

Combinatorial Testing for Feature Models Using CITLAB

Andrea Calvagna
Dip. di Matematica e Informatica
University of Catania - Italy
Email: calvagna@cs.unict.it

Angelo Gargantini
Dip. di Ingegneria
University of Bergamo - Italy
Email: angelo.gargantini@unibg.it

Paolo Vavassori
Dip. di Ingegneria
University of Bergamo - Italy
Email: paolo.vavassori@unibg.it

Abstract—Feature models are commonly used to represent product lines and systems with a set of features interrelated each others. Test generation from feature models, i.e. generating a valid and representative subset of all the possible product configurations, is still an open challenge. A common approach is to build combinatorial interaction test suites, for instance achieving pairwise coverage among the features. In this paper we show how standard feature models can be translated to combinatorial interaction models in our framework CITLAB, with all the advantages of having a combinatorial testing environment (in terms of a clear semantics, editing facilities, language for seeds and test goals, and generation algorithms). We present our translation which gives a precise semantics to feature models and it tries to minimize the number of parameter and constraints while preserving the original semantics of the feature model. Experiments show the advantages of our approach.

I. INTRODUCTION

Feature models (FMs) allow designers to specify families of products, generally called Software Product Lines (SPLs) in a simple way. A feature model lists the features in a product line together with their possible values and constraints. In this way, it can represent in a compact and easily manageable way millions of variants, each representing a possible product. There exist several tools for designing feature models, like FeatureIDE [19], and repositories and libraries like SPLOT [13]. However, only few tools support feature model testing, also because exhaustive FM testing is unfeasible in most cases.

A possible solution is to apply basic principles of combinatorial interaction testing (CIT) to feature models. Generally combinatorial testing algorithms and tools are adapted to feature models, like PACOGEN [10] and ICPL [11], or testing criteria similar to those for combinatorial testing are applied to SPLs [6]. All these works share the common approach that consists in adapting combinatorial algorithms to feature models testing. In this paper, we try to proceed in the other direction: translating feature models to combinatorial problems. In this way, thanks also to our framework for combinatorial testing, CITLAB [4], [8], we have access to all the features, algorithms and techniques normally used in combinatorial testing. We optimize our approach over the classical one, in several directions:

- 1) We use enumerative variables and types when it is more suitable than using only Boolean variables.

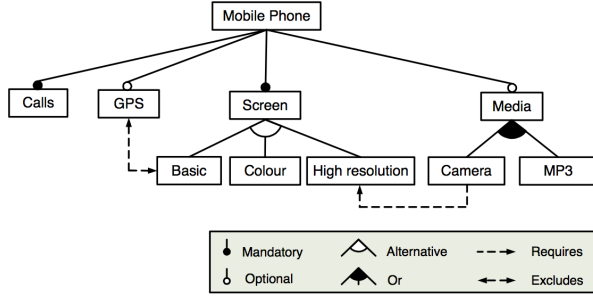
- 2) In this way we minimize the use of implicit constraints, i.e. those constraints which are not part of the feature models but are inserted only to correctly represent feature relationships.
- 3) We operate several simplifications in order to make models more easy to solve by combinatorial algorithms.

This strategy provides several benefits. At the model *representation* level, it allows us to generate a more human readable model and also prevents the use of the flattening algorithm [14], which disrupts the feature hierarchy. The resulting model is simpler both to read and understand (with less syntactical elements) and to deal with (with less constraints). At *generation* level, the tester has access to all the algorithms and tools commonly used for combinatorial interaction testing. Moreover, the model simplification could give access to tools and algorithms not suitable to the original models. For example, several combinatorial testing tools work only with models without constraints, and our approach is capable of removing all constraints in particular cases (e.g., when there is only one level in the feature model).

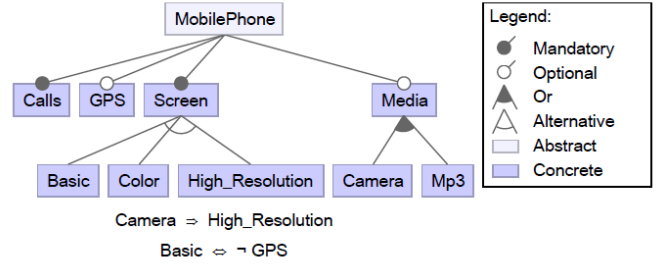
The paper is organized as follows. Section II presents the feature model notation together with the tools used in this paper (FeatureIDE and SPLOT) and our framework for combinatorial testing, CITLAB [8]. Section III presents our translation from FM to combinatorial models. In Section IV we present some simplifications we apply to the combinatorial models in order to further reduce their complexity by removing useless parameters and constraints while preserving their original semantics. Note that these simplifications are not specific and limited to combinatorial problems obtained from feature models, and they can be applied to any combinatorial problem regardless its origin. In Section V, we present the data we obtained using a wide set of examples commonly used as benchmarks in the SPL community in order to show how our techniques compare with other classical ones. Section VI presents some related work and Section VII concludes the paper and presents some future work.

II. BACKGROUND

In software product line engineering, feature models are a special type of information model representing all possible products of a software product line in terms of features and relationships among them. Specifically, a basic feature model



(a) Conventional notation



(b) FeatureIDE notation

Figure 1: Examples of Feature model notations

is a hierarchically arranged set of features, where each parent-child relation between them may be one of the following types:

Mandatory – child feature is required.

Optional – child feature is optional.

Or – at least one of the sub-features must be selected.

Alternative (xor) – exactly one of the sub-features must be selected

In addition to the parental relationships, cross-tree relations are allowed, to specify incompatibility or requirement kind of constraints between features, in the form:

A requires B – The selection of feature A in a product implies the selection of feature B.

A excludes B – A and B cannot be part of the same product.

Feature models can be visually represented by means of feature diagrams. Figure 1 depicts a simplified example model presented in [2] and inspired by the mobile phone industry, in order to present the visual notation commonly adopted for feature modeling. The example also shows how a model can be used to specify a product family, that is, to determine the features that will be supported (loaded) in a particular phone configuration of the considered family. According to the model, all phones must include support for calls, and displaying information in either a basic, color or high resolution screen. Furthermore, the software for mobile phones may optionally include support for GPS and multimedia devices such as camera, MP3 player or both of them.

Extensions to the basic feature model notation have been proposed in literature, e.g. allowing specifying the cardinality of the features and/or additional type of information. However, in this paper we consider only basic feature models.

A. Feature Modeling frameworks

Several languages/tools for specifying/analyzing feature models are currently available, some of them already mature enough to be part of a software production IDEs. In this work, FeatureIDE [19] has been used to design or import the models used in the evaluation section. FeatureIDE is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. FOSD is a paradigm for the construction, customization, and synthesis of software systems. Code artifacts are mapped to features, and a customized software

Type	Notation	Propositional formula
Mandatory		$p \rightarrow A \wedge A \rightarrow p$
Optional		$A \rightarrow p$
Alternative		$p \rightarrow alt(a1, a2 \dots an) \wedge a1 \rightarrow p \wedge a2 \rightarrow p \wedge \dots \wedge an \rightarrow p$
Or		$p \rightarrow (a1 \vee a2 \vee \dots \vee an) \wedge a1 \rightarrow p \wedge a2 \rightarrow p \wedge \dots \wedge an \rightarrow p$

Table I: Conventional translation

system can be generated given a selection of features. The set of software systems that can be generated is then a software product line (SPL).

B. Feature Model semantics

Feature models semantics can be rather simply expressed by using propositional logics as already done in [2]. Every feature becomes a propositional letter or a Boolean variable, and every relationship among features becomes a propositional formula modeling the constraints about them as reported in Tab. I¹.

Example 1. Note that in case of alternative features, the translation using Boolean variables introduces many variables which are mutually exclusive. For instance, the model in Fig. 2 introduces 8 Boolean variables for a system with just two main features (A and B), because each of them expands to four alternative leaf features $a_1 \dots a_4$ and $b_1 \dots b_4$. Several complex constraints are necessary to constraint valid products. In case of many alternative features, the complexity of the model would grow. Reducing their complexity would be a benefit to the user, which would be facilitated in the model

¹Where the *alt* operator represents the exclusive or among all its arguments and it is defined as $alt(a1, a2, \dots, an) = (a1 \wedge \neg a2 \wedge \dots \wedge \neg an) \vee \dots \vee (\neg a1 \wedge \neg a2 \wedge \dots \wedge an)$.

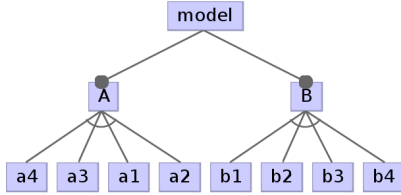


Figure 2: A two alternative model

comprehension, and would also allow the available tools to manage larger models.

Variability factor: This index measures the ratio between the number of valid products and the total number of possible feature combinations, which is equal to 2^n where n is the number of features. In case of Boolean variables, n is also equal to the number of variables. The feature model variability can be used to measure the flexibility of the feature model. For instance, a small factor means that the number of combinations of features is very limited compared to the total number of potential products [2].

C. CitLab

In this paper the CITLAB tool for Combinatorial Interaction Testing (CIT) is used to solve the combinatorial generation task associated with every feature model. The CITLAB tool allows importing/exporting models of combinatorial problems from/to different application domains, by means of a common interchange syntax notation and a corresponding interoperable semantic meta-model. Moreover, the tool is a framework allowing embedding and transparent invocation of multiple, different implementations of combinatorial algorithms. CITLAB has been designed tightly integrated with the Eclipse IDE framework, by means of its plug-in extension mechanism. It is intended to ease the spread of CIT testing both in industrial practice and in academic research, by allowing users and researchers to apply multiple test suite generation algorithms, each with its peculiarities, on the same problem models, and let them compare the results in order to select the one that best fits their needs, while alleviating from the pain of knowing all the different details and notations of the underlying CIT tools. In order to support the model conversion activity presented in this work, a new importer plugin for CITLAB has been designed and implemented, which converts a feature model specification into a corresponding combinatorial task specification, by applying the optimized language translation described in Sect. III. Although CITLAB does not introduce its own test generators, existing test generator tools can be and have been easily embedded in it. We have used several external tools to generate combinatorial tests in the experiments of Sect. V. All the code and the experiments are openly available at the CITLAB web site (<http://code.google.com/a/eclipselabs.org/p/citlab/>).

III. TRANSLATION FROM FM TO CITLAB

In this section we present a procedure to translate feature models into combinatorial models. Our translation is per-

formed in three steps:

- A. Every feature, starting from the root feature, is translated to an element (variable or literal constant) in the combinatorial problem.
- B. Additional constraints are added in order to represent relationships among features as specified by the hierarchies in the feature model.
- C. Cross-tree constraints are translated and added to the model.

Several simplifications can be applied to the final combinatorial model, but these will be presented in the next section.

A. Representation of every feature

During the first step, all the features (corresponding to all the nodes in the feature model diagram) are translated as Boolean variables, enumerative variables or enumerative constants, according to Tab. II. Alternative features are translated as an enumerative variable, where the sub features, which are mutually exclusive by definition, are represented as values in a corresponding enumerative type. This represents a main difference between the proposed approach and the classical one: just a single enumerative variable is introduced to model alternative sub-features, instead of a set of Boolean variables. However, any other feature type is still translated to a Boolean variable, as in the classical approach.

During this step, the function *isChosen* is set for each encountered feature. *isChosen* defines the predicates that must hold if a feature is selected. It associates every feature to a propositional formula (i.e., a Boolean expression):

$$isChosen : Feature \rightarrow Expression$$

Consequently, *isChosen*(x) means that feature x is present in the considered product configuration.

B. Adding implicit constraints

During the second step of the translation, the feature model is visited again starting from the root node and constraints among the features in the original model are translated into constraints between the variables of the corresponding combinatorial task. Constraints are built depending on the node and its parents semantics mapping, as shown in Table III. We refer to these constraints as *implicit*, since they are implicitly implied by the type of relationships child-parent of the nodes.

C. Cross-tree constraints

Sometimes cross-tree constraints are used to limit valid product configurations and to model relationships among features. These constraints can be translated easily in CITLAB, which accepts as constraints general form Boolean expressions. Specifically, they are translated as follows:

FM constraint	CITLAB constraint
$A \text{ requires } B$	$isChosen(A) \Rightarrow isChosen(B)$
$A \text{ excludes } B$	$isChosen(A) \Leftrightarrow \neg isChosen(B)$

Example 2. Fig. 3 reports a small example which, however, contains all the feature kinds. Table IV reports the results

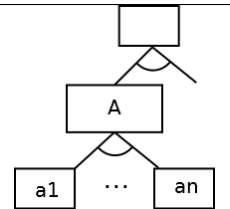
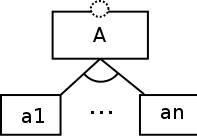
Feature			Any other node A_i whose parent is alternative P (A_i is a child in an alternative feature)	Any other node x whose parent is not Alternative
Parameter	Enumerative A {a1 ... an NONE}		skip (the father is alternative, the node is already translated into an element of an enumerative)	Boolean x
<i>isChosen</i>	skip <i>isChosen</i> (A)	$isChosen(A) \equiv A! = \text{NONE}$	$isChosen(a_i) \equiv P = A_i$	$isChosen(x) \equiv x = \text{true}$

Table II: Representing features and setting *isChosen* function

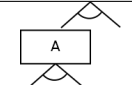
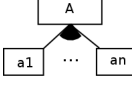
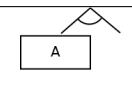


Feature		Constraint
	A and its parent are alternative	$A \neq \text{NONE} \Leftrightarrow isChosen(A)$
	A is Or and its parent is of any kind	$isChosen(A) \Rightarrow (isChosen(a1) \vee \dots \vee isChosen(an)) \wedge \forall i isChosen(a_i) \Rightarrow isChosen(A)$
	A is different from Or and parent is alternative	skip
	A mandatory and parent not alternative nor or	$isChosen(A) \Leftrightarrow isChosen(P)$
	A optional and parent not alternative nor or	$isChosen(A) \rightarrow isChosen(P)$

Table III: Constraints to be added

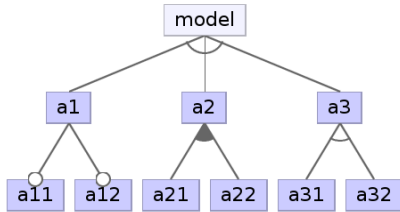


Figure 3: A small complete example

obtained by visiting each node during the visit of the diagram. The columns show the CITLAB Parameter, the value of the *isChosen* function, and the implicit constraints. Listing 1 reports the CITLAB code before simplification.

Variability factor: In our case, the variability factor can be computed as the ratio between the number of valid products and the size of Cartesian product of the parameter domains.

Listing 1: CITLAB code for model in Fig. 3

```

Model model
Parameters:
  Enumerative model { a1 a2 a3 NONE };
  Boolean a11;
  Boolean a12;
  Boolean a21;
  Boolean a22;
  Enumerative a3 { a31 a32 NONE };
end
Constraints:
  # model!=NONE #
  # a11==true => model==a1 #
  # a12==true => model==a1 #
  # model==a2 => ( a21==true || a22==true ) #
  # a21==true => model==a2 #
  # a22==true => model==a2 #
  # a3!=NONE <=> model==a3 #
end

```

D. Extra testing requirements

Once the combinatorial model is derived from a feature model, the tester can apply the usual combinatorial interaction testing criteria in order to obtain set of products that cover the family of products in a desired way. Besides this standard use of CIT, our approach supports two additional features that are implemented in CITLAB.

1) *Pre-built product configurations*:: known configurations that must be included in the final suite of test configurations can be imported into the combinatorial model as a list of *seeded* tuples of feature combinations, which are already supported by some combinatorial generation algorithms integrated in CITLAB. For instance, in the FM of Fig. 1 if the tester wants to force the inclusion of a specific product, he/she can write the following seed in CITLAB:

```

Seeds:
  # Calls = true, GPS = false, Screen = Color, Media = MP3#

```

CITLAB automatically checks that a seed sets the value to every parameter (hence to every feature) and that it satisfies the constraints (implicit and cross-tree).

node	CITLAB Parameter	<i>isChosen</i>	CITLAB constraint
model	Enumerative model {a1 a2 a3 NONE}	model!=NONE	model!=NONE
a1	skip	model == a1	skip
a11	Boolean a11	a11 == true	a11 == true => model == a1
a12	Boolean a12	a12 == true	a12 == true => model == a1
a2	skip	model == a2	model == a2 => (a21 == true ∨ a22 == true)
a21	Boolean a21	a21 == true	a21 == true => model == a2
a22	Boolean a22	a22 == true	a22 == true => model == a2
a3	Enumerative a3 {a31 a32 NONE}	model == a3	a3 != NONE <=> model == a3
a31	skip	a3 == a31	skip
a32	skip	a3 == a32	skip

Table IV: Translation to CITLAB model for model of Fig. 3

2) *Further testing goals*: Testers may want to add extra requirements about the testing activity in form of further conditions over the product space that they want to cover. These test goals must be achieved either beside or by the combinatorial coverage and represent particular product configurations the tester wants to be sure that they will be included in the final test suite. These extra constraints can be added in CITLAB by means of *test goals*. For instance, in the FM of Fig. 1 if the tester wants to include at least a product in which the GPS is present but Media is not MP3, he can write the following test goal:

TestGoals: # GPS == true and Media != MP3#

A test goal is *covered* if there exists at least a test that satisfies it.

For now, we assume that these extra testing requirements are manually inserted by the user after the translation is performed. We plan to study a way introduce them in the source feature models. For a precise semantics of test goals and seeds, see [8].

IV. SIMPLIFICATION PROCESS

The combinatorial model resulting from the imported feature model can be optimized in order to reduce the number of unnecessary variables and constraints. Although the proposed optimization is proposed in the context of feature model combinatorial testing, it can be applied to any combinatorial model and it has been implemented in CITLAB as model transformation and added to its public APIs. We perform two types of syntactical simplifications:

- We simplify the constraints of the model in a semantic preserving way. The goal is to ease the test generation, which in the presence of constraints can be more difficult [5].
- We completely remove unnecessary parameters and constraints.

A. Constraint simplification

We apply the rules presented in the following table, which reports the constraint to be simplified (column A), the condition under which it can be simplified (column B), and the performed simplification (column C), where a and b can be either atomic proposition or complex propositions. It is

Listing 2: CITLAB code for model in Fig. 3 after simplification

```

Model model_smp
Parameters:
  Enumerative model { a1 a2 a3 };
  Boolean a11;
  Boolean a12;
  Boolean a21;
  Boolean a22;
  Enumerative a3 { a31 a32 NONE };
end
Constraints:
  # a11==true => model==a1 #
  # a12==true => model==a1 #
  # model==a2 => ( a21==true || a22==true ) #
  # a21==true => model==a2 #
  # a22==true => model==a2 #
  # a3!=NONE <=> model==a3 #
end

```

Parameter	Constraint	Action
Boolean x	$x = true$ or $x = false$	remove x and remove the constraint (if x does not compare in other constraints).
enum $A\{a1... an\}$	$A \neq ai$	remove ai from type A and remove the constraint (if ai does not compare in other constraints).
enum $A\{a1... an\}$	$A = ai$	remove A and the constraint (if neither A nor ai compare in other constraints).

Table V: Parameter simplification

straightforward to prove that the simplifications preserve the final semantics of the constraints.

(A) Constraint	(B) If already present	(C) Replaced by
$a \rightarrow b$	a	b
$a \rightarrow b$	b	- (remove)
$a \leftrightarrow b$	a	b
$a \leftrightarrow b$	b	a

B. Parameter simplification

Once the constraints have been simplified, we can simplify the model parameters, as reported in Tab. V. In those cases, under some assumptions, the parameters can be simplified and the constraints removed.

Example 3. For the example of two alternative features given

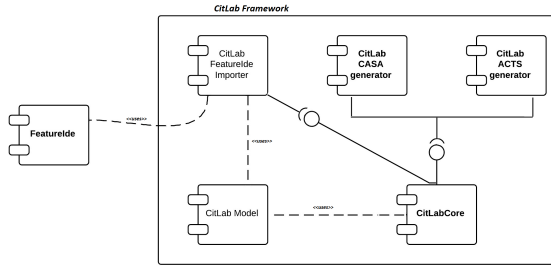


Figure 4: FeatureIDE importer plugin

in Fig. 2, the resulting CITLAB file is the following one:

```

Model model
Enumerative A {a1 a2 a3 a4};
Enumerative B {b1 b2 b3 b4};
end

```

Note that in this specific case, our simplification algorithm is able to remove all the constraints. Using unconstrained models, the tester has access to a wide set of tools, including most algebraic and bio-inspired tools for combinatorial testing (like [3]), which are very powerful in terms of generation time and test suite size, but they cannot easily deal with the constraints.

Note that since we operate only at syntactical level, we may miss some possible simplifications but the process is simple and fast. We plan to use in the future a constraint solving engine (like a SMT solver).

V. EXPERIMENTS

We have implemented in CITLAB an importer of FeatureIDE models, which can read feature models in the FeatureIDE native format and also in the SPLOT format (sxfm). We have also implemented an importer that translates feature models in SPLOT format into a combinatorial problem with only Boolean variables with the classical semantics given by [13]. These importers are CITLAB plugins and they are integrated in the framework together with other test generators as shown in Fig. 4.

As case studies we have taken 52 SPLOT² models which are often used as benchmarks³

A. Testing the correctness of the transformation

Formally proving the correctness of the proposed transformation would require a standard formal framework for feature models semantics, which is not available yet, although Schobbens and Benavides have made some progress in this direction [18], [2]. In this paper we focus on *testing* the correctness of the proposed transformation by comparing it with the classical transformation using propositional logics. We want to assess the *semantic equivalence*: all the proposed transformation rules preserve the semantics of the feature

²SPLOT can be found at <http://www.splot-research.org/>.

³Note that because of a bug in FeatureIDE sxfm parser, we had to modify few SPLOT examples.

model. To systematically test the correctness of our translation we need to check the following characteristics:

- *Consistency*: our approach generates only valid products, i.e., products that satisfy the constraints.
- *Completeness*: our approach generates all the valid products.

The consistency is checked automatically within the integrated test case evaluator of CITLAB. CITLAB has an integrated logic evaluator to check for inconsistency and whether a parameter (feature) configuration is a valid test case or not. We rely on the fact that the internal evaluator works correctly.

The completeness is checked by simply verify for every case study that the number of distinct valid product of our approach is equal to the number of valid products found by the SPLOT analyzer. Note that the number of products including invalid ones may differ, but the number of valid configurations must be the same in order to preserve semantical equivalence.

The semantical equivalence ensures a biunivocal correspondence between one test case produced by CITLAB and one possible valid product.

B. Effect of the simplification over the parameters and the constraints

We want to check if the simplification process reduces the number of parameters and constraints. Fig. 5 reports the number of parameters (Fig. 5a) and constraints (Fig. 5b) before (CTL) and after the simplification (CTL S). The simplification process has always reduced the number of both quantities in the models considered for experimentation. Our technique was able to remove all the constraints in 4 models.

C. Comparison with approaches using Boolean variables

We want to compare our approach with those using Boolean variables (like [2]) to check the effectiveness of our methodologies over the following quantities of the final models:

parameters: we should obtain *smaller* models

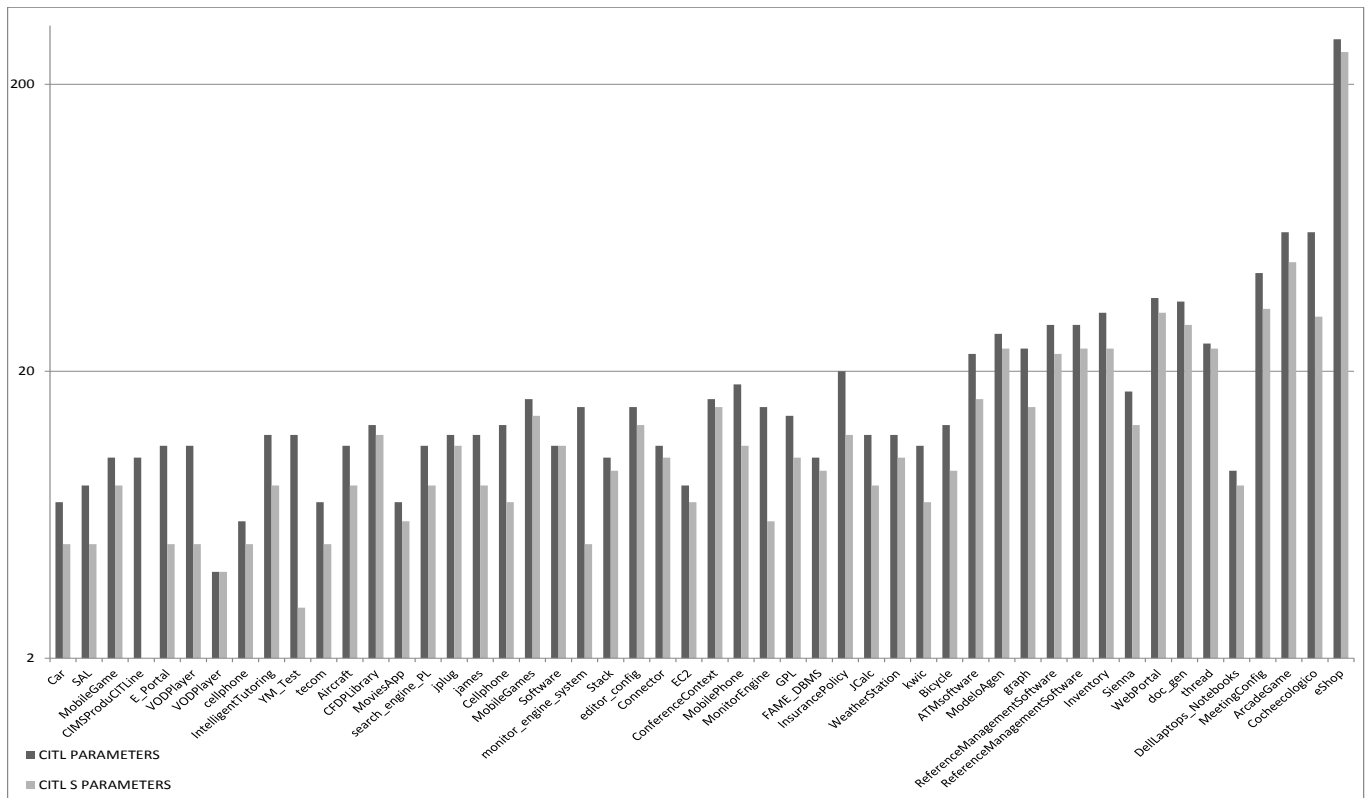
constraints: we should obtain *simpler* models

variability: we should obtain more *compact* models

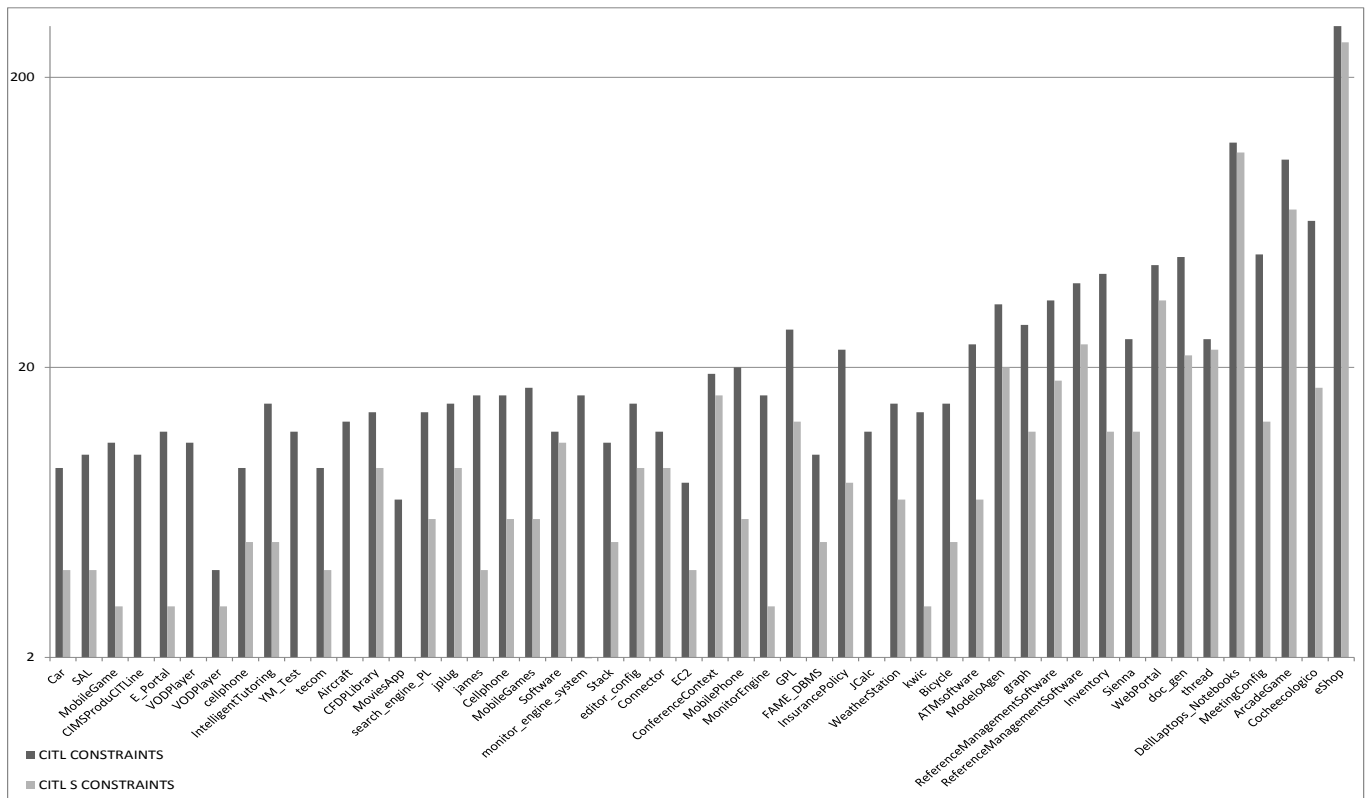
Fig. 6a compares the number of CITLAB parameters (after simplification) w.r.t. the number of features in the original model. It is apparent that our technique is able to reduce the number of parameters of problem. This should make the test generation faster to perform.

Fig. 6b compares the number of CITLAB constraints w.r.t. the number of constraints in the model obtained by using the classical translation implemented by SPLOT. Since SPLOT represents all the constraints in CNF, in order to perform a fair comparison, we have converted also the CITLAB constraints to CNF and the figure compares the total number of clauses in the CNF expressions. As the figure shows, our final models have always fewer constraints than those in SPLOT.

Fig. 7 shows the variability factor of our final models. The figure on the left shows the variability factor before and after the simplification process, while the figure on the right compares the variability factor of CITLAB models with that of SPLOT models. The figures show that the simplification

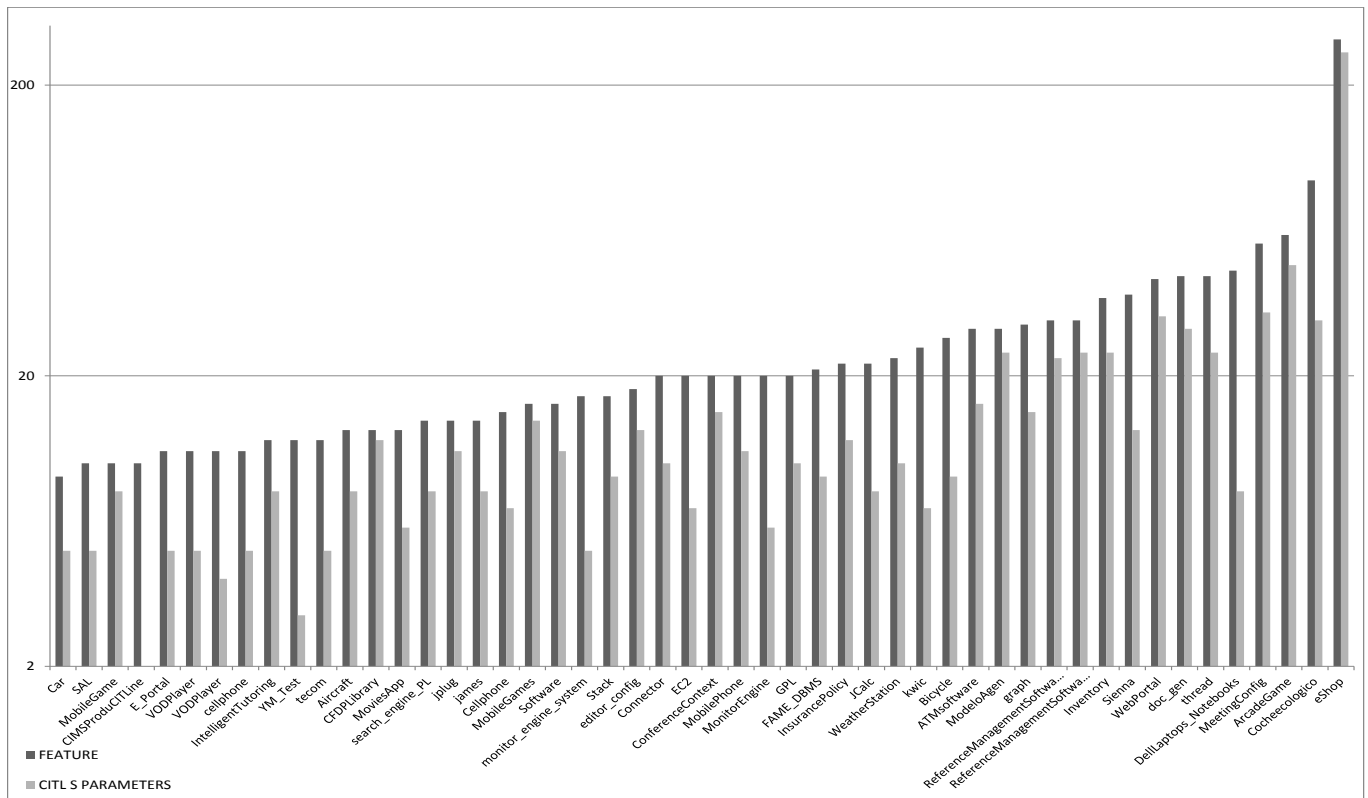


(a) Number of parameters

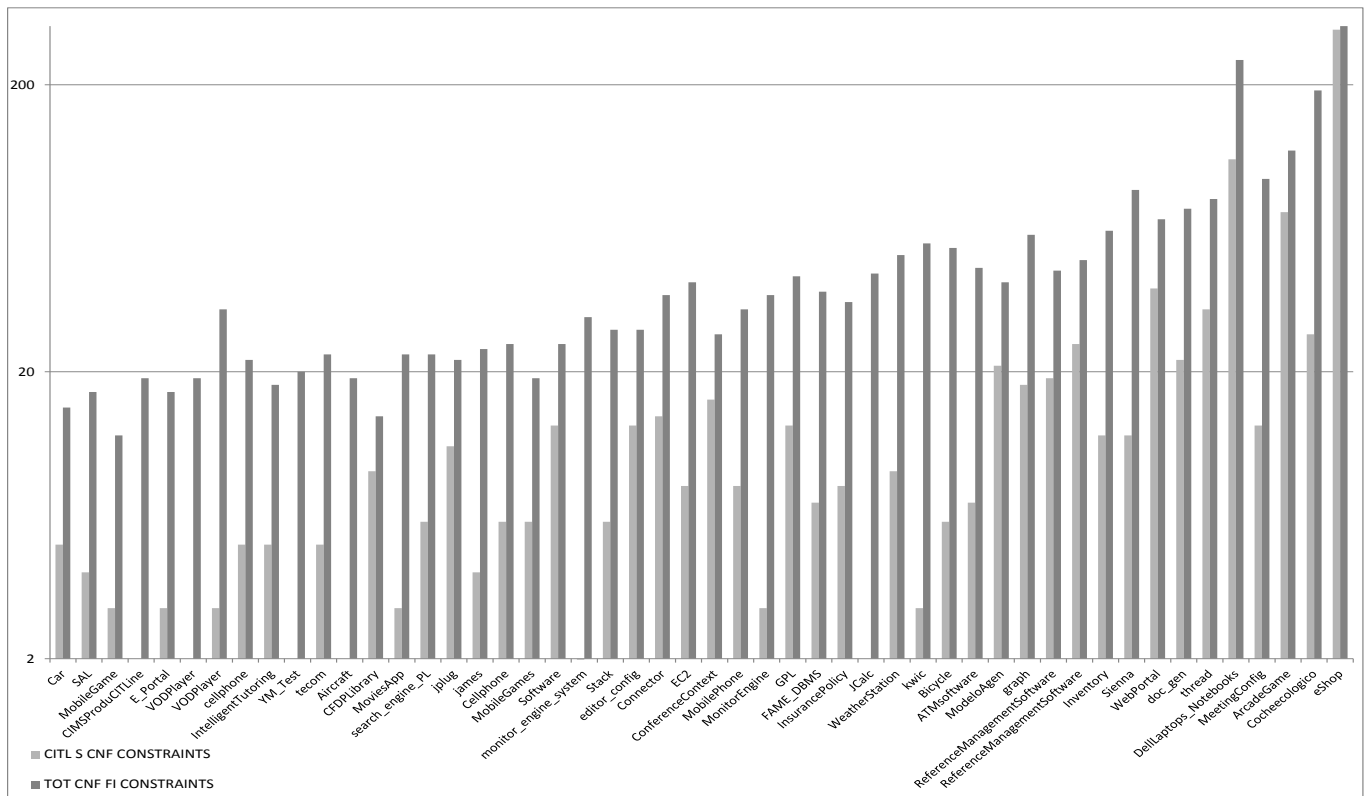


(b) Number of constraints

Figure 5: Benefits of our model simplification process



(a) Comparison of number of CitLab Parameters vs Boolean variables



(b) Comparison of CitLab constraints vs SPLIT constraints

Figure 6: Modeling comparison

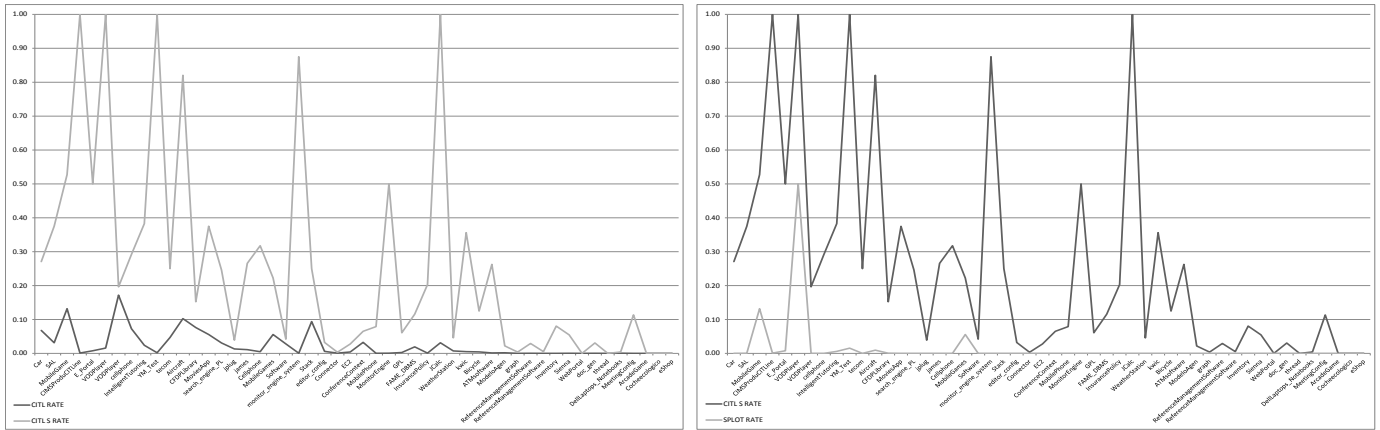


Figure 7: Variability comparison

	Aircraft		FameDBMS		Dell laptops		Printer	
	Time	Size	Time	Size	Time	Size	Time	Size
BOOLEAN	0.046	10	1.169	13	0.519	39	out of memory	
BOOLEAN S	0.014	10	0.976	13	0.785	39	out of memory	
CITL	0.017	10	0.020	13	4.296	36	20	180
CITL S	0.005	8	0.006	12	2.043	37	12	180

Table VI: Pairwise with ACTS (time in seconds)

	CITLAB by CASA		Oster [15]	PACOGEN [10]
	Size	Time	Size	Size
Sienna	20	1.76	24	20
Inventory	9	3.06	12	15
ArcadeGame	13	37.90	25	14
Web Portal	15	24.01	26	16
Doc generator	13	11.41	18	17

Table VII: Pairwise with CASA (time in seconds)

increases the variability factor and that our final models have a higher variability factor than the original SPLIT models. This means that in our approach valid products occur more often in the product space.

D. Test generation

A major advantage of our approach is that the tester can relay for test generation on different algorithms and tools developed for CIT. We want to evaluate the impact of our approach to the actual generation of combinatorial test suites.

Table VI reports the results of pairwise test generation using ACTS⁴ which implements the IPOG algorithm [12] and which is integrated in CITLAB as generator plugin. We have run 100 test generations for 4 SPLIT models, by using the translation to only Boolean variables (BOOLEAN) and the method proposed in this paper (CITL), where the simplification is applied (S) or not. Note that the coverage requirements for all these translation methods are the same. We generally obtain better results, i.e., both faster generation *and* smaller or equal test suites, with the proposed translation than that using only Boolean variables. However, there is one exception. For the Dell Laptop example, we have a smaller test suite but at the expenses of the test generation time. It seems that IPOG can further reduce the test suite size when using our models containing fewer parameters (8 parameters against 48 features) and simpler constraints. Per the biggest case study (Printer) and the BOOLEAN translation, ACTS did not complete the generation.

Table VII reports the test generation of 5 case studies⁵ using

⁴<http://csrc.nist.gov/groups/SNS/acts/>

⁵We chose these examples because data about using other tools is available in literature papers.

our translation and the test generator tool CASA [9] which is integrated in CITLAB as generator plugin. In our experiments, CASA is able to produce smaller test suites than others test generation algorithms specifically designed for SPL testing.

Overall, we can say that our encoding has also advantages during test generation (in terms either of time or test suite size): some tools can take advantage of our simpler encoding and testers can benefit from having access to very powerful test generation frameworks not designed specifically for SPLs. However, further experiments are needed in order to give a final positive evaluation regarding test generation.

VI. RELATED WORK

There exist several attempts to give a precise semantics to feature models. Batory connects feature models, grammars, and propositional formulas [1] by giving a very simple yet clear meaning to feature models. The proposed connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfiability solvers to debug feature models. It would also allow the use of tools for generating combinatorial test suites, although this topic is not tackled in [1].

A more complete (including several variants of Feature Diagrams) and powerful semantics of feature models is presented by Shobbens [18], who emphasizes the necessity of precision and unambiguity for efficient and safe tool automation. They provide a formal semantics by a generic construction called Free Feature Diagrams (FFDs). Test generation from FFDs can be complex though.

Benavides et al. present several mappings from feature models to other formal notations (propositional logic and CSP) [2]. They focus more on automated analysis instead of testing, but our translation has been greatly influenced by their survey.

Regarding testing feature models and SPLs, a good survey can be found in [7]. A first attempt to apply CIT in the form of covering arrays to SPLs, using a simple variant of feature models called Orthogonality Variability Model (OVM), can be found in [6]. They define several testing criteria which are adaptation of combinatorial criteria to OVMs and identify some open issues like scalability, the use of constraints, and the benchmarking. We believe that reducing the problem of SPL testing to a CIT problem can help to deal with all the issues mentioned in that paper.

In [16], the authors propose a scalable toolset using Alloy to automatically generate test cases satisfying T-wise from SPL models. The proposed tools set is based on the use of a SAT solver. However, an extension of the approach by using also Constraint Programming is presented in [17]. In that paper the authors present and evaluate two techniques, one focusing on generality and using high level strategies in order to improve the test generation. The other emphasizes providing efficient generation.

The tool PACOGEN is presented in [10]. PACOGEN relies on constraint programming to generate configurations that satisfy all constraints imposed by the feature model and to minimize the set of the tests configurations. Extensive experiments, based on the state-of-the art SPLOT feature models repository, shows that PACOGEN scales well and produces reasonable small test suites also for large SPLs. A specialized algorithm (called ICPL) for generating covering arrays from feature models is presented in [11].

All these algorithms are specific to SPL testing but we believe that they could be used also for combinatorial testing, although they may need some modification in order to accept generic combinatorial problems. On the other hand, SPL testing could benefit from CIT algorithms, tools, and concepts (like seeds and test goals). Our experiments show that reusing CIT test generation tools for SPL testing can have great advantages.

VII. CONCLUSION AND FUTURE WORK

We have presented a mapping from feature models to combinatorial interaction problems which can leverage a combinatorial testing framework like CITLAB in order to generate test suites for software product lines or for system products. Our translation provides several advantages over classical mappings, like the reduction of the parameters and constraints, the possible use of concepts like seeds and test goals, and the exploitation of external tools developed during these years for CIT. The approach has been implemented in CITLAB, a framework for CIT, which aims to integrate notations (using importers) and test generation algorithms in a friendly easily usable way.

For now, we support only simple feature models, but we plan to extend our translation to support feature models with

extended notations, like those with cardinality and grouping.

REFERENCES

- [1] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.
- [2] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
- [3] Renée C. Bryce and Charles J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.
- [4] A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial interaction testing with CitLab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tools Track.*, 2013.
- [5] 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(5):633–650, sept.-oct. 2008.
- [6] M.B. Cohen, M.B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63. ACM, 2006.
- [7] E. Engström and P. Runeson. Software product line testing—a systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.
- [8] Angelo Gargantini and Paolo Vavassori. Citlab: a laboratory for combinatorial interaction testing. In *Workshop on Combinatorial Testing (CT) - ICST*, pages 559–568, Montreal, Canada, 2012. IEEE Computer Society.
- [9] B.J. Garvin, M.B. Cohen, and M.B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 13–22. IEEE, 2009.
- [10] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. PACOGEN: Automatic generation of pairwise test configurations from feature models. In *Proc. of Int. Symp. on Soft. Reliability Engineering (ISSRE'11)*, 2011.
- [11] M.F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55. ACM, 2012.
- [12] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, September 2008.
- [13] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.
- [14] S. Oster, I. Zorcic, F. Markert, and M. Lochau. MoSo-PoLiTe: tool support for pairwise and model-based software product line testing. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 79–82. ACM, 2011.
- [15] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin Heidelberg, 2010.
- [16] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 459–468. IEEE, 2010.
- [17] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.
- [18] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
- [19] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012. to appear.