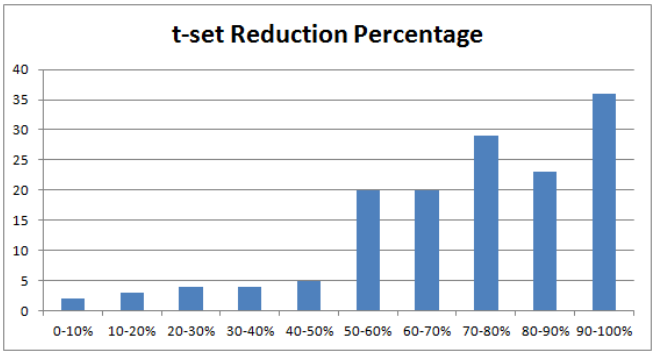

Figure 3: Histogram of the Reduction of Possible t-sets in %

Consider now Table 1. It summarizes the recorded results of the 12 largest feature models used for our evaluation. The column labeled with *#f* corresponds to the number of

[4]Notice that we did not include the four largest models in our evaluation because ICPL requires a large cluster of at least 8 Cores and 128GiB of RAM for its execution.

Feature Model	#f	#i	Original [ms]	Modified [ms]	Speedup [%]	t-set [%]	tCA [%]
Arcade Game	61	19	1187.0	811.0	32	68	6
Reuso-UFRJ-Eclipse1	72	18	3854.0	1811.0	53	58	1
J2EE web architecture	77	28	2669.0	1210.0	55	75	1
Berkeley	78	19	8713.5	3246.0	63	57	3
Billing	88	12	2091.0	1576.0	25	36	0
Model Transformation	88	18	6186.5	3828.5	38	50	-2
Coche ecologico	94	40	9010.0	5227.0	42	81	-1
UP estructural	97	17	10983.0	9057.0	18	44	2
Violet	101	3	48915.0	50233.0	-3	9	-6
xtext	137	42	44055.5	11646.5	74	67	6
Test	168	52	545089.5	64827.5	88	67	-1
Printers	172	49	180891.0	155658.0	14	64	1

Table 1: Recorded Results of Largest Models

features of the FM and $\#i$ represents the number of features that could be ignored for the tCA generation, i.e. the root feature plus the number of features that are in a mandatory relation with their parent. The column labeled with *t-set [%]* contains the number of reduced possible t-sets and the column labeled with *tCA [%]* contains the change in size of the generated tCAs.

Among the twelve largest feature models the *Violet FM* is the only one for which we did not gain a speedup, because only 3 out of 101 features could be skipped during the tCA generation. Thus it is not surprising that our reduction rules do not have much effect. The most speedup was recorded for the *Test FM*. The original version of ICPL terminated after about 9 minutes, while the version of ICPL using our reduction rules needed only 1 minute to generate the tCA.

The percentage of reduced possible t-sets is depicted in Figure 3. For a majority of feature models more than 50% of all possible t-sets do not need to be considered during the tCA generation. For only 5 models less than 20% can be reduced.

Finally a comparison of the generated tCAs is shown in Figure 4. Our reduction rules do not tend to lead to smaller or bigger tCAs in general. For 47 of 146 FMs we produced on average smaller tCAs and for 55 the average size of the produced tCAs were larger. The average difference of the size of the generated arrays over all runs for all models is 0.

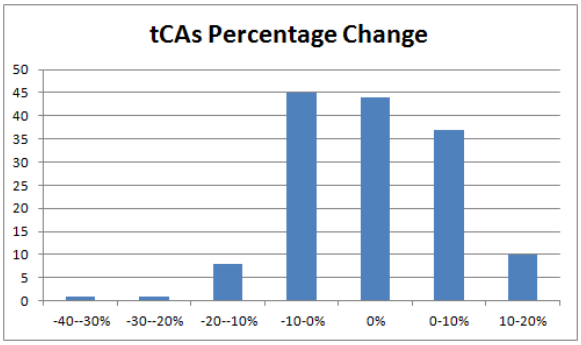

Figure 4: Histogram of the Change in Size of the tCAs %

5. RELATED WORK

Combinatorial Testing has shown great potential to detect interaction faults. Different techniques have been applied to generate test suites, such as greedy algorithms, search based techniques and mathematical methods. Many of the proposed test generation algorithms do not support constraints and simply ignore them [19]. The remainder of this section describes three t-wise CA generating algorithms that do support constraints and presents work by Segura that is similar to our second reduction rule.

Johansen et. al present the ICPL Algorithm [13] that applies a greedy approach to generate tCAs. It is an extension to the algorithm they introduced in [12], with several enhancements made. They use for instance the fact that some of their procedures are data independent, hence they parallelized as many steps as possible. Moreover they use the fact that a (t-1)-wise CA is always the subset of a t-wise CA [10] to apply recursion. This recursive approach enables them for instance to speed up the detection of invalid t-sets. They evaluated their approach using 19 feature models. For the three largest models, having between one thousand and seven thousand products, they were not able to generate the 3-wise CAs[5].

Garvin et al propose CASA [11], where they use a meta-heuristic search technique, simulated annealing, to generate tCAs. They compared different versions of their implementation with a greedy algorithm showing that while the simulated annealing approach has a longer run time, the generated tCAs tend to be on average 25% smaller.

Oster et. al present the Model-based Software Product Line Testing framework (*MoSo-PoLiTe*) [20]. Their framework can be used to generate CAs for pairwise testing. First they flatten the FM to convert it into a constraint solving problem (*CSP*). Next they apply a subset extraction algorithm, that uses forward checking solving on the CSP corresponding to the input FM, to generate the 2-wise CA. It is part of their future work to also evaluate MoSo-PoLiTe on 3-wise and 4-wise CAs.

Segura use in [23] the concept of atomic sets to simplify FMs, i.e. features that form an atomic set are united to a single node in the FM tree. Their experiments show improvements in runtime and memory usage when some of the operations of the FAMA tool are performed. This FM simplification is similar to our second reduction rule, as it elimi-

[5]For two of these models they generated an array that covers 1/8th of the 3-sets, which is in our opinion no 3-wise CA.

nates all mandatory child features in the FM tree. However, our reduction rules do not alter the used FM, they rather determine which features need to be considered to determine the set of valid t-sets.

6. CONCLUSIONS AND FUTURE WORK

In this research-in-progress paper we propose a set of reduction rules to reduce the number of t-sets that have to be covered by any t-wise CA generation algorithm, without weakening the strength of the generated arrays. We evaluated these rules by adapting ICPL [13], where we recorded speedups of up to 88% in median execution time.

Our reduction rules are not only applicable to ICPL, but to any tCA generation algorithm. Hence, we plan to evaluate our rules also on CASA [11] and MoSo-PoLiTe [20]. We also plan to explore Search Based algorithms such as that presented by Ferrer et al. [9].

Additionally we have one more reduction rule in mind, which claims that any t-set that contains a feature `f` and one of its descendants does not need to be covered. It is part of our future work to formally proof this claim and to extend our current evaluation by also considering this rule.

Acknowledgment

This research is partially funded by the Austrian Science Fund (FWF) project P21321-N15 and Lise Meitner Fellowship M1421-N15.

7. REFERENCES

[1] Software Product Line Online Tools(SPLOT), Accesed July 2011. http://www.splot-research.org/.

[2] N. Alon, D. Moshkovitz, and S. Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms*, 2(2):153–177, 2006.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, In Press, Corrected Proof:–, 2010.

[5] M. Cohen, M. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on*, 34(5):633 –650, sept.-oct. 2008.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] E. S. de Almeida, C. Schwanninger, and D. Benavides, editors. *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*. ACM, 2012.

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

[9] J. Ferrer, P. M. Kruse, J. F. Chicano, and E. Alba. Evolutionary algorithm for prioritized pairwise test data generation. In T. Soule and J. H. Moore, editors, *GECCO*, pages 1213–1220. ACM, 2012.

[10] S. Fouché, M. B. Cohen, and A. A. Porter. Incremental covering array failure characterization in large configuration spaces. In G. Rothermel and L. K. Dillon, editors, *ISSTA*, pages 177–188. ACM, 2009.

[11] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[12] 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. Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 638–652. Springer, 2011.

[13] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In de Almeida et al. [7], pages 46–55.

[14] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[15] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[16] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, June 2004.

[17] J. Lee, S. Kang, and D. Lee. A survey on software product line testing. In de Almeida et al. [7], pages 31–40.

[18] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *ECBS*, pages 549–556. IEEE Computer Society, 2007.

[19] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, Feb. 2011.

[20] 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.

[21] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[22] 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, 2006.

[23] S. Segura. Automated analysis of feature models using atomic sets. In S. Thiel and K. Pohl, editors, *SPLC (2)*, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[24] E. Uzuncaova, S. Khurshid, and D. S. Batory. Incremental test generation for software product lines. *IEEE Trans. Software Eng.*, 36(3):309–322, 2010.

[25] P. Zave. Faq sheet on feature interaction. http://www.research.att.com/ pamela/faq.html.