Open access • Proceedings Article • DOI:10.1109/ISSRE.2011.31

PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models — Source link

Aymeric Hervieu, Benoit Baudry, Arnaud Gotlieb

Published on: 29 Nov 2011 - International Symposium on Software Reliability Engineering

Topics: Feature model, Feature (computer vision), Automatic test pattern generation and Constraint programming

PACOGEN : Automatic Generation of Pairwise Test Configurations from Feature Models

Aymeric Hervieu, Benoit Baudry, Arnaud Gotlieb

PACOGEN : Automatic Generation of Pairwise Test Configurations from Feature Models

Aymeric Hervieu, Benoit Baudry
INRIA Rennes Bretagne Atlantique
Rennes, France
Email: {Aymeric.Hervieu,Benoit.Baudry}@inria.fr

Arnaud Gotlieb
Certus Software V&V Center
Simula Research Laboratory,
Oslo, Norway

***Abstract*—Feature models are commonly used to specify variability in software product lines. Several tools support feature models for variability management at different steps in the development process. However, tool support for test configuration generation is currently limited. This test generation task consists in systematically selecting a set of configurations that represent a relevant sample of the variability space and that can be used to test the product line. In this paper we propose PACOGEN to analyze feature models and automatically generate a set of configurations that cover all pairwise interactions between features. 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. This work also proposes an extensive experiment, based on the state-of-the art SPLOT feature models repository, showing that PACOGEN scales over variability spaces with millions of configurations and covers pairwise with less configurations than other available tools.**

I. INTRODUCTION

Feature models (FMs) allow companies to reason over a large number of variants for their software systems [1]. Several tools exist to manage variability with feature models [2], debug configurations [3] and to derive specific configurations [4]. However, very few techniques or tool support the testing activity in software product line engineering. The size of the variability space is a major challenge for testing. Realistic feature models can represent millions of variants, which means that exhaustive testing is impossible in most cases. For example, the feature model for Arcade video games in the SPLOT repository [5] is composed of 61 features, but more than 1 million valid products can be derived from this model.

One challenge consists in selecting a small subset of all possible configurations for testing. This sample should cover relevant characteristics of the feature model, while staying as small as possible. Keeping a small sample is crucial to limit the effort necessary for testing each selected configuration. Recent work suggest exploring pairwise coverage [6] to sample the variability space. *The automatic generation of a minimal set of configurations that cover pairwise interactions faces two challenges: (i) feature models specify dependencies between features which forbid some pairwise feature interactions; (ii) the generation of a minimal pairwise test set is a complex optimization problem*. Perrouin et al. [7] and Oster et al. [8] have recently investigated the first issue. In this work, we explore both issues at the same time (minimization and dependencies between features).

We propose PACOGEN [1] for the automatic generation of test configurations that cover all valid pairwise interactions in a feature model. PACOGEN has two major characteristics: it processes feature models directly from Eclipse, the most common IDE for model-driven development; test generation is based on constraint programming. We choose constraint programming first because of its flexibility to deal with dependencies between features for test generation: this programming paradigm allows us to design a tailor-made *pairwise* constraint. Second, unlike SAT-solving, constraint programming is well suited for optimization problems such as those related to the minimization of the size of test sets. PACOGEN users can decide to ask for the smallest test set that covers pairwise interactions, or they can ask for the smallest solution that can be found in a given amount of time (*anytime minimization*).

The paper presents the following contributions:

1) PACOGEN , a constraint-based testing tool for automatic generation of test configurations that cover all pairwise interactions in a feature model ;
2) A series of experiment with 69 feature models from SPLOT [2], one of the largest and up-to-date feature model repository. The main results show that our strategy for pairwise generation scales over variability spaces that specify millions of configurations and that we can generate less configurations than state of the art techniques [7], [8].

The rest of the paper is organized as follows. Section 2 presents some background on combinatorial interaction testing. Section 3 details our constraint-based model of FMs and how pairwise coverage can be enforced in a set of test configurations. Section 4 explains how the number of test configurations can be minimized through the usage of

[1] http://www.irisa.fr/lande/gotlieb/resources/Pacogen/Pacogen.html
[2] http://www.splot-research.org/

well-known Constraint Programming techniques. Section 5 presents experiments that compare our approach to other techniques and that run the generation 69 models from SPLOT. Finally, Sec. 6 draws some conclusions and perspectives of this work.

II. Background

This section briefly introduces the metamodel we use to build feature models. Then, we define pairwise interaction coverage over a feature model. We also emphasize some issues that must for the automatic generation of a minimal test configurations set.

A. Feature models

Perrouin et al. [9] have built a metamodel that formally captures the definition of a feature model, on the basis of the work by Schobbens et al. [10]. The metamodel is displayed in figure 1 and defines the structure of a feature model as follows:

- A feature model (FeatureDiagram class) is composed of a set of Features.
- We distinguish between Features and Primitive-Features. Our test generation process considers only interactions between primitive features.
- One parent Feature is related to a set of children features through an Operator or through a binary constraint. A Feature can also have a list of Attributes.
- Five different Operators can relate a parent feature to its children: And, Or, Xor, Opt, Card.
- Two Constraints can relate features that are not parent / children: Require and Mutex.

Figure 2 displays a small feature model for a car break system (White et al. [11]). This model conforms to the metamodel definition. It specifies that a car has a backward sensor that can associated with a Lateral Range Finder (*LRF*) or Forward Range Finder (*FRF*). A car also has an optional Automated Driving Controller (*ADC*). An ADC must have a Collision Avoidance Breaking (*CAB*): either Standard Avoidance (*SA*) or Enhanced Avoidance (*EA*). An ADC also has an option for Parallel Parking (*PP*). The feature model specifies two cross-tree constraints: if the PP option is chosen, then the car must have a LRF; if the EA is chosen, then the car must have FRF.

B. Pairwise testing for selecting test configurations

Pairwise testing is a particular case of *combinatorial interaction testing* (CIT) introduced by Cohen et al. [6] to sample large test input domains. This test selection technique focuses on the subset of the input domain that covers all value combinations for each pair of variables. For example, the car configuration [ADC, EA, PP, LRF, FRF] covers the interactions between the following pairs of values: (ADC,EA); (ADC,PP); (ADC,LRF); (ADC,FRF); (EA,PP); (EA,LRF); (EA,FRF); (PP,LRF); (PP,FRF); (LRF,FRF).

CIT uses the mathematical structure called a mixed-level covering array.

Definition 1: A mixed level *covering array*

$$MCA(N; t, k, (v_1, v_2, ..., v_k))$$

is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1) *Each column i ($1 \leq i \leq k$) contains only elements from a set D_i of size v_i.*
2) *The row of each $N \times t$ sub-array covers all t-tuples of values from the t-combination of columns at least once.*

For pairwise testing over feature models, the strength t of the array is equal to 2, v is the number of primitive features and all these variables are defined on a domain v_i of size 2 (all boolean variables).

A general issue for pairwise test generation is to determine the number N of necessary lines in the array in order to cover all interactions between variable pairs. This is a complex optimization problem that has been tackled mainly through three types of solutions [12]: algebraic constructions, greedy algorithms and meta-heuristics. However, most previous solutions consider that all values for all variables are independent, *i.e.*, all value interactions are possible.

Building a set of pairwise configurations from feature models requires to think about new generation strategies because a number of value interactions are forbidden. For example, the pair (EA = selected, FRF = unselected) is not a valid pair according to the feature model in figure 2. Thus the optimization problem we tackle here can be formulated as follows

> *Given a feature model we look for the minimum number of valid configurations that cover all the authorized interactions between pairs of features.*

In the following section we introduce a new constraint model for feature model that targets both the generation of valid configurations with respect to the feature model and the minimization of test configurations to cover pairwise interactions.

III. Feature Models as constraints

Our approach is based on constraint programming through the mapping of a feature model into a finite domain constraint model. This section details our model and the structure used to generate test configurations for pairwise testing. We start by recalling what is a global constraint (Sec.III-A). Then, we detail the constraints generated for modeling parent/children operators from the feature model and those generated for handling cross-tree links (Sec.III-B). Finally, we explain how to enforce pairwise coverage on a set of test configurations (Sec.III-C). An important contribution of our

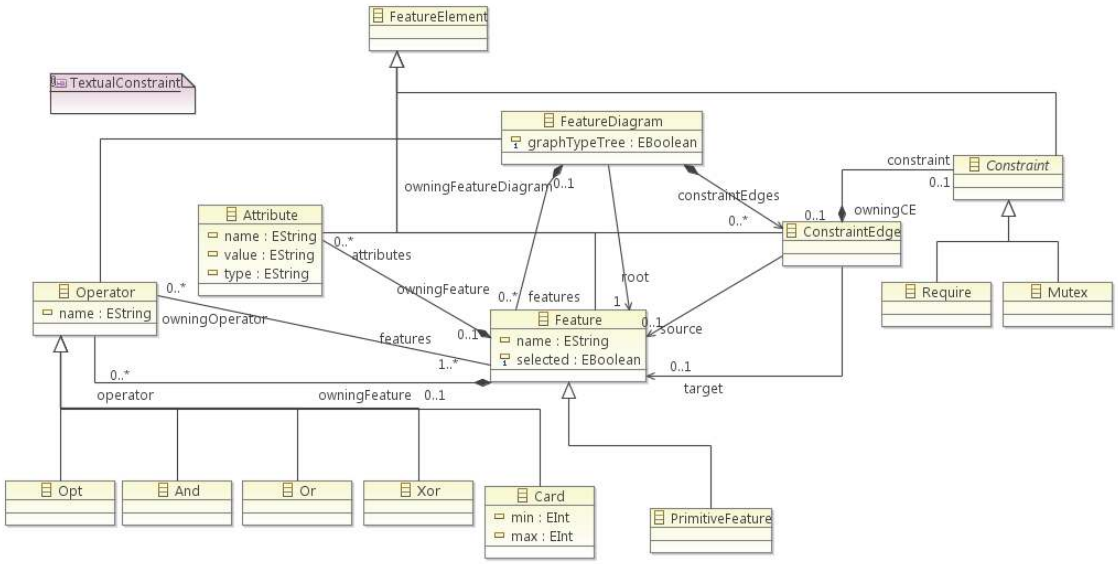

Figure 1. A metamodel for Feature Models

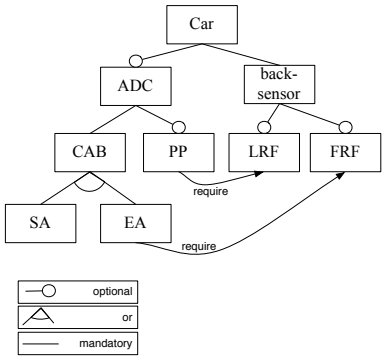

Figure 2. Feature Model of a Car Break System [11]

work is the definition of a new global constraint that enforces a set of values to be included within a pair of variable vectors. This constraint was not reported elsewhere, although the usage of global constraints for handling feature models was already skeched [13]. The filtering algorithm developed for this global constraint is a key point of our approach as it allows pairwise coverage to be enforced within a set of test configurations.

A. Global constraints

An important feature of constraint programming techniques is their ability to allow users to define new special-purpose constraints under the form of global constraints. A *global constraint* is a relation defined by an interface (operator's name and constrained variables), a filtering algorithm and awakening conditions. For example, the global constraint $\mathsf{all\_different}([X_1, .., X_n])$ constrains the set of finite domain variables $X_1, .., X_n$ to take distinct values whatever those values. Its filtering algorithm is based on the matching theory [14] and its awakening conditions may vary from one implementation to another but often correspond to the discovery that one of the variables has become instantiated. In fact, a global constraint is handled as any primitive constraint by the solver and its filtering algorithm is launched anytime one of its awakening conditions is satisfied during constraint propagation.

B. Constraints for feature model links

From a feature model, the problem of generating a sequence of test configurations enforcing pairwise coverage is mainly a constraint modelling problem. The model includes two sets of constraints: the constraints handling inheritance links and the constraints handling cross-tree links. As described above, inheritance links are hierarchical relations between features (namely, OPT, AND, OR, XOR, CARD) while cross-tree links are alternative relations among unrelated features (namely, REQUIRE, MUTEX). Based on their definition, we have defined dedicated global constraints to capture these relations. For inheritance links, global constraints

$$\mathsf{opt}(A, [B, C, ...]), \mathsf{and}(A, [B, C, ...]), \mathsf{or}(A, [B, C, ...])$$

$$\mathsf{xor}(A, [B, C, ...], \mathsf{card}(A, [B, C, ...], N, M)$$

holding over two-valued (i.e., 0 and 1) domain variables are used. These relations express the hierarchical relation among the feature parent A and his children $B, C, ...$ The cardinality global constraint existed already in the CP community [15]. In addition to the $A, B, C, ...$ variables, it takes

two additional variables N and M as inputs for the minimum and maximum number of children, but these variables are always instantiated in feature models. For REQUIRE and MUTEX, we encoded the corresponding logical relations (*implication* and *mutual exclusion* respectively) onto the two-valued domain variables associated to feature children. No further details on these relations are necessary as they are trivial to implement.

C. Enforcing pairwise coverage through constraints

In order to build a sequence of test configurations enforcing pairwise coverage of a given feature model, we defined a specific data structure that is incrementally filled in. The data structure is basically a matrix with columns containing values of all the features and rows representing test configurations. Our framework fills in this matrix in order to find a sequence of test configurations (the rows of the matrix) that covers the pairwise criterion. At the begining of the process, the matrix is filled in with two-valued unknown variables and its size is positionned to a sufficently large constant[3] as the number of test configurations is unknown (and cannot be computed from analytical results [16]). At the end of the process, this matrix encodes a solution of the problem, that looks like:

$Conf. \diagdown Feat.$	A	B	C	...
1	1	0	0	...
2	0	1	1	...
...	...			

Enforcing pairwise coverage within this matrix data structure requires each pair of feature values to be included in the matrix. This requirement is implemented through the usage of a new global constraint, also called pairwise. This relation holds over a variable I representing an unknown line in the matrix and two vectors of feature values corresponding to columns of the matrix. The constraint enforces a specific pair of values (e.g., $(1,1)$) to be included in the vectors. For example, $\mathsf{pairwise}(I, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (1,1))$ constrains an unknown row I of a 3-rows matrix to contain the pair $(1,1)$, meaning that the corresponding features should be included within the test configuration of rank I. The domain of I is $1..3$ in this example. During the final labelling step, if I becomes instantiated to 2 then $(X_2, Y_2) = (1,1)$ whereas if X_3 is instantiated to 0 then 3 will be removed from the domain of I. In this latter case, the pair (X_3, Y_3) cannot be equal to $(1,1)$ but there is not enough information to instantiate the variables. The constraint pairwise will be suspended until more information becomes available.

The filtering algorithm shown below is used for constraint $\mathsf{pairwise}(I, (L_1, L_2), (v_1, v_2))$, where I is finite domain variable, L_1 and L_2 are two lists of finite domain variables, and (v_1, v_2) is a pair of values. This algorithm is launched each time at least one of the domains of I,$L_1[i]$ or $L_2[j]$ is pruned (awakening conditions). The underlying idea is to explore each of the possible values of I and to determine whether this value is still consistent with the domain of other variables. The complexity of this algorithm is linear w.r.t. the domain size of I, as it iterates only on the possible values of I.

Input: I a finite domain, L_1, L_2 two lists of finite domains of the same size and (v_1, v_2) a pair of integer values
Output: $Fail$ or pruned domains for (I, L_1, L_2)

function pairwise$(I, (L_1, L_2), (v_1, v_2))$
$I' \leftarrow I,\ T_1 \leftarrow \emptyset,\ T_2 \leftarrow \emptyset$;
foreach $i \in I$ **do**
  if $(v_1 \notin L_1[i])$ ***or*** $(v_2 \notin L_2[i])$ **then**
    $I' = I' \backslash \{i\}$
  else
    $T_1 \leftarrow T_1 \cup L_1[i],\quad L'_1[i] \leftarrow L_1[i]$;
    $T_2 \leftarrow T_2 \cup L_2[i],\quad L'_2[i] \leftarrow L_2[i]$;
if $I' = \{a\}$ **then**
  $L'_1[a] = v_1;\ L'_2[a] = v_2;$ **return** $(\{a\}, L'_1, L'_2)$;
else if $(I' = \emptyset$ ***or*** $v_1 \notin T_1$ ***or*** $v_2 \notin T_2)$ **then**
  return *Fail*
else
  return (I', L'_1, L'_2)

IV. PAIRWISE COVERAGE AS A TIME-CONSTRAINED MINIMIZATION PROBLEM

As said previously, finding the minimum number of test configurations covering the pairwise criterion is a challenging problem. This section explains how this problem can be addressed using the constraint model described above. We explain first how to formulate it as an optimization problem (Sec.IV-A), and second, we solve it using an anytime labelling search procedure (Sec.IV-B).

A. An optimization problem

Our goal is to find the minimum number of test configurations, i.e., the minimum number of lines to instantiate in the matrix for covering the pairwise criterion. This can be achieved by searching the minimum of a cost function f, as follows:

Find $I_1, ..., I_{4n^2}$ **such that** $Min(f)$
And $\forall i, j$ **in** $1..n$,
$\mathsf{pairwise}(I_k, (C_A, C_B), (1,1))$,
$\mathsf{pairwise}(I_{k+1}, (C_A, C_B), (1,0))$,
$\mathsf{pairwise}(I_{k+2}, (C_A, C_B), (0,1))$,
$\mathsf{pairwise}(I_{k+3}, (C_A, C_B), (0,0))$
where n denotes the number of features while the C_A, C_B denote the columns of the matrix representing features A

[3] In practice, we used the value 50

and B. Each I_k denotes the line of the matrix. Note that additional constraints implicitly enforce all the $C_A[i], C_B[j]$ to be part of the feature model.

Several functions can be considered for minimization. In our framework, we explored two semantically equivalent formulations, namely
$f_1 = \sum_{k \in 1..4n^2} I_k$ and $f_2 = \mathsf{Max}_{k \in 1..4n^2} I_k$
Both functions can be used in order to find the minimum number of values for the I_k, such that pairwise is satisfied in the matrix. For solving the optimization problem, we used the well-known *branch-and-bound method* that explores feasible solutions while maintaining the cost function as low as possible. Roughly speaking, at each node of the search tree, the branch-and-bound method evaluates the cost function, prunes subtrees for which the cost will be clearly higher than a current value and selects the subtree that has the least cost. Several parameters impact the search, including the way variables and values are selected for labelling. Another characteristic of the constraint solving techniques we used is their versatility for addressing feature models. This has already been reported in the literature [17]. In particular, additional constraints can easily be defined to take into account extended feature models and several search heuristics can be exploited to generate test configurations. In our framework, we selected the variable with the smallest domain to be enumerated first and from the domain of this variable, we selected the smallest value first. Other heuristics may be chosen but our experimental results showed that these ones are sufficient to handle the largest feature models.

B. Anytime minimization

The constraint solving techniques used in our framework share an interesting property with anytime algorithms [18]: they can be stopped at any time or can be given a time or resource contract. In our framework, we gave a time-contract to the branch-and-bound method. As a result, as soon as a first feasible solution has been found, it returns the optimal number of test configurations enforcing pairwise coverage, found only in a given amount of time (ranging from a few seconds to more than three hours). Of course, better feasible solutions might be found if more time is allocated to the search. However, allocating a time-contract to the search permits one to balance advantageously between quality of the solution w.r.t. time needed to find it.

In our experiments, we observed that the *branch-and-bound method* computes good-quality solutions (i.e., near-optimal solution) in little amount of time and most of the remaining time is used to prove that no better quality solutions really exist. Hence, by relaxing the problem to the finding of near-optimal solutions only, we got a very efficient way to solve our challenging minimization problem.

V. PACOGEN

In this section, we present our PACOGEN implementation and we show how it processes a feature model for generating a set of valid pairwise-covering test configurations (i.e., a set of configurations that covers all pairwise interactions between features). PACOGEN has characteristics that make it suitable for software product line engineering (SPLE). First, the input Feature Model is interpreted as an instance of the metamodel of Fig.1. This design decision is meant to include PACOGEN directly in a modelling environment dedicated to SPLE. In addition, the processing of Feature Models is independant from the *test configurations* generation process Second, we used constraint programming to develop PACOGEN , which is flexible enough to easily customize the *test configurations* generation process.

Implementation. The *test configurations* generation process shown in Fig.3 can be explained as follows: first, PACOGEN transforms the feature model into a constraint model ; second, it adds all the valid pairs under the form of pairwise constraints to the constraint model and then it fills in a special data structure, called *constrained matrix*, with constrained variables ; third, all the variables in the matrix are labelled in order to satisfy pairwise coverage. We implemented this process within the PACOGEN tool that is freely available[4]. PACOGEN contains four main components:

1) **FM Analyzer**: this component transforms a feature model into a constraint model, under the form of an abstract syntax tree. As an example of the concrete structure generated within PACOGEN , consider the constraint model generated for the Car Break System FM example of Fig.2 (called CarFM in the following) that is shown below. `FList` corresponds to the features, while `CList` represents the relations, and `solver` corresponds to the call to the constraint solver with a list of parameters.

```
FList=[CAR,ADC,CAB,SA,EA,PP,BACKSENSOR,LRF,FRF],
CList=[and(CAR,[BACKSENSOR]),opt(CAR,[ADC]),
       and(ADC,[CAB],opt(ADC,[PP]),
       or(CAB,[SA,EA]),opt(BACKSENSOR,[LRF,FRF]),
       require(PP,LRF), require(EA,FRF)]

solver(FList,CList,Size,TimeOut,Minimization).
```

   `Size` corresponds to an over-estimation of the size of the matrix. This parameter can be used both for proving that there is no solution under a given threeshold or to refine an existing bound. A default value of 50 is usually a good threeshold to start with, but note that the minimization process will always try to find the smallest size within 1 and the parameter value. `TimeOut` and `Minimization` allow the user to customize the *anytime minimization* step with its own values. Default values can be used for those who have not any requirement on the *test configurations* generation time or the size of the test set.
2) **Consistency checker**: this component evaluates the FM constraints within the *constrained matrix* data

[4] `www.irisa.fr/lande/gotlieb/resources/Pacogen/`

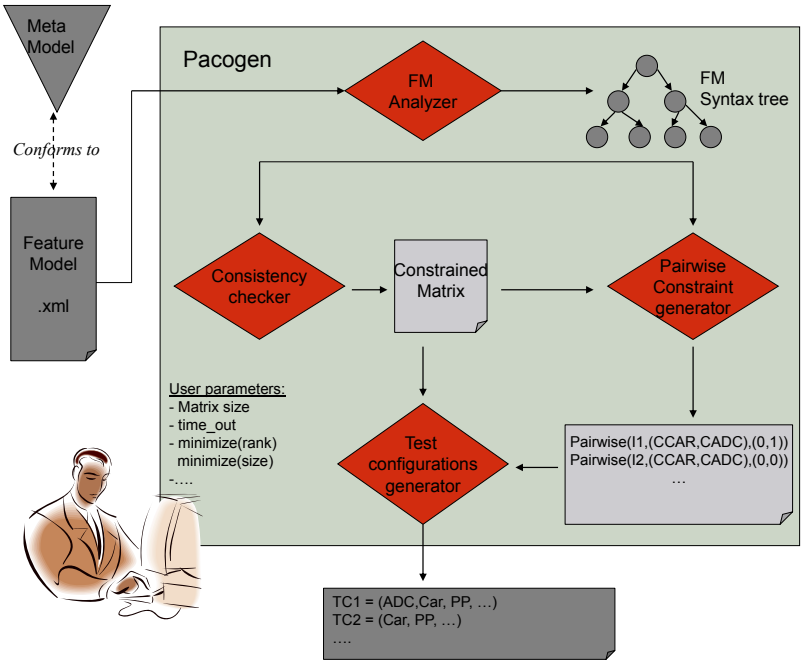

Figure 3. The *test configurations* generation process of PACOGEN

structure. The matrix has size $K \times n$ where K is the number of features while n is an over-estimation of the number of configurations.

3) **Pairwise Constraint Generation**: this component adds the global constraints that enforce pairwise coverage in the test configuration generation process. For the CarFM example, 103 pairwise constraints are generated, such as:

$$\text{PAIRWISE}(I_1, ((CCAR, CADC), (1, 0)),$$

$$\text{PAIRWISE}(I_2, ((CCAR, CADC), (1, 1)),$$

$$\text{PAIRWISE}(I_3, ((CCAR, CCAB), (1, 1)),$$

$$...$$

where

$$CCAR = \begin{pmatrix} CAR_1 \\ CAR_2 \\ . \\ . \\ CAR_n \end{pmatrix} \quad CADC = \begin{pmatrix} ADC_1 \\ ADC_2 \\ . \\ . \\ ADC_n \end{pmatrix} \quad CCAB = ...$$

4) **Pairwise** *test configurations* **generation**: this component calls the constraint solver and the anytime minimization process. It generates the first-found solution under the form of an instantiated constrained matrix, such as the one shown in Fig.4.

Tool validation. In order to validate our tool PACOGEN , we implemented some automated analysis and checked our tool results with published results:

- we computed the total number of valid test configurations with PACOGEN and checked the results with those provided by the SPLOT repository [5]. On all the cases but one, PACOGEN gave us the published results. After investigation, it turned out that the FM for which we found a difference was due to an interpretation difference within the cross-tree constraints of SPLOT. We corrected our constraint model to mimic the semantics of SPLOT cross-tree constraints ;
- on every SPLOT FM, we checked that all pairwise interactions were actually covered by 1) generating all possible pairs of features without cross-tree constraints, and 2) checking that all pairs uncovered by PACOGEN were indeed invalid pairs (i.e., pairs that do not satisfy the cross-tree constraints specified by the FM). The results showed that PACOGEN behave as expected.

VI. EXPERIMENTAL EVALUATION

This section introduces a series of experiments to evaluate Pacogen w.r.t. available techniques and tools. We performed these experiments on the SPLOT repository [5], which contains more than a hundred FMs. SPLOT has been developed to support empirical studies on the performance and scalability of automated techniques for reasoning on FMs. All the experiments were performed on a standard Intel Core i7 CPU 2.67GHz with 4GB memory.

A. Experimental results

The goal of our first experiment was to evaluate the capability of PACOGEN to find the minimum set of *test configurations* that covers all the pairwise interactions among features. Although the approach of Oster *et al.* [8] do not explicitely target this objective (see below in the related

CAR	*ADC*	*CAB*	*SA*	*EA*	*PP*	*BACKSENSOR*	*LRF*	*FRF*
✔	✖	✖	✖	✖	✖	✔	✖	✖
✔	✔	✔	✔	✖	✔	✔	✔	✖
✔	✔	✔	✖	✔	✔	✔	✔	✔
✔	✔	✔	✔	✔	✖	✔	✖	✔
✔	✖	✖	✖	✖	✖	✔	✔	✔

Figure 4. A constrained matrix solution covering all pairwise interactions of the CarFM

Feature models	#F	#N	Oster *et al.* 2010	PACOGEN	Gain
crisis management	17	–	15	5	-66.7%
Smart Home	35	1 048 576	11	8	-27.3%
Inventory	37	2 028 096	12	15	+20%
Sienna	38	2 520	24	20	-16.7%
Web portal	43	2 120 800	26	16	-38.5%
Doc generation	44	$5.57.10^7$	18	17	-5.5%
Arcade Game	61	$3.3.10^9$	25	14	-44.0%
Model Transformation	88	$1.65.10^{13}$	40	26	-35.0%
Coche Ecologico	94	$2.32.10^7$	114	92	-19.3%
Electronic shopping	287	$2.26.10^{49}$	62	37	-40.3%
Average	74.4	–	34.7	25	-29.2 %

Table I
NUMBER OF TEST CONFIGURATIONS ENFORCING PAIRWISE

work section), its greedy algorithm also aims at minimizing the number of configurations and it is currently considered as the most efficient approach for this task [7]. Tab.I shows the experimental results we got with PACOGEN and how it compares with the results of [8] on ten FMs from SPLOT. Tab.I contains the number of features of each FM (#F), the number of valid configurations (#N) to characterize the size of the search space, the results of both [8] and PACOGEN in terms of number of configurations to cover all the valid pairwise interactions between features, and finally the gain obtained with PACOGEN . Negative gains indicate that PACOGEN proposes less configurations than other approaches, which was the main objective of our work. The goal of this experiment was to evaluate the minimization process and thus we did not try to optimize on the test configuration generation time. In all the cases but one, the generation time required by PACOGEN was less than a few minutes. On *Electronic shopping*, PACOGEN took 12 hours to get an optimal result which remains an acceptable amount of time for a one-shoot generation. The results indicate that, in average, our approach requires 28.2% lesser configurations than the [8] results. In real-world applications, testing a configuration may be costly because it may require to build the configuration with physical components, to set up a dedicated testing environment and to execute a time-consuming result evaluation process. Depending on the cost and time required to test a configuration, the improvement we got with PACOGEN may be crucial. Still, there is an anormal result for *Inventory* where PACOGEN selected more configurations of the results given in [8]. We manually double-checked the results of PACOGEN without finding how to get a smaller value than 15 for covering all the valid pairwise interactions on this example. Apart from this example, our results show that the PACOGEN approach is well-suited to find the minimum number of configurations that cover all the pairwise interactions of features in a FM.

The goal of our second experiment was to evaluate the anytime minimization process of PACOGEN , which permits to find the best compromise between generation time and result quality. We conducted a large-scale experiment over 67 FMs extracted from SPLOT that is reported in Fig.5. We launched PACOGEN to compute the minimum number of test configurations by allocating 2 seconds, 10 seconds, 3 min and 15 min to the anytime minimization process. First, the curve shows that PACOGEN can provide an optimal solution (a minimum set of test configurations covering all pairwise feature interactions) in less than 15min in all the cases. Second, the anytime minimization process is interesting to refine the set of configurations in a number of cases. For almost half of the FMs, the anytime minisation process has been useful. Third, the number of test configurations that is refined during this process is usally very small ; there are ony two cases where the number of configurations is refined of more than 5 configurations. These results show that PACOGEN implements a usefull anytime minimization process that has potential to help finding good compromises between number of configurations and generation time.

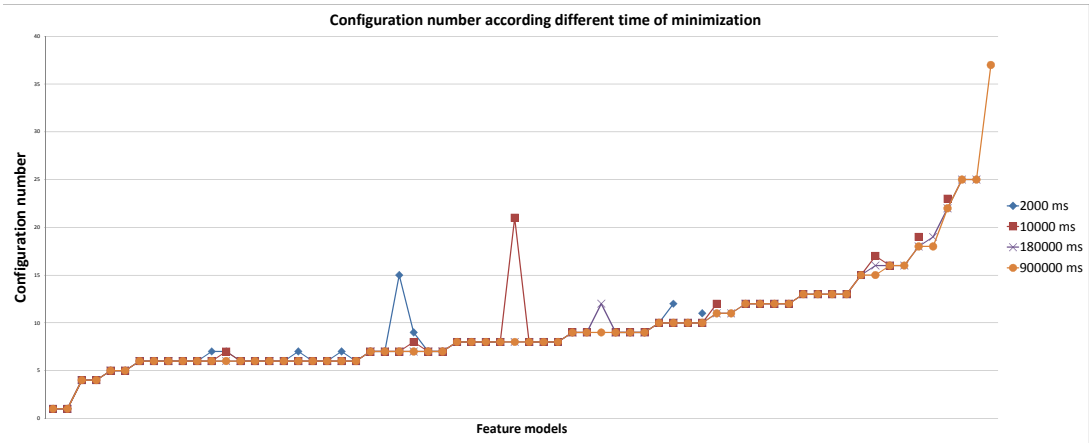

Figure 5. Number of test configurations using various time-allocations

B. Threats to validity

External threats to validity lie on the source of the empirical data. We have selected our subjects in the SPLOT repository, which allowed us to experiment over a large set of models and to compare our results to Oster's. However, SPLOT is fed with models that mostly come from academia and might not perfectly reflect industrial usage of feature models in terms of number of features and size of the variability space. The threats to internal validity come from biases that can be introduced by errors in the prototypes we used to perform our experiments. In order to limit these threats, we performed exhaustive unit testing on the transformations used PACOGEN in addition to the tests mentionned in Sec. V.

VII. Related work

Benavides et al. [1] surveyed the various constraint models that have been proposed in the literature. In particular, Constraint Satisfaction Problems (CSPs) have been used to encode automated analyses such as filtering the configurations or checking the consistency of the feature model (e.g., dead feature, valid product). Recently, Karatas et al. proposed the usage of global constraints for handling with extended feature models [19], [13]. Our approach shares similarities with these works, as our model is also based on finite domain constraints filtering. However, it differs on its goals, our approach being targeted to generate pairwise covering set of configurations, and on its underlying technology. A key-point of our approach is the development of a new global constraint (i.e., pairwise) and the usage of a time-contract optimization labelling procedure, that both are unavailable elsewhere. Another trend in Software Product Line Testing is related to the usage of SAT-solving to generate test configurations [1]. The recent work of Uzuncaova et al. [20] showed that an extended SAT solver could be used to incrementally generate test inputs for each product in a product line. However, one advantage of Constraint Programming over SAT-solving is its capability to solve optimization problems, instead of pure satisfiability problems [21]. The problem we attacked in this paper is clearly an optimization problem (i.e., to generate the smallest set of configurations ensuring the coverage of the pairwise criterion over the possible feature values).

The main related work that share our testing objectives are the proposals by Perrouin [7] and by Oster [8]. Perrouin et al. transform feature models in Alloy to select configurations that are valid with respect to the initial model. This approach was the first work to propose the adaptation of pairwise testing in the context of feature modelling, but it faces scalability issues. First because it uses a *generate-and-test* approach to select the valid pairs that must be covered (generate all possible pairs, then filter all invalid pairs). Second, the Alloy model needs to be transformed in CNF formulae before it can be solved by SAT solvers. This additional step increases the scalability problems. Our approach limits these problems by adopting a *test-and-generate* approach for identifying valid pairs and by transforming the feature model directly into constraint problem over finite domains. Oster et al. deal with dependencies between features by a flattening transformation over the feature model and the adaptation of the AETG [6] algorithm in order to consider dependencies between features. The authors do not provide a detailed description of their transformation process, but their experimental results are good. Their approach is efficient and effective: it can select a small number of configurations over large feature models in very reasonable times. These work can be seen as an extension of those of Cohen et al. [22] where a SAT-solver is used to build a greedy approach tackling the problem of dependencies within highly-configurable systems. However, nor this approach or Olster's or Perrouin's work tackle the selection of the minimum number of configurations to cover pairwise interactions.

In our work, we propose a deterministic (as opposed to "greedy") approach able to compute the optimal number of configurations. If necessary, our approach can be relaxed by using a time-contract process to provide a near-optimal value. In addition, our work is the first to perform a large experiment on a significant set of feature models .

VIII. Conclusions and Perspectives

In this work we proposed PACOGEN a tool for generating a set of test configurations from FM that covers all the valid pairwise interaction among features. The tool offers an anytime minimization process that allows the user to define an objective stipulating the amount of time he allocates to the generation. Based on Constraint Programming techniques including a dedicated branch-and-bound algorithm, the tool returns the minimum number of configurationd found in the allocated time. We conducted a large-scale experiment over 67 FMs extracted from the SPLOT repository [5] showing that 1) PACOGEN overcomes the State-Of-the-Art technique of [8] in terms of number of configurations and 2) the anytime minimization process is useful in practice. More precisely, PACOGEN generated 29.2% less configurations than [8] in average and generated the optimal number of configurations for the 67 FMs in less than 15 minutes.

On the basis of these promising results we plan first to continue investigating constraint-based exploration of variability spaces in future work. In particular, we plan to include the values of attributes associated to features in the selection process. For example, if features are associated to execution times, or energy consumption, it is possible to reuse our constraint model to optimize quality of service requirements. Second, we will investigate extensions of our work to consider the relationships between several variability spaces. For example, in component-based systems, the configuration of one component can have an impact on the way another component can be configured. Thus, when sampling the configuration space for the first component it is necessary to consider the dependencies with the configuration space of the second component. This should prevent selecting only configurations that force the second component to be in a narrow area of its configuration space.

Thanks

We are grateful to Pierre Flener and Joseph Scott for some valuable comments and suggestions on an earlier version of this article.

References

[1] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models: A detailed literature review," *Information Systems*, no. 35, pp. 615–636, 2010.

[2] D. Beuche, H. Papajewski, and W. Schrder-Preikschat, "Variability management with feature models," *Science of Computer Programming*, vol. 53, no. 3, pp. 333 – 352, 2004.

[3] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. R. Cortés, "Automated diagnosis of feature model configurations," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1094–1107, 2010.

[4] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration using feature models," in *Software Product Line Conference (SPLC'04)*, 2004, pp. 266–283.

[5] M. Mendonça, M. Branco, and D. D. Cowan, 2009. [Online]. Available: http://www.splot-research.org/

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437 – 444, 1997.

[7] 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 *International Conference on Software Testing (ICST'10)*, Paris, France, 2010.

[8] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *Software Product Line Conference (SPLC'10)*, 2010.

[9] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel, "Reconciling automation and flexibility in product derivation," in *Software Product Line Conference (SPLC'08)*. Limerick, Ireland: IEEE Computer Society, Sep. 2008, pp. 339–348.

[10] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, vol. 51, no. 2, pp. 456–479, 2007.

[11] J. White, B. Dougherty, D. C. Schmidt, and D. Benavides, "Automated reasoning for multi-step feature model configuration problems," in *Software Product Line Conference (SPLC'09)*, 2009, pp. 11–20.

[12] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *International Symposium on Software Testing and Analysis (ISSTA'07)*, London, UK, 2007, pp. 129–139.

[13] A. S. Karatas, H. Oguztüzün, and A. H. Dogru, "Global constraints on feature models," in *Principles and Practice of Constraint Programming - CP 2010 - CP 2010, St. Andrews, Scotland, UK, Sep. 6-10, 2010. LNCS 6308*, 2010, pp. 537–551.

[14] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," in *Proc. of AAAI'94*, 1994, pp. 362–367.

[15] P. V. Hentenryck and Y. Deville, "The cardinality operator: A new logical connective for constraint logic programming," in *International Conference on Logic Programming (ICLP'91)*, 1991, pp. 745–759.

[16] B. Hnich, S. Prestwich, E. Selensky, and B. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, pp. 199–219, 2006, 10.1007/s10601-006-7094-9. [Online]. Available: http://dx.doi.org/10.1007/s10601-006-7094-9

[17] C. Salinesi, D. Diaz, O. Djebbi, R. Mazo, and C. Rolland, "Exploiting the versatility of constraint programming over finite domains to integrate product line models," in *17th IEEE Int. Requirements Engineering Conference (RE'09)*, 2009, pp. 375–376.

[18] S. Zilberstein, "Using anytime algorithms in intelligent systems," *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996.

[19] A. S. Karatas, H. Oguztüzün, and A. H. Dogru, "Mapping extended feature models to constraint logic programming over finite domains," in *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. - LNCS 6287*, 2010, pp. 286–299.

[20] E. Uzuncaova, S. Khurshid, and D. Batory, "Incremental test generation for software product lines," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 309–322, May 2010. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.30

[21] R. Nieuwenhuis and A. Oliveras, "On sat modulo theories and optimization problems," in *In Theory and Applications of Satisfiability Testing (SAT), LNCS 4121*. Springer, 2006, pp. 156–169.

[22] 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*, vol. 34, no. 5, pp. 633–650, 2008.