Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming

Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, Benoit Baudry

▶ **To cite this version:**

Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, Benoit Baudry. Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming. Information and Software Technology, 2016, 71, pp.129-146. hal-01352831

HAL Id: hal-01352831

https://inria.hal.science/hal-01352831v1

Submitted on 10 Aug 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimal Minimisation of Pairwise-covering Test Configurations Using Constraint Programming

Aymeric Hervieu, INRIA Rennes Bretagne Atlantique, France
Dusica Marijan, Certus Software V&V Center, SIMULA Research Laboratory, Lysaker, Norway
Arnaud Gotlieb, Certus Software V&V Center, SIMULA Research Laboratory, Lysaker, Norway
Benoit Baudry, INRIA Rennes Bretagne Atlantique, France

Context: Testing highly-configurable software systems is challenging due to a large number of test configurations that have to be carefully selected in order to reduce the testing effort as much as possible, while maintaining high software quality. Finding the smallest set of valid test configurations that ensure sufficient coverage of the system's feature interactions is thus the objective of validation engineers, especially when the execution of test configurations is costly or time-consuming. However, this problem is NP-hard in general and approximation algorithms have often been used to address it in practice.

Objective: In this paper, we explore an alternative approach based on constraint programming that will allow engineers to increase the effectiveness of configuration testing while keeping the number of configurations as low as possible.

Method: Our approach consists in using a (time-aware) minimisation algorithm based on constraint programming. Given the amount of time, our solution generates a minimised set of valid test configurations that ensure coverage of all pairs of feature values (a.k.a. pairwise coverage). The approach has been implemented in a tool called PACOGEN.

Results: PACOGEN was evaluated on 224 feature models in comparison with the two existing tools that are based on a greedy algorithm. For 79% of 224 feature models, PACOGEN generated up to 60% fewer test configurations than the competitor tools. We further evaluated PACOGEN in the case study of large industrial highly-configurable video conferencing software with a feature model of 169 features, and found 60% fewer configurations compared with the manual approach followed by test engineers. The set of test configurations generated by PACOGEN decreased the time required by test engineers in manual test configuration by 85%, increasing the feature-pairs coverage at the same time.

Conclusion: Extensive evaluation concluded that optimal minimisation of pairwise-covering test configurations is efficiently addressed using constraint programming techniques.

Keywords: Configuration testing, configurable software systems, constraints programming

1. INTRODUCTION

Motivations. Modern software systems often contain a base functionality common to a set of related products and a number of specific functionalities and features that introduce variability within a set of software configurations. *Highly-configurable systems* are thus those systems that can be configured in many different ways and adapted to diverse user needs by a configuration process. Testing highly-configurable systems is challenging due to a large number of possible test configurations that must be minimised in order to reduce the testing effort, while not compromising the effectiveness of the testing process. If validation engineers ignore commonalities within the products when selecting test configurations, the overall testing effort/time usually exceeds the available testing time. A practical approach to cope with this problem is to abruptly cut off the number of test configurations, given the available testing time. This approach may cause missing serious software faults that are due to the interaction of specific software features. Moreover, in highly-configurable systems, particular failures can be detected only by testing specific software configurations [Qu et al. 2008], as different combinations of features lead to different functionality/products.

The MVPF problem. In a highly-configurable software setting, adopting a systematic testing methodology is of paramount importance for ensuring software quality. This means adopting well-recognised testing criteria to build a software testing pro-

cess based on rational choices. In our work, we aim at selecting the *smallest* subset of valid test configurations ensuring that each possible pair of feature values is covered at least once. We call this problem the *Minimum Valid Pairwise-Feature-values coverage (MVPF) problem*. Through a specific constraint optimisation model, MVPF problem aims at finding true optima (not an approximation to the optimal solution) of a configuration sampling problem, which is one differentiating characteristic of our approach over most of the existing work. Furthermore, MVPF problem incorporates the requirement for the validity of test configurations, satisfying the constraints existing among the features (including physical and business constraints).

Solving the MVPF problem is regarded as a minimum requirement when testing highly-configurable systems. Indeed, as observed in several studies [Kuhn et al. 2004; Klaib et al. 2008], having covered all 2-way feature interactions increases the confidence of validation engineers in the quality of tested products (assuring, at least, against 2-way errors). Solving the MVPF problem also means finding a set (not necessarily unique!) of valid test configurations of minimum cardinality. The rationale is to minimise the overall test effort by minimizing the number of considered test configurations, while maintaining their ability to detect failures.

NP-hardness of the MVPF problem. Unfortunately, as soon as constraints are involved among the features, the problem of generating exactly the minimum number of test configurations that cover all pairs of feature values is NP-hard. This is shown by the reduction to the boolean satisfiability problem [Batory 2005]. There are several classes of approaches to such problems [Cohen et al. 2007b]. *Greedy algorithms* [Cohen et al. 2008; Oster et al. 2010; Perrouin et al. 2010; Johansen et al. 2012b], consider one by one the configurations that cover the most pairs, checking their validity. This means that once a configuration has been included in the current set of valid configurations, it cannot be deleted and replaced by another configuration covering fewer but different pairs. Being approximations, these algorithms can only seldom reach the true minimum number of valid test configurations. In domains where test configurations take a long time to execute, finding an optimal number of test configurations becomes essential to decrease the overall testing effort. *Meta-heuristic searches* [Cohen et al. 2003], as more sophisticated heuristics, can often find good solutions with less computational effort than simple heuristics, but still not guaranteeing a globally optimal solution. *Mathematical construction* are fast approaches [Williams 2000], but they do not handle arbitrary feature constraints, and often they are not well suited for a large number of parameters and bigger covering arrays [Bryce et al. 2005]. Moreover, contemporary mathematical constructions do not handle arbitrary feature constraints well.

Solving an industrial problem. The MVPF problem was identified in the domain of large highly-configurable networking software by Cisco Systems Norway, an industrial partner of the Certus Centre[1]. Testing highly-configurable networking systems in a continuous integration environment requires not only selecting the relevant configurations to test (configurations that underwent changes), but also fitting the testing process for a limited testing time. For our partner, the ability to control (at any stage of the testing process) the time needed to complete testing is mandatory. As test configuration execution often requires manual hardware setup, which may take up to 30 minutes, finding an optimal set of valid test configurations is of paramount importance.

[1]Certus Centre, hosted by SIMULA Research Laboratory, is a research-based innovation center aiming at developing methods and tools in the domain of validation and verification of software-intensive systems, in collaboration with industrial and public administration partners.

The proposed solution. To deal with this problem, we propose a novel Constraint Programming (CP) approach that manages test configurations from feature models. Feature modelling is a common approach for representing variability within a software product line, while Constraint Programming (CP) is a well-known programming paradigm dedicated to the resolution of hard combinatorial problems. Given the available time, our approach seeks for the minimum set of valid configurations solving the MVPF problem. A special data structure composed of finite-domain variables is used to encode a set of valid configurations. We introduce a dedicated combinatorial constraint (called $pairwise$ *global constraint*) to enforce feature pairwise coverage over the data structure, while other logical constraints are used to capture hierarchical and cross-tree links between features. We also introduce special heuristics to order the feature pairs and to select the variable to enumerate first (*Pair-ordering and Variable-selection heuristics*). Finally, a constraint optimisation procedure, called *branch and bound*, is used to find the minimum number of test configurations. This characteristic allows continuing the constraint solving process after the first solution has been found, until a timeout occurs, which is the second differentiating characteristics of our approach over the most of the existing work. Our approach can find a candidate set of minimum cardinality and prove that there is no smaller set. However, as the problem is NP-hard, time spent in constraint optimisation should be balanced with the time required to test a single configuration. This approach proved to be successful in solving the MVPF problem in practice.

Implementation and experimental results. We implemented the approach in a freely available tool called PACOGEN[2]. The tool results from a two man-year development effort and incorporates subtle optimisations that are discussed in detail in Section 4 and evaluated in Section 6. We performed the experimental comparative study with the two existing greedy approaches that aim at solving the same problem, MosoPolite tool [Oster et al. 2010] and SPLCAT tool [Johansen et al. 2011; 2012b]. The comparison was performed using a large benchmark consisting of 224 feature models from SPLOT [Mendonça et al. 2009]. Our results show that PACOGEN produces up to 60% fewer configurations on average, respecting the MVPF problem, for 79% of the benchmark feature models for one approach, and up to 42% fewer test configurations for 6 of 7 feature models for the another approach. Furthermore, we applied and validated PACOGEN on an industrial case study provided by Cisco. The feature model of a Cisco video software system consists of 169 features and more than 10^9 possible configurations (valid and invalid). The experimental results show that PACOGEN produces 60% fewer configurations, compared with the manual approach followed by Cisco, while substantially increasing 2-way feature coverage. The automatically generated set of configurations decreased the time required by validation engineers in manual configuration specification by 85%.

Contributions. The idea of using CP to solve the MVPF problem was introduced by the authors in [Hervieu et al. 2011]. We have further enhanced the initial idea by introducing a dedicated algorithm for filtering invalid pairs (Section 4.3), and by proposing new search heuristics (Section 4.4). We have also developed a new version of PACOGEN that significantly improved the experimental results presented in [Hervieu et al. 2011]. We have applied and verified PACOGEN in a new industrial context. In summary, this paper makes the following contributions:

[2]http://hervieu.info/pacogen.

(1) Defining combinatorial configuration sampling in its full generality as a problem amenable to standard constraint optimisation techniques, and finding a true minimum solution of the sampling problem;
(2) Verification in an industrial context, showing that favouring sample size over sample generation time has practical relevance;
(3) Enhanced time-aware method to generate the minimum-cardinality set of test configurations (addressing the MVPF problem). The method incorporates common CP techniques such as constraint propagation and filtering, but also dedicated approaches such as global constraint design (*pairwise*) and search heuristics (*Pairs-ordering and Variable-selection heuristics*);
(4) Industry-strength prototype implementation of the proposed method (PACOGEN);
(5) Extensive experimental evaluation and comparison with two state-of-the-art approaches;

The rest of the paper is structured as follows: Section 2 presents a motivating example for test configuration selection in highly-configurable software systems. Section 3 describes the theoretical concepts and notations used in our approach, and gives an overview of the related work. Section 4 details the CP model used for generating the minimum set of valid test configurations. Section 5 describes PACOGEN implementation. Section 6 contains an extensive experimental evaluation of the proposed approach, while Section 7 concludes the paper.

2. A MOTIVATING EXAMPLE

To illustrate the challenges that appear in testing highly-configurable software systems in practice, we present the case study of video-conferencing (VC) systems, provided by our industrial partner Cisco Systems.

We describe the industrial software system under study, C90 codec, as illustrated in Figure 1. C90 comprises core functionality (providing basic video-conferencing), common to all VC variants, and a set of features that can be used to configure the software according to users' specific needs, requirements or preferences. As a highly configurable system, C90 offers a very wide range of options to configure the VC software for its purpose, what makes it difficult for validation engineers to ensure that all configurations are working correctly. According to the variability model built together with system domain experts, C90 consists of 169 software features with mutually exclusive and inclusive relations. For example, the $touch\ panel$ feature of the C90 video system allows users to interact with the video system through the physical contact on the panel. On the contrary, the $remote\ control$ feature allows users to operate the video system remotely, through the remote control device. There is a design constraint that forbids coexistence of these two software features in one video system. This constraint is modelled as exclude feature relation. Similarly, C90 cannot be configured for $1080p60\ video\ resolution$ without the $high\ definition$ feature support. This constraint is modelled as require feature relation. The constraints among features must be respected when specifying configurations, in order to select only the valid ones.

VC software is configured in two stages. First, the product-line engineers select features for a final product based on the product type or product price. The example of a variation point at this early configuration stage is the $video\ conferencing\ call\ type$. A product of more expensive series supports the $multi\text{-}site\ calls$ (more than two video units in a call), whereas a product from cheaper series supports only the $point\text{-}to\text{-}point\ calls$. Second, after the product has been delivered, users are offered a number of parameters, each with a range of values, to configure particular software features of the product. For example, a user can choose a video input format or a video resolution

Fig. 1. Software variability of the video conferencing system C90

for the supported video codec. We refer to *features* as both the VC core functionalities and user-configurable software parameters.

In our partner's testing practice, a typical scenario for a validation engineer is to manually select and combine features into the valid test configurations and execute the test cases for these configurations in a continuous integration framework. In this process, the validity of the generated configurations highly depends on the engineer's interpretation and understanding of documentation. If feature dependencies have not been taken into account and there are failing tests in execution, it is difficult to interpret the results; the failures can be due to faulty software, but also due to an invalid combination of features. Another challenge that validation engineers face in manual test configuration is avoiding test case redundancy. Good testing practice aspires to cover as much software functionality under test as possible with as few tests as possible. We have observed that the set of manually specified test configurations contains repeating configurations, as the set was specified some time ago and was not maintained regularly. Similarly, if validation engineers do not perform coverage analysis for the generated test configurations, some test configurations will appear more than once in the test set, while some others will not exist in the set at all. Furthermore, test configurations should be able to detect failures that are specific for a particular combination of software features (a.k.a. feature interaction problem) [Qu et al. 2008]. For example, a validation engineer detected that a VC configuration with SIP call protocol and 128 call rate leads to a failure in establishing a video call, while the configurations with all other combinations of call protocols and call rates work correctly. This particular failure can be detected only by testing the interaction between the $SIP\ call\ protocol$ and the 128 call rate features. Additionally, testing in a continuous integration environment imposes limited testing time. Specifying valid test configurations manually is a time consuming task, as it requires selecting and combining many software parameters. Defining a valid VC configuration takes on average one man-hour. The execution of tests for one configuration takes up to one hour on average and testing all configurations must be completed within 16 hours (over-night). To comply with these realistic timing constraints, an automated technique that could help generate the minimum set of valid test configurations is of core interest for our partner.

3. BACKGROUND

In this section we describe the theoretical concepts and notations used to address the MVPF problem. We also give an overview of the related approaches.

3.1. Feature models for specifying configuration variability

Although initially proposed to represent products in software product lines [Kang et al. 1990], feature models can also specify variability and commonalities in software configurations. In this case, feature models are used to capture: (i) configurable software parameters (features), and (ii) dependencies between features. There are various variability modelling notations [Pohl et al. 2005] and feature model notations [Czarnecki et al. 2005], [Eriksson et al. 2005] proposed in the literature, but we selected a formalism that extends the metamodel of [Perrouin et al. 2008], based on the Free Feature Diagram [Schobbens et al. 2007]:

— A *feature model (FM)* is a hierarchical organisation of a set of features;
— A *feature* is an abstraction that represents a composition element or a configuration parameter;
— A *configuration* is a selection of features from an FM;
— A *valid configuration* is a configuration that satisfies all constraints of an FM;
— A *constraint* is a relation between two or more features in an FM. We distinguish between several types of constraints, such as hierarchical constraints, cross-tree constraints and CNF constraints:
  (1) *A hierarchical constraint* is a relation between a father and its children features based on the following operators $NT = \{\text{AND}, \text{OR}, \text{XOR}, \text{OPT}, \text{CARD}\}$;
  (2) *A cross-tree constraint* is a binary relation between any pair of features in an FM based on the operators $GCT = \{\text{REQUIRE}, \text{MUTEX}\}$;
  (3) *A CNF constraint* is a non-binary relation among any subset of features that can be expressed as a boolean formula in Conjunctive Normal Form (CNF).

Figure 2 represents a partial feature model of the VC software from our case study. The model specifies that the VC software supports making calls, which can be either $P2P$ or $Multisite$ calls. For the $Multisite$ calls, possible settings are either $3x1024x576max$ or $3x720p30max$ or $1080p30$–$720p60max$. Optionally, the VC software can support the $720p60$ premium resolution, the $1080p30$ premium resolution, or both resolutions. The configuration that supports the $3x720p30max$ or the $1080p30$–$720p60max$ resolution for the $multisite$ calls must have the $Premium\ resolution$ feature.

For the feature model shown in Figure 2, the set of features $S = \{VCS, Call, Multisite, 3x1024x576max\}$ is a valid configuration, but the configuration $S = \{VCS, Call, Multisite, 3x720p30max\}$ is invalid, because the constraint between the $3x720p30\ max$ and the $Premium\ resolution$ features is not satisfied.

For the sake of simplicity, we will consider only feature models that have homogeneous nodes. This means that all children features are related to their father feature using a single relation. When a feature model is not homogeneous, it can be easily transformed by introducing an intermediate feature. For example, the feature model shown in Figure 2 can be transformed with an intermediate feature, a mandatory child of the VCS, so that the root node becomes homogeneous (currently it has two children related to two distinct operators, namely mandatory and optional).

3.2. Pairwise testing of highly-configurable software

3.2.1. Combinatorial Interaction Testing. Combinatorial Interaction Testing (CIT) is an effective technique to test highly-configurable software that has many dependencies be-

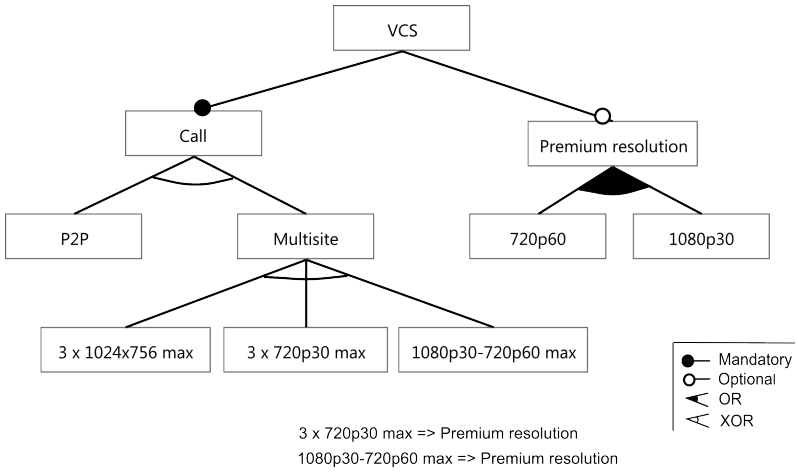

Fig. 2. Simple feature diagram for a video conferencing software

tween configuration parameters. The effectiveness of CIT is based on the observation that software failures are often due to interactions between only few (N) software parameters [Kuhn et al. 2004], [Bell and Vouk 2005], [Grindal et al. 2005]. The tests generated using CIT, therefore, cover all N-way combinations of input parameters and are able to detect faults that arise from the interactions of N or fewer components. CIT has been successfully applied to various problems [Czarnecki and Eisenecker 2000], including large distributed systems [Kuhn et al. 2004], GUI testing [Memon and Soffa 2003], and fault localisation [Yilmaz et al. 2006]. More recently combinatorial testing has been applied to a highly configurable system [Johansen et al. 2012a]. Pairwise testing is a special case of CIT, widely used by practitioners, that subsets a test input domain into the set that covers every combination of values for each pair of input parameters. Introduced by Cohen et al. [Cohen et al. 1997], pairwise testing aims at detecting failures triggered by interactions between two parameters (2-way tests). It is known that a pairwise-covering test suite can have lower error detection capabilities compared with the test suites with higher interaction strength (3-way, 4-way). However, 3- and 4-way CIT techniques entail high computational complexity and produce a larger number of test configurations. In some contexts, this can impose limitations on wider practical application of such techniques [Batory 2005], [Cohen et al. 1997], [Cohen et al. 2008] and if there are no requirements to test the interaction coverage higher than pairwise, pairwise coverage is a good compromise [Kuhn et al. 2004; Klaib et al. 2008].

3.2.2. Testing highly-configurable systems. In the context of highly-configurable software testing, CIT can be used to generate a set of configurations that cover all pairs of software features. For example, in the VC software from our industrial case study, the configuration [$Call$, $\neg P2P$, $Multisite$, $\neg 3x1024p576max$, $3x720p30max$, $\neg 1080p30$–$720p60$, $Premium\ resolution$, $720p60$, $\neg 1080p30$] covers the interactions between 30 pairs of values: ($Call$, $Multisite$); ($Call$, $\neg P2P$); ($Call$, $3x720p30max$), to present a few.

Pairwise testing with feature models uses a mathematical structure called a binary covering array, which is $n \times k$ (0,1)-matrix M, where n denotes the number of test configurations and k denotes the number of features of a feature model. The matrix M

is defined as follows:

$$M(i;j) = \begin{cases} 1 & \text{iff feature } j \text{ belongs to configuration } i \\ 0 & \text{otherwise.} \end{cases}$$

With no constraints among features, the minimum number of rows n in the covering array to cover all combinations between features is the least positive integer such that:

$$\left(\frac{n-1}{\lceil \frac{n}{2} \rceil} \right) \geq k \tag{1}$$

where k is the number of features [Lawrence et al. 2011]. When applied to configurable software with constrained features, pairwise testing faces a challenge to determine the value for n. For example, the pair ($3x720p30$ = selected, $Premium\ resolution$ = unselected) is not a valid pair according to the feature model in Figure 2.

3.2.3. Existing approaches for CIT. Researchers have proposed various approaches for generating test configurations that cover N-wise interactions among the set of variables. Many of the proposed approaches in the literature are greedy methods [Bryce et al. 2005; Bryce and Colbourn 2009]. AETG [Cohen et al. 1997], based on a greedy algorithm, is able to efficiently generate N-wise covering arrays for a set of parameters, taking their values in finite domains. However, greedy approaches do not guarantee reaching the true minimum number of test configurations with pairwise coverage. Every time a new pair of parameter values is considered, the algorithm seeks the pair among already selected test configurations. If found, the pair is just discarded, while if not, a new test configuration covering that pair is added. Note that once a configuration is added to the current set of configurations, it cannot be withdrawn from the set, in order to use some other configuration covering that pair. Greedy techniques have also been combined with heuristic search techniques to generate more accurate results [Bryce and Colbourn 2007]. These techniques start from a given test suite and apply calculations until all combinations have been covered. They can often generate fewer test configurations than greedy methods, but usually have longer runtime. TConfig and CTS, as mathematical methods, have been proposed for generating covering arrays [Hartman 2005; Williams 2002]. However, these approaches do not handle constraints among parameters. One of the approaches dealing with this problem, was proposed by Hnich [Hnich et al. 2006], using constraint programming techniques. However, this approach suffers from two drawbacks. Firstly, the notion of side constraints corresponding to hierarchical and cross-tree constraints is mentioned as an extension, but neither properly introduced nor evaluated. Secondly, this approach faces scalability problems for bigger CIT problems. Handling CIT problems with constraints has been extensively studied in [Cohen et al. 2007a; 2007b; 2008]. Further, CTE-XL tool [3] allows users to generate pairwise and triple-wise covering test sets from a category-partition of the input domain. CTE-XL handles additional constraints over the input parameters, but only in a passive way, by checking afterwards if generated test cases satisfy the constraints or not. On the contrary, several authors have proposed the tools handling constraints in an active way, either by solving them or using their evaluation to compute values of a fitness function. IPOG is a tool for generating covering arrays based on the 0-1 linear programming model [Lei et al. 2008], while CASA is an approach to constrained interaction testing based on simulated annealing [Garvin et al. 2011]. Authors proposed an approach for multi-objective optimal test suite computation based on integer linear programming [Lopez-Herrejon et al. 2013]

[3] http://www.berner-mattner.com/en/berner-mattner-home/products/cte-xl.

More recently, Perrouin proposed transforming feature models into Alloy declarative programs, in order to select only valid configurations, with respect to the initial model [Perrouin et al. 2010]. This is one of the first approaches proposing the usage of pairwise testing in the context of feature modelling. However, the approach exhibited insurmountable scalability issues [Perrouin et al. 2012]. Firstly, using a generate-and-test approach to select uncovered valid pairs involves too many repeated calls to the underlying constraint solver (i.e., SAT-solver). Secondly, transforming Alloy models into boolean formula under CNF provokes a combinatorial explosion on some models. The approach by Oster [Oster et al. 2010] does not rely on SAT solver to generate pairwise-covering test configurations. Instead, it uses greedy and ad-hoc algorithm, based on a selection of test configurations, that maximises the number of valid pairs within each configuration. In [Oster et al. 2010], the author provides the experimental results showing that the approach is effective in selecting a small number of configurations over large feature models in a reasonable time. MosoPolite is an industry-strength implementation of Oster's approach exploited in the Automotive domain [Steffens et al. 2012]. Another tool for constructing N-way combinatorial test sets is ACTS [4], which is based on a greedy approximation algorithm. Johansen recently proposed several optimisations to the classical greedy approach to efficiently generate test configurations with 1-3-way coverage for large feature models [Johansen et al. 2012b], with a relatively low running time. The approach is supported by SPLCAT, which is able to generate a set of pairwise-covering test configurations for the Linux kernel FM, containing more than 6000 features. According to our knowledge, Johansen's method and tool is one of today's most advanced approaches available for generating valid test configurations meeting N-wise criteria.

However, none of these approaches guarantee the selection of the minimum number of configurations to cover N-wise interactions. Given a feature model, addressing the MVPF problem involves two distinct problems: (i) finding the true minimum number of configurations, and (ii) making sure that every configuration is indeed valid. In [Johansen et al. 2011], Johansen reports that solving the first problem is equivalent to the Set Cover problem, known to be NP-hard. Although Lawrence suggests that restricting to pairwise interactions might be less complex [Lawrence et al. 2011], we are not aware of any tractable algorithm able to solve it. There is an approximation algorithm for this NP-hard problem, known as Chavtal algorithm [Chvatal 1979]. The algorithm prioritises configurations by starting from the ones covering the most of uncovered pairs, until all pairs are covered. However, in the presence of constraints among features, this algorithm does not necessarily generate valid configurations. In fact, relating to the second problem mentioned above, generating a valid configuration from a feature model is equivalent to general SAT-solving [Batory 2005], which is known to be NP-hard as well. Even if Mendonca reported that SAT-based analysis of realistic feature models is feasible in practice [Mendonca et al. 2009], we still have to solve as many SAT-instances as there are configurations in a pairwise covering set of test configurations. This means that solving the MVPF problem not only involves solving the optimisation problem, whose generalisation is known to be NP-hard, but also a number of satisfiability problems.

3.3. Constraint programming

Constraint programming (CP) is a paradigm describing the relations between variables in the form of constraints. Modelling and solving a constraint problem is based on a three-step scheme: (i) defining variables and specifying their variation domain, (ii) identifying constraints between the variables, and (iii) solving the constraint system

[4] http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.

using well-defined processes such as *constraint propagation* and *labelling*. Constraints represent relations among variables and are classified as logical ($and, or, not, \ldots$), arithmetic ($A < B, B = C, \ldots$), global or symbolic (***AllDiff, Element,*** $\ldots$), etc. A global constraint is a relation defined by an interface (operator's name and constrained variables), a filtering algorithm, and awakening conditions. A solution of a CP system is a full assignment of variables to the values within their domain, satisfying all constraints. Constraint propagation is an efficient process[5] ensuring that some invalid combinations of variable values are discarded from its domain. It is only a partial test for satisfiability, as it does not guarantee that all invalid combinations will be filtered out from the domains. This is a reason why constraint propagation is interleaved with a labelling process, which can instantiate all possible variables to the values from their domain.

The reason why we chose CP to address the MVPF problem is that it allows defining new special-purpose constraints (global constraints) for fine-tuned modelling and time-aware optimisation, in order to deal with NP-hardness. It has been recognised that CP is versatile for managing variability in software product lines [Salinesi et al. 2009]. In contrast to the SAT-solving, which implies the reformulation of relations in terms of boolean formulas, CP enables global constraint design, constraint optimisation and more generally, a higher level of abstraction for solving specific problems. Several existing approaches use a constraint model to reason on feature models [Benavides et al. 2005], [Karatas et al. 2010].

Labelling strategies. Constraint propagation prunes a variable domain but does not necessarily give a solution. To obtain the solution, we need to evaluate some assumptions regarding the values of variables. This process is called *labelling* and consists of adding a constraint $X = v$ to the constraint system, where X is an unbound variable, and v is a possible value from its domain. The order in which variables and values are selected for labelling is configurable. In general, constraint solvers propose several general-purpose labelling strategies, based on two parameters: (i) the selection order of variables, which can be selected statically or dynamically during the constraint solving process, and (ii) the exploration order of the domain: from the lowest to the highest value, or the opposite, from a randomly chosen value, or from a split of the domain, etc.

Time-aware optimisation in CP. Finding an optimal value for a cost function in CP can be achieved by implementing a *branch and bound* procedure. First, during the search process, when the solution is found, the value of the cost function being minimised can be recorded. Later, using a backtracking mechanism, other solutions can be enumerated to find better values of the cost function. The process can be controlled by a timeout value, allowing a search interruption when a defined time threshold has been reached. In other words, CP and labelling enable anytime optimisation problem solving, also known as time-constrained or time-aware optimisation. These capabilities are essential for minimizing the number of test configurations within the given time bounds, while enforcing pairwise coverage.

4. A CONSTRAINT OPTIMISATION MODEL FOR TEST CONFIGURATIONS GENERATION

This section provides a detailed description of our constraint optimisation model for automatic generation of valid test configurations with pairwise interaction coverage, thus, addressing the MVPF problem. First, we introduce the notations and relations modelled in this constraint optimisation model. Second, we propose a special global constraint enforcing the coverage of pairs in a set of configurations. Third, we ex-

[5]Constraint propagation runs in $O(N * M * D)$ where N is the number of variables, M is the number of constraints, and D is the size of the largest domain. It is thus a process of polynomial complexity.

VCS	CALL	P2P	MULTISITE	3 X 1024 X 756 MAX	3 X 720P 30 MAX	1080P30 - 720P60 MAX	PREMIUM RESOLUTION	720P60	1080P30
A_1	B_1	C_1	D_1	E_1	F_1	G_1	H_1	I_1	J_1
A_2	B_2	C_2	D_2	E_2	F_2	G_2	H_2	I_2	J_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
A_N	B_N	C_N	D_N	E_N	F_N	G_N	H_N	I_N	J_N

Fig. 3. An example matrix containing up to N configurations, created for the FM presented in Figure 2.

plain how to filter invalid pairs during the constraint optimisation process. Fourth, we present a heuristic to pre-compute the ordering of features to be labelled. Finally, we explain our time-aware optimisation process to minimise the selection of test configurations.

4.1. Notations and relations in an FM

Our *constraint optimisation model* is composed of a structured set of boolean variables, marked with capitals (e.g., $A, B, ...$), a set of finite-domain variables representing indexes, marked with $I_1, I_2, ...$, and a set of constraints or relations. Variables take their values in domains, denoted as I in $m..n$, where m, n are integers and where $m..n$ represent the integers in the range (i.e., all i such that $m \leq i \leq n$).

4.1.1. Variables and matrix data structure. In our model, boolean variables denote the selection/unselection of a feature in a configuration. Formally speaking, if variable A is associated with a feature, then $A = 1$ means that A is selected, while $A = 0$ means the opposite. We define a special matrix data structure to capture the status of configurations, composed of variables representing the features. The matrix is incrementally filled with values so that each of its rows represents a test configuration. The example matrix presented in Figure 3 is filled with boolean variables ($A_1, B_1, ...$) that are associated with the corresponding feature. Index variables $I_1, I_2, ...$ correspond to indexes of this matrix data structure. Their domain is $1..N$, if N denotes an upper bound on the number of configurations. The goal of our constraint optimisation model is to incrementally instantiate variables to values in this matrix, such that rows fulfil pairwise coverage criterion and the number of test configurations is minimised.

4.1.2. Relations. All configurations of the matrix have to satisfy the set of relations coming from the feature model, in order to represent valid configurations. To fulfil this requirement, we implemented dedicated constraints for capturing the relations in a feature model. These constraints filter domain information over the boolean variables, while constraint propagation is used to remove inconsistencies throughout the matrix data structure. We distinguish three types of relations: *hierarchical constraints*, such as and, or, xor, mand, opt, and card, *cross-tree constraints*, such as require and mutex, and *CNF constraints*, such as CNF. CNF constraints extend previous work [Hervieu et al. 2011] by enabling N-ary constraints, i.e., logical combinations of constraints over an

arbitrary number of features. This characteristic contributes to the better expressiveness of our approach and does not impact its performance.

Hierarchical constraints. Constraints such as and, or, and xor express relations between a parent feature A and its sub-features $B, C, ...$ For example, xor$(A, [B, C, ...]$ is true iff[6] $(B \Rightarrow A) \wedge (C \Rightarrow A) \wedge ... \wedge (A \Leftrightarrow B\ xor\ C\ xor\ ...)$. An alternative approach to model such constraints could have been the usage of ternary boolean operators that are available in most constraint solvers. However, we wanted to capture a global hierarchical relation between a parent feature and its sub-features, in order to propagate as far as possible the information in the constraint solving process. For example, considering an xor relation between 10 features permits us to instantiate all unbounded feature variables to false at once, after detecting that one sub-feature is true. Correspondingly, relation mand$(A, [B, C, ...])$ is true iff $A, B, C, ...$ are all true, opt$(A, [B, C, ...])$ is true iff either at least one of $B, C, ...$ is true and A is true, or all $B, C, ...$ are false, whatever be the value for A. The Cardinality relation takes two integer variables, N and M, for constraining the minimum and maximum number of sub-features with a value true. Note that Cardinality constraint is available in most CP solvers [Hentenryck and Deville 1991].

Cross-tree constraints. These relations include requires and mutex, where requires(A, B) is true iff $A \Rightarrow B$, and mutex(A, B) is true iff $A\ xor\ B$. The difference between cross-tree and hierarchical constraints lies in the absence of parental relationship between the features. Cross-tree constraints relate the parts of a feature model that are distinct.

CNF constraints. Relation $CNF([A_1, A_2, \ldots], [B_1, B_2, \ldots])$ captures a disjunctive clause in a CNF expression: it is true iff $\neg A_1 \vee \neg A_2 \vee \ldots \vee B_1 \vee B_2 \vee \ldots$ is true. Note that CNF constraints generalise cross-tree constraints and can be used to replace them. In our framework, cross-tree constraints are kept for the sake of generality and expressivity. Note also that in an FM, whenever neither a cross-tree constraint nor a CNF constraint is present, the FM necessarily contains at least one valid configuration [Mendonca et al. 2009].

4.2. Enforcing pairwise coverage

In our approach, enforcing pairwise coverage requires that each pair of values is present at least once in the matrix data structure. For this purpose, we introduce a new global constraint called pairwise. The pairwise relation holds over an index variable I, representing a row in the matrix, and two vectors of feature variables (to address 2-way feature coverage). The pairwise constraint forces a specific pair of boolean values to be included at some location in the vectors, depending on the variable I. For example, pairwise$(I, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (1, 1))$ constrains the unknown row I of the matrix to contain the pair $(1, 1)$, for the features associated with rows $[X_1, X_2, X_3]$ and $[Y_1, Y_2, Y_3]$, meaning that the corresponding pair must be included within the test configuration of a rank I. During the solving process, if I is instantiated to value 2 then $(X_2, Y_2) = (1, 1)$, while if X_3 is instantiated to value 0, then value 3 will be removed from the domain of I, because the pair of variables (X_3, Y_3) cannot be equal to $(1, 1)$ in this case. The filtering algorithm shown in Alg. 1 represents the implementation of the global constraint pairwise$(I, (L_1, L_2), (v_1, v_2))$, where I, $L_1[i], L_2[i]$ represent the sets of integers (a.k.a., finite domains), and (v_1, v_2) represents boolean constants. This algorithm is launched each time the domain of any of the variables $I, L_1[1], L_1[2], ..$ is

[6] In Constraint Programming, adding redundant constraints is a convenient way to speed-up the resolution process.

ALGORITHM 1: A filtering algorithm for the pairwise constraint

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 boolean values
Output: $Fail$ or pruned domains for (I, L_1, L_2)

```
function pairwise(I, (L1, L2), (v1, v2))
I' ← I;
foreach i ∈ I do
    L'1[i] ← ∅, L'2[i] ← ∅;
    if (v1 ∉ L1[i]) or (v2 ∉ L2[i]) then
        I' = I'\{i}
    else
        L'1[i] ← L1[i];
        L'2[i] ← L2[i];
    end
end
if I' = {a} then
    L'1[a] ← {v1}; L'2[a] ← {v2};
return ({a}, L'1, L'2);
else if (I' = ∅) then
    return Fail
else
    return (I', L'1, L'2)
end
```

reduced (a.k.a., awakening conditions). The idea behind this algorithm is to explore all possible values for I and to determine whether these values are still consistent with the domain information for other variables. The complexity of this algorithm is linear in the size of the domain for I, as it iterates on its values. Other operations implemented in the algorithm are performed on finite domains and can be bounded by a constant, equal to the domain's largest size. Note that our approach has similarities with the usage of *global-cardinality constraints* proposed by Hnich [Hnich et al. 2006]; the difference being that the pairwise global constraint makes no use of costly network maximum flow computations during domain filtering. As mentioned above, the complexity of pairwise algorithm is linear, while the complexity of global-cardinality filtering algorithm is at least quadratic in the number of nodes of a flow network.

In general, several pairwise constraints are posted in the constraint solving process and the ordering in which these constraints are posted impacts the resolution time, but not the final result. In our framework, we have introduced the *pairs-ordering heuristic* to manage the order in which pairwise constraints are posted. This heuristic is described in detail in Section 4.4.

4.3. Filtering invalid pairs

The pairwise global constraint presented above is used to enforce the presence of all pairs within the matrix data structure. However, in the presence of constraints between features in an FM, some pairs become invalid. Thus, covering all pairwise interactions requires filtering invalid pairs in the selection process. In this section, we present how to detect and remove invalid pairs.

A pair of features from an FM is invalid if there are no valid configurations containing this pair. For example, two features linked with the mutex operator are necessarily invalid, as they cannot be selected at the same time in a valid configuration. In order to remove invalid pairs, we introduce the following two-step procedure. Firstly,

constraint propagation is used to remove some invalid pairs through inconsistency detection. Secondly, a selective labelling procedure allows us to eliminate residual invalid pairs. More specifically, constraint propagation is an efficient yet an incomplete process for detecting inconsistencies (i.e., invalid configurations), therefore, it may not be able to eliminate all invalid pairs. Still, all inconsistencies can be detected by complementing constraint propagation with a labelling procedure (i.e., giving a concrete value to the selected variables). The labelling procedure is built on a failure-driven backtracking. This means that if an added hypothesis (e.g., choosing a value for a variable) leads to a contradiction, then the labelling procedure automatically reverts the choice and makes a different hypothesis (e.g., choosing another value for a variable).

$$\text{REQUEST: } \mathsf{or}(A, [B, C]), B = 1, \ \mathrm{solve}().$$

$$\text{RESULT: } A = 1, C \text{ in } 0..1.$$

Fig. 4. A constraint propagation example

Consider the simple example shown in Figure 4. The first line is a typical request to the constraint solver, where $\mathsf{or}(A, [B, C])$ denotes a hierarchical constraint between feature A and sub-features B, C, and $B = 1$ entails the selection of feature B in the configuration. The second line is a result returned by the constraint solver. Constraint propagation, as launched by a call to the constraint solver (i.e., solve), leads to $A = 1, C$ in $0..1$, meaning that feature A must be selected in the configuration, while feature C is left unspecified. Implicitly, constraint propagation removed the invalid pair $(A = 0, B = 1)$, which was never considered in the process. We say that constraint propagation pruned the search space of some invalid pairs. However, as mentioned above, constraint propagation may not discard all invalid pairs. Figure 5 shows an interesting example, including three operators and two CNF constraints, that illustrates this principle.

$$\text{REQUEST: } \mathsf{and}(R, [A, B]), \mathsf{xor}(B, [B_1, B_2, B_3]), \mathsf{xor}(A, [A_1, A_2]), CNF([B_1], [A_1]),$$
$$CNF([B_2, B_3], [A_2]), R = 1, A_1 = 0, \ \mathrm{solve}().$$

$$\text{RESULT: } A = 1, B = 1, A_2 = 1, B_1 \text{ in } 0..1, B_2 \text{ in } 0..1, B_3 \text{ in } 0..1.$$

Fig. 5. An example where constraint propagation does not discard all invalid pairs

In this example, the feature R is selected ($R = 1$) and the feature A_1 not selected ($A_1 = 0$). Consequently, constraint propagation instantiates the features A, B and A_2, leaving the features B_1, B_2 and B_3 uninstantiated. From the result (the second line), it can also be deduced that the following pairs are valid: $(A_1 = 0, B_1 = 0)$, $(A_1 = 0, B = 1)$, $(A_1 = 0, B_2 = 0)$, $(A_1 = 0, B_2 = 1)$, $(A_1 = 0, B_3 = 0)$, $(A_1 = 0, B_3 = 1)$. However, the result does not exclude the pair $(A_1 = 0, B_1 = 1)$, which is invalid. In fact, if $B_1 = 1$, then the relation $\mathsf{xor}(B, [B_1, B_2, B_3])$ leads to $B_2 = 0$ and $B_3 = 0$. As $A_2 = 1$, the CNF constraint $CNF([A_2], [B_2, B_3])$ is violated (i.e., instantiated to false). This result illustrates that the constraint propagation, although efficient, must be completed by a labelling procedure that can backtrack on the choice $B_1 = 1$.

This labelling procedure aims at checking the validity of every pair containing at least one uninstantiated variable. The idea simply consists in reusing constraint propagation while making additional hypothesis, and backtracking when an inconsistency is detected. For the example in Figure 5, to detect an inconsistency, it suffices to call solve(), when $B_1 = 1$ is added to the constraint system (Result: $false$). On the contrary,

Table I. Pairs-ordering metrics for two pairs of features

Pairs	♯1 : 3X1024X576 max; 1080p30	♯2 : 720p60; 1080p30
depth_father	0	1
depth_features	7	4
common_operator	$mand$	xor

adding $B_1 = 0$ to the system does not lead to any inconsistency, meaning that the pair $(A_1 = 0, B_1 = 0)$ is actually valid.

4.4. Search heuristics

Labelling procedures come with *search heuristics*, i.e., techniques to explore a search space in an optimised way. In this section, we investigate the definition of a search heuristic, taking advantage of the structure of an FM to improve the labelling process. In our framework, enforcing pairwise coverage involves considering two types of constraints: (i) Pairwise constraints, such as $\mathsf{pairwise}(I, (L_1, L_2), (v_1, v_2)$, and (ii) Relations extracted from an FM. Both types of constraints are considered together during the constraint propagation and labelling processes. Still, some options, such as introducing the constraints in the constraint solver or selecting variables to enumerate first are left to the user, which can improve the performance of the constraint model.

Pairs-ordering heuristics. In our framework, we observed that introducing Pairwise constraints in different order impacts the performance of the solving process. To examine this phenomenon more thoroughly, for each pair of features from an FM we defined three metrics, all making use of the notion of a *depth* of a feature. For a feature F, $depth(F)$ denotes the number of its ancestors to the root of an FM structure.

(1) $depth\_features$: for a given pair of features (F_1, F_2), $depth\_features$ represents $depth(F_1) + depth(F_2)$. This metric intends to capture the following intuition: as soon as a feature is selected, its parent feature is automatically selected through constraint propagation and thus, labelling the children features first may accelerate the solving process;
(2) $depth\_father$: for a given pair (F_1, F_2), $depth\_father$ represents the depth of the closest common ancestor of F_1 and F_2 in an FM structure. This metric captures the value representing the distance between two features;
(3) $common\_operator$: for a given pair (F_1, F_2), $common\_operator$ represents the type of the closest common operator of F_1 and F_2. We order distinct relations that can be found in an FM as follows:

$$mand > and > xor > card > or > opt$$

Regarding the metric $common\_operator$ and the proposed ordering, when selecting a pair to be labelled first, an and relation has higher priority than an opt relation. This is due to a higher deduction capability of and over opt. For example, in $\mathsf{and}(R, [A, B])$, if R is set to 1 during the resolution process, then, through constraint propagation, both A and B are set to 1 as well. This obviously does not happen when an opt operator is used instead of and. Hence, it is preferable to start labelling the and relations before the opt relations. The ordering proposed above reflects these priority levels.

Table I shows the values of these three metrics for two pairs of features extracted from the FM shown in Figure 2. The Pairs-ordering search heuristics prioritise the selection of pairs using these metrics. The goal of these heuristics is to maximise the deduction capabilities of the reasoning engine. The metrics can be used individually or in combination, leading to different Pairs-ordering heuristics. Of course, these metrics are only indicators and using them while selecting pairs does not guarantee achieving

the most efficient pair selection process in all cases. However, our experiments show that using the Pairs-ordering heuristic with these metrics is advantageous for the resolution process, as described in Section 6.3.

Variable-selection heuristics. In our framework, we propose two variable-selection labelling heuristics: *left-most* and *first-fail* (provided by the constraint solver).

(1) $left\_most$ heuristic takes the first element in the list of finite-domain variables, X, and assigns it the smallest value in its domain, v. Then, the constraint $X = v$ is added to the constraint storage and propagated through the constraint network. As a result, other variables may become instantiated. After that, the process selects the next unbounded element in the list and further iterates until all variables have been labelled. This strategy implements a static selection of variables to enumerate first, as the list is set before the constraint solving process has been started;
(2) $first\_fail$ heuristic selects the element with the smallest domain and assigns it the smallest value in its domain. This strategy is dynamic as the domain of an unbounded variable can be reduced by the propagation of the constraint $X = v$.

Both heuristics use the same domain covering order, from the lowest to the highest value. In our framework, the ranks of a generated pair (i.e., index variables) are unknown and correspond to the variables of the constraint system. In this context, trying to instantiate index variables with the lowest values is preferable, as this will ensure that the minimum number of configurations is reached faster. This helps to minimise the number of test configurations required to cover pairwise. In Section 6.3, we evaluate the heuristics in order to determine the optimal performance of the constraint model.

4.5. A time-constrained minimisation problem

The constraint model described above is used to find the minimum number of test configurations fulfilling pairwise coverage criterion. First, we explain how to formulate the problem as an optimisation problem. Afterwards, we show how to solve this problem using a time-constrained algorithm with various search strategies.

Solving the MVPF problem can be seen as the following question:

$$\textbf{Find } I_1, ..., I_{4n^2} \textbf{ such that } f \textbf{ is minimised and } \forall i, j \textbf{ in } 1..n, \forall k \textbf{ in } 1..4n^2 \textbf{ such that } (k \mod 4) = 1$$
$$\textsc{Pairwise}\ (I_k, C_i, C_j, (0,0)), \textsc{Pairwise}\ (I_{k+1}, C_i, C_j, (0,1)),$$
$$\textsc{Pairwise}\ (I_{k+2}, C_i, C_j, (1,0)), \textsc{Pairwise}\ (I_{k+3}, C_i, C_j, (1,1)).$$

where n denotes the number of features, while C_i, C_j represent the columns of the matrix data structure. In our model, a new pairwise constraint with a new index variable is introduced for each pair of feature values that must be present in the minimized set of configurations. Since many pairs can be found in a single configuration, the overall objective is to minimize the number of distinct values used for indexes. In our proposed approach, we explore the usage of two distinct cost functions, $f_1 = \sum_{k \in 1..4n^2} I_k$ and $f_2 = \mathsf{Max}_{k \in 1..4n^2} I_k$. Both functions can be used to find the minimum number of values for the I_k, such that the pairwise coverage criterion is satisfied in the matrix. Formulating the problem as a generic optimisation problem gives the advantage of using several optimisation techniques, while keeping a single problem formulation. We base our approach on the *branch and bound* method, which consists in exploring feasible solutions while maintaining the cost function as low as possible. Generally, at each search tree node, *branch and bound* method evaluates the cost function, prunes the sub-trees for which the cost will be clearly higher than the current value, and

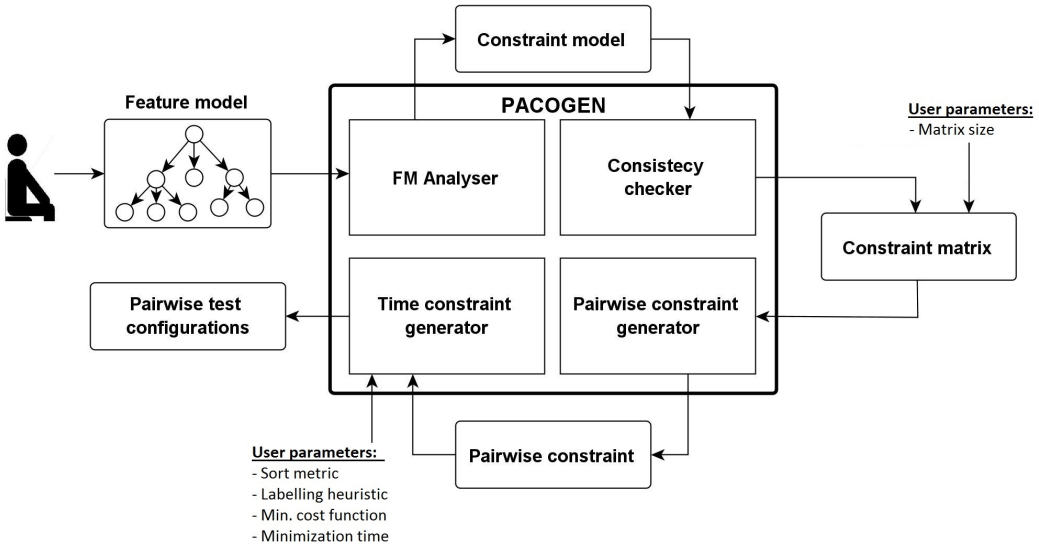

Fig. 6. PACOGEN architectural overview

selects the sub-tree that has the lowest cost. There is a timeout value set for the algorithm, after which the minimisation process is automatically stopped. This is a property common to anytime algorithms [Zilberstein 1996]. The timeout value provides a time-controlled way to solve the minimisation problem. This idea is based on the fact that *branch and bound* method provides near-optimal solution very quickly and most of the remaining time is used to prove that no better solution exists. It may, however, happen that a better solution will be found if more time is allocated, but the proposed solution is a good compromise in cases where there is a need to control the test generation/execution time.

5. TOOL IMPLEMENTATION

We implemented our approach for optimal minimisation of valid test configurations with pairwise-interaction coverage presented in Section 4 in a software prototype tool called PACOGEN. The tool is mainly developed in Prolog and Java with approximately 7K LOC. We used SICStus Prolog `clpfd` library to resolve the constraint optimisation problem, since it is known as one of the best finite-domain constraint solving libraries. PACOGEN is available as a free open-source tool [7].

In Figure 6 we present a general architecture of PACOGEN and a flow that relates its core components. PACOGEN comprises four main components:

(1) **FM Analyzer** transforms a *feature model (FM)* into a *constraint model (CM)*, as an abstract syntax tree. This component transforms all cross-tree relations into a set of constraints specified in the SICStus Prolog concrete syntax. The CM generated from the FM shown in Figure 2 is presented in Figure 7. `FList` parameter corresponds to the list of features, while `CList` represents the relations between features. `solver` method represents the call of constraint solver that triggers the resolution process. The method has `5` parameters. `FList` is the list of features, `CList` is the list of relations between the features, `Size` is the matrix size. The `Size` param-

[7] http://hervieu.info/pacogen.

eter represents an over-estimated value of the global minimum solution, sought by our approach. In Section 6.3, we discuss how to initially set this input parameter value. The `TimeOut` parameter allows users to specify the time available for the minimisation process. In practice, this parameter is selected depending on the time available for the overall testing process and the time required to test each configuration. For example, it is pointless to allocate hours for minimizing the set of test configurations if the time required to test a single configuration is less than a few seconds. On the contrary, if the time needed to test a configuration is more than an hour, then spending some more time to further minimise the set of test configuration is beneficial. Finally, `Minimisation` parameter allows users to specify the minimisation function, e.g., f_1 or f_2. If the parameters are not specified by the user, default values are used in the tool.

FList = [VCS, CALL, P2P, MULTISITE, 3X1024X576 MAX, 3X720P30 MAX, 1080P30–720P60 MAX, PREMIUM RESOLUTION, 720P60, 1080P30],
CList = [mand (VCS, [CALL]), opt (VCS, [PREMIUM RESOLUTION]),
xor (CALL, [P2P, MULTISITE],
xor (MULTISITE, [3X1024X576 MAX, 3X720P30 MAX, 1080P30–720P60 MAX]),
or (PREMIUM RESOLUTION, [720P60,1080P30])
require (3X720P30 MAX, PREMIUM RESOLUTION),
require (1080P30–720P60 MAX, PREMIUM RESOLUTION)]

solver (FList, CList, Size, TimeOut, Minimisation).

Note: Logical variables are in upper-case; Constants and predicate calls are in lower-case.

Fig. 7. Constraint model generated for the video conferencing system FM

(2) **Consistency checker** evaluates the consistency of the constraint model and produces a *constraint matrix* data structure. The matrix is of $K \times N$ size, where the number of columns K is the number of features, and the number of rows N is a user-specified value for the number of configurations. For example, the constrained matrix generated for the VC software FM is shown in Figure 8. Since each row represents a valid configuration, PACOGEN automatically generates the constraints over the variables from a row, forcing the corresponding configuration to satisfy the constraint model.

$$\begin{pmatrix} VCS_1 & CALL_1 & P2P_1 & MULTISITE_1 & \dots & \_720P60_1 & \_1080P30_1 \\ VCS_2 & CALL_2 & P2P_2 & MULTISITE_2 & \dots & \_720P60_2 & \_1080P30_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ VCS_n & CALL_n & P2P_n & MULTISITE_n & \dots & \_720P60_n & \_1080P30_n \end{pmatrix}$$

$[mand(VCS_1, [CALL_1]), opt(VCS_1, [PREMIUM\ RESOLUTION_1]),$
$xor(CALL_1, [P2P_1, MULTISITE_1],$
$xor(MULTISITE_1, [3X1024X576MAX_1, 3X720P30MAX_1, 1080P30–720P60MAX_1]),$
$or(PREMIUM\ RESOLUTION_1, [720P60_1, 1080P30_1])$
$require(3X720P30MAX_1, PREMIUM\ RESOLUTION_1),$
$require(1080P30–720P60MAX_1, PREMIUM\ RESOLUTION_1)]$
$[mand(VCS_2, [CALL_2]), opt(VCS_2, [PREMIUM\ RESOLUTION_2]),$
$xor(CALL_2, [P2P_2, MULTISITE_2],$
$xor(MULTISITE_2, [3X1024X576MAX_2, 3X720P30MAX_2, 1080P30–720P60MAX_2]),$
$or(PREMIUM\ RESOLUTION_2, [720P60_2, 1080P30_2])$
$require(3X720P30MAX_2, PREMIUM\ RESOLUTION_2),$
$require(1080P30–720P60MAX_2, PREMIUM\ RESOLUTION_2)]$ ll

Fig. 8. Resolution matrix and the constraints for the first two rows of the matrix, for the VC system FM

(3) **Pairwise constraint generator** takes as input the constraint matrix and adds a set of global constraints enforcing pairwise coverage. For a given pair of features, four possible pairs of values are $(0,0),(0,1),(1,0),(1,1)$. Before adding a pairwise constraint enforcing the presence of a pair of values in a set of test configurations, each pair is evaluated for validity. However, to keep the process as efficient as possible, we have implemented only a partial validity test as follows: after a pair is selected and labelled, constraint propagation automatically resumes and drives the process to an early detection of inconsistencies. Using constraint propagation, PACOGEN can eliminate invalid pairs when the corresponding constraint system is shown to be unsatisfiable. As explained in Section 4.3, not all invalid pairs can be removed in this way, as constraint propagation is incomplete in essence. Still, our experience shows that most of the invalid pairs are removed. This process is applied to all pairs, so that at the end, in the worst case, only few invalid pairs are among the pairs to be enforced. For the VC example provided by our industrial partner, 97 pairwise constraints were generated in total, each of them corresponding to a valid pair. Figure 9 shows three examples of pairwise constraints. In the first example, the pair $(1,1)$ is enforced at the rank I_1 for the pair of features $(CVCS, CCALL)$ of the matrix shown in Figure 8. Note that I_1 is unknown at the time of pairwise constraint generation and will be instantiated later by the constraint resolution process.

$$\text{PAIRWISE}(I_1, ((CVCS, CCALL), (1,1)),$$

$$\text{PAIRWISE}(I_2, ((CVCS, CPREMIMUM\_RESOLUTION), (1,1)),$$

$$\text{PAIRWISE}(I_3, ((CVCS, CCPREMIMUM\_RESOLUTION), (1,0)), ...$$

Fig. 9. Created pairwise constraints

(4) **Time-constrained pairwise-covering test configuration generator** launches the constraint optimisation process aiming to find the minimum set of test configurations that satisfy pairwise feature coverage. The process fills the constraint matrix by assigning possible values for the pairs of features and for the ranks in pairwise constraints. At the end, the matrix contains only the valid test configurations that satisfy pairwise feature coverage. The constraint optimisation model finds the minimum of the cost function in a given contract of time. The set of test configurations is provided in CSV format.

6. EXPERIMENTAL EVALUATION

In this section, we present the experimental study performed to evaluate PACOGEN. We first compare PACOGEN with the two related approaches for generating test configurations with pairwise-interaction coverage. Then, we evaluate PACOGEN on a real case study of large highly-configurable video-conferencing software, provided by our industrial partner. Finally, we evaluate different PACOGEN search heuristics, in order to propose a tool configuration that maximises its efficiency. Note that we present the summary of the experiments, which are available online[8]. The experiments were performed on a standard 64-bit Linux machine, with 2 CPU Intel Xeon $E5520$ (quad core), with 16GB memory.

Research questions. The research questions addressed in this evaluation are:

— **RQ1:** Can PACOGEN generate a smaller set of test configurations compared with greedy approaches?

[8]http://hervieu.info/pacogen.

— **RQ2:** Does PACOGEN prove the potential for automating the generation of test configurations in an industrial context?
— **RQ3:** Does the capability of configuring PACOGEN for different minimisation problems improve the effectiveness of the approach?

6.1. Minimizing the number of configurations

To answer RQ1, we designed an experimental study where PACOGEN was compared with two competitive approaches: SPLCAT [Johansen et al. 2011; 2012b] and MosoPolite [Oster et al. 2010]. The comparison was made in terms of the number of generated test configurations and generation time.

6.1.1. Experiment subjects. The comparison between PACOGEN and SPLCAT was performed on the FMs from SPLOT repository [Mendonça et al. 2009]. At the time we ran the experiments, there were 224 FMs available, ranging from 9 to 290 features. The comparison between PACOGEN and MosoPolite was performed on 7 FMs used by the author in his evaluations [Oster et al. 2010].

6.1.2. Experiment setup. First we ran SPLCAT and then PACOGEN without the minimisation process on 224 FMs from SPLOT. Next, we ran PACOGEN in two iterations: (i) with the minimisation and time-contract set to $4 * ResolutionTime$, where $ResolutionTime$ corresponds to the time taken to find the results without minimisation (in Section 6.3, we explain how to set the time-contract for the minimisation), and (ii) with a three-minute minimisation. All experiments were run on the same machine. Finally, MosoPolite tool was not available to reproduce the experiments. Hence, we ran PACOGEN on 7 FMs that the author used to evaluate his tool and we compared the results with the experimental results reported in Oster's publication [Oster et al. 2010].

6.1.3. Results and analysis. Table II shows an excerpt of the experimental results: 17 FMs have been selected to illustrate the most interesting results, while the complete experimental results are available online. The first three columns of the table show FM properties: the number of features, the number of valid configurations, and the number of 2-way combinations (pairs). The next two columns present the number of valid test configurations, found by SPLCAT and MosoPolite respectively. The next column presents the number of valid test configurations generated by PACOGEN. The last two columns of Table II show the difference in the number of test configurations found by PACOGEN versus SPLCAT and MosoPolite respectively, in percentage. Negative values indicate that PACOGEN found less configurations than the other tools. Table III shows the resolution time for PACOGEN and SPLCAT for the same 17 FMs.

The comparison of PACOGEN with SPLCAT on 224 FMs showed that PACOGEN generated fewer test configurations for 79% of 224 FMs. For only 2% of FMs, SPLCAT provided better results and for 18% of FMs both tools obtained the same number of configurations. The results further show that PACOGEN found up to 60% fewer configurations than the SPLCAT. Figure 10 illustrates the difference in the number of test configurations found by these two tools across 224 FMs. X-axis represents the difference in the number of test configurations in percentage, grouped in 9 groups. Negative values on an x-axis mean that PACOGEN generated fewer configurations, while positive values mean the opposite. Y-axis represents the number of FMs. The group of FMs labelled with 0% in the figure represents the models where both tools gave the same results.

Table II. Comparison of SPLCAT, MosoPolite and PACOGEN in term of the minimum number of valid configurations with pairwise interaction coverage. Parameters: first.fail, sort

Model	Metrics			SPLCAT	MosoPolite	PACOGEN	Change in Array Size	
	♯F	♯Conf	♯Pairs				vsSPLCAT	vsMosoPolite
Car PL	9	13	102	7	–	6	−14%	–
Aircraft PL	13	315	240	9	–	8	−11%	–
Movie APP PL	13	24	211	7	–	6	−14%	–
Search Engine	14	126	275	13	–	11	−15%	–
Stack PL	17	432	465	12	–	9	−25%	–
Connector PL	20	18	537	15	–	14	−7%	–
Fame DBMS	21	320	616	9	–	8	−11%	–
Smart Home	35	1048576	1465	10	11	9	−10%	−18%
Inventory	37	2028096	1974	13	12	10	−23%	−17%
Sienna	38	2520	1911	22	24	20	−9%	−17%
Doc generation	44	55700000	3093	12	18	12	0%	−33%
Web Portal	43	2120800	3196	19	26	15	−21%	−42%
Arcade game	61	$3.3 * 10^9$	5209	18	–	12	−33%	–
Model Trans.	88	$1.63 * 10^{13}$	13139	25	40	24	−4%	−40%
Coche Ecologico	94	23200000	11075	93	92	90	−3%	−2%
UP estructural	97	$> 10^9$	15958	36	–	33	−8%	–
Printers	172	$> 10^9$	42638	182	–	180	−1%	–

Metrics: number of features (♯F), number of valid configurations (♯Conf), number of valid 2-way combinations (♯Pairs).

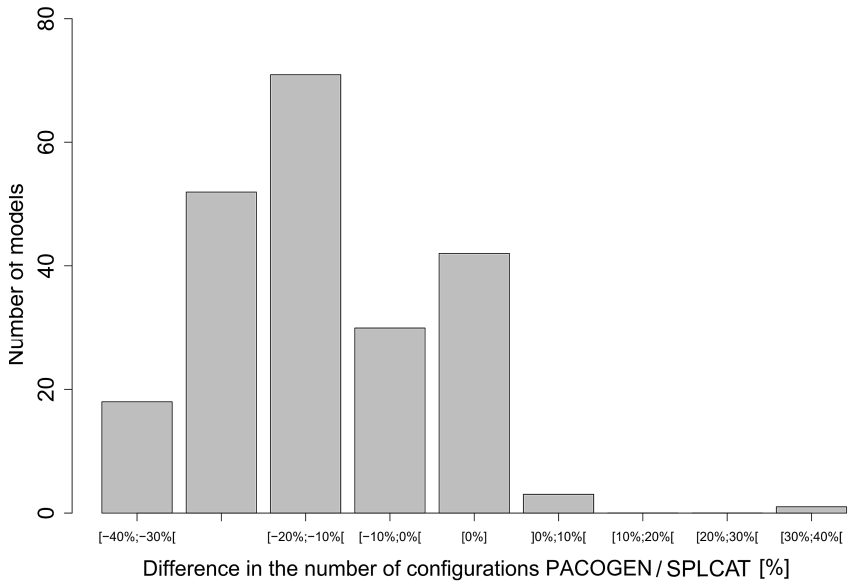

Fig. 10. Difference in the number of configurations found by PACOGEN and SPLCAT (in percentage) for 224 FM. Settings: labelling = first_fail; sort; minimisation function 1; matrix size = 200.

In the comparison of PACOGEN with MosoPolite on 7 FMs, PACOGEN obtained up to 42% fewer test configurations for 6 FMs and the same number of test configurations as MosoPolite for 1 FM. The results are presented in Table II.

The comparison of PACOGEN and SPLCAT in terms of resolution time shows that PACOGEN is slower than the other two tools. The average resolution time for PACOGEN is 132 074 ms, and the median time is 9 855 ms. The average resolution time for SPLCAT is 611 ms, while the median time is 326 ms. However, since PACOGEN is the approach that aims at finding a true optima and not approximations, it is not appropriate to compare generation times. The trade-off between sample size and sample generation time depends on the context, how expensive configurations are to test. In this paper, we show that there are contexts where favouring sample size over sample generation time is practically relevant.

In summary, the results of this experiment show that PACOGEN generates fewer test configurations than the greedy approaches SPLCAT and MosoPolite.

6.2. Applicability to industrial context

To answer the research question RQ2, we performed the experimental study where PACOGEN was applied to a large industrial highly-configurable networking software described in Section 2. We evaluated the practical relevance of PACOGEN in the testing process of our partner by two means: experimentally and through the set of interviews. Relating to the later, we also discuss the challenges raised by the use of PACOGEN in industrial context and we provide practical ways to address those challenges.

6.2.1. Experiment setup. We applied PACOGEN to Cisco highly-configurable networking software, making the change to their traditional testing process. Instead of specifying test configurations manually, validation engineers built a feature model and used

Table III. Comparison of the resolution time in ms for PACOGEN and SPLCAT.

Model	Time [ms]	
	PACOGEN	SPLCAT
Car PL	1240	340
Aircraft	3570	971
Movie App	12040	520
Search Engine	214340	596
Stack PL	198520	273
Connector	1019950	803
Fame	889930	574
Smart Home	198520	974
Inventory	295736	400
Sienna	134297	1 693
Doc generation	120287	734
Web Portal	176080	2 270
Arcrade Game	109009	2 559
Model Transformation	98457	3 165
Coche Ecologico	100336	4 724
UP Estructural	89447	3 320
Printers	78946	15 306

PACOGEN to automate the generation of test configurations. They built the model iteratively, through collaborative review iterations and continuous discussions between the members of the test team, which almost completely assures the correctness of the model. The feature model of the analysed VC software consisted of 169 features, which corresponds to more than 10^9 configurations (valid and invalid). First, we measured the time to generate test configurations and compared it to the time taken by the engineers to generate the same number of configurations manually. Second, we compared the number of test configurations generated by PACOGEN for the built FM with the number of test configurations that the engineers use to test the software modelled by the FM. Third, we measured a two-way feature coverage for the set of test configurations used by the engineers to test the software modelled by the FM and compared it with a full two-way feature coverage ensured by PACOGEN. The first two measurements evaluate a practical benefit of PACOGEN's capability to seek an exact minimum and to halt with intermediate results, while also automating the process and thus decreasing the manual effort of test engineers. The third measurement evaluates the benefit of pairwise configuration sampling.

In addition, we conducted the set of interviews to evaluate the interest of engineers in applying PACOGEN in an industrial setting. The subjects consisted of 2 validation engineers and 3 managers, which had from 2 to 7 years of experience in testing a video conferencing software. The subjects were selected to involve individuals with different roles in the company and different level of experience. The interviews were conducted with the subjects, both separately and in a group. The interviews were semi-structured and were following a pyramid model, beginning with specific questions and opening more general questions during the course of the conversation. All subjects were involved in the experiment of applying PACOGEN in Cisco, to automate the process of generating test configurations. In the interviews, the subjects were asked about their opinion on the potential of PACOGEN for application in practice: decreasing the time spent generating test configurations, increasing the quality (coverage, validity) of test data, improving the quality of testing processes by providing a systematic method for testing feature interactions, helping managers to plan and schedule delivery deadlines easier. The purpose of the interviews was to understand the engineers' perceived

usefulness of PACOGEN in practice, which is one of the most important factors determining the success of adopting new technology in an industrial context.

6.2.2. Results and analysis.

Decreased generation time of test configurations. Given the VC software FM, PACOGEN generated a minimum set of 35 valid test configurations in 12.3 hours. Time taken for validation engineers to manually specify test configurations for the same VC software is approximated by the engineers themselves to 87 hours. These results show that PACOGEN can decrease the time required by manual test configuration generation by 85%. Time taken by feature modelling has not been accounted for in the calculation, since it is one-time effort in most cases, as an FM can be updated and reused in consecutive test case generation processes.

Reduced number of test configurations. For the VC software FM used in the experiments, PACOGEN obtained a minimum set of 35 valid test configurations (the first computed solution consisted of 37 test configurations). The set of test configurations specified manually and used by validation engineers contains 87 valid configurations. These results show that PACOGEN can reduce the number of test configurations used in our partner's testing practice by 60%. Decreased number of test configurations directly decreases overall testing time, which has significant impact on efficiency in domains where configurations are expensive to test. We use configuration count metric for measuring overall testing effort, since in our context configurations are roughly equally expensive. Otherwise, it would be necessary to measure actual testing time.

Complete 2-way feature coverage. In the VC software FM, there are 2625 possible valid feature pairs. We analysed the manually specified set of test configurations and extracted 505 feature pairs, which makes 19% of total pairwise feature coverage. These results indicate a low pairwise feature coverage for the manually generated test configurations, which means that there are many untested 2-way feature interactions. On the contrary, PACOGEN can guarantee complete 2-way feature coverage of an FM. However, we did not analyse the link between pairwise-coverage of a test suite and its effectiveness in detecting errors, since the objective of the test engineers, at that point, was to achieve pairwise coverage only.

In summary, the results of the experiment show that practical relevance of PACOGEN for an industrial testing process is attributed to three characteristics: (i) ability of finding an exact minimum number of test configurations, thus reducing the time of test execution, (ii) capability of pairwise configuration sampling, which increases the confidence of test managers in the validity of tested software, and (iii) test generation automation, making the testing process more systematic and less prone to human errors.

The interviews. The interviewees stated that PACOGEN helps them generate tests faster and improves their confidence in the quality of the tested products, as the generation process is formal, systematic and offers precise coverage criteria. The used FMs are constructed iteratively, through reviews and continuous discussions between the members of the test team, to ensure the correctness of models. Therefore, PACOGEN helps them assure that the used (generated) test configurations are valid, meaning that the failures can be only due to incorrect software implementation (and not an invalid configuration or a model). Knowing that a minimum-size set of test configuration (for a specified coverage level) is used in execution is valuable information, because it assures that resources are not misspent in test execution. The interviewees with managerial roles supported the idea of defining a time interval for generating test configurations. Very often, in practice, engineers (especially in agile teams) are given a

fixed time interval for testing. Having a way to specify the time for generating test configurations would help them meet product release deadlines easier. However, the interviewees also pointed to two particular issues they found difficult; the estimation of the: (i) minimisation time, and (ii) matrix size parameters needed to start PACOGEN. If the allocated minimisation time is too short, PACOGEN will not be able to find a solution. On the contrary, if the minimisation time is too long, it will lead to inefficiency, as PACOGEN will find the solution and then spend the rest of the time just proving that the found solution is the best one. Similarly, a matrix too small will prevent PACOGEN from finding the solution, while a matrix too large will increase the size of the constraint model and will degrade the performance of the algorithm.

In most cases, the product under test is an upgrade of a previously tested product and validation engineers can reuse the values for the minimisation time and matrix size from the previous runs. Later, when validation engineers become well experienced with the tool, they will be able to manually update the previously obtained parameter values with respect to the modifications in the feature model. Still, to respond to this objection more formally, we provided several practical ways of estimating these two parameters before launching PACOGEN (Section 6.3). These practical strategies are meant to increase the usability of the tool and help validation engineers increase their productivity.

6.3. PACOGEN best configuration

Launching PACOGEN involves specifying two parameters: the initial size of the constraint matrix, and time allocated to the constraint optimisation solver. Optimally selecting values for these parameters can increase the performance of the constraint optimisation process. We propose several ways to determine the best values for these parameters.

The simplest approach consists of, first, running PACOGEN without minimisation, providing us with the approximation of the solution size (matrix size) and the reference time. These values can then be used to set up the minimisation process. We followed this dedicated approach in all our experiments and it proved to be an efficient usage process for PACOGEN. Another approach is to use the existing pairwise testing tools to estimate the parameters. Also, the values for these parameters can be pre-computed based on the number of features for a given FM. For example, we propose a function to compute the time required to find a solution from the number of features of an FM. The function has been empirically calculated based on the set of 224 feature models, using polynomial regression of degree 3, as shown in Figure 11. The function can be used to determine a time contract for the minimisation process by extrapolation. Of course, it is only formally valid for the FM that we considered in the experiment, but it seems reasonable to extrapolate the values for other FMs with similar size.

Furthermore, PACOGEN implements several different internal heuristics in order to optimise the constraint solving process: $depth\_father$, $common\_operator$ and $depth\_features$ as the pairs-ordering heuristics, $first\_fail$ and $left\_most$ as the variable-selection labelling heuristics, and two cost functions: $f_1 = \mathsf{Max}_{k \in 1..4n^2} I_k$ and $f_2 = \sum_{k \in 1..4n^2} I_k$. The heuristics are listed in Table IV. We experimentally evaluated these heuristics on 224 FMs from SPLOT, in order to find the best PACOGEN configuration that maximises the tool efficiency. First, we analyse how the pairs-ordering heuristics affect the number of configurations found by PACOGEN. Second, we evaluate the performance of the variable-selection labelling heuristics. Last, we evaluate two cost functions used in the constrained minimisation process.

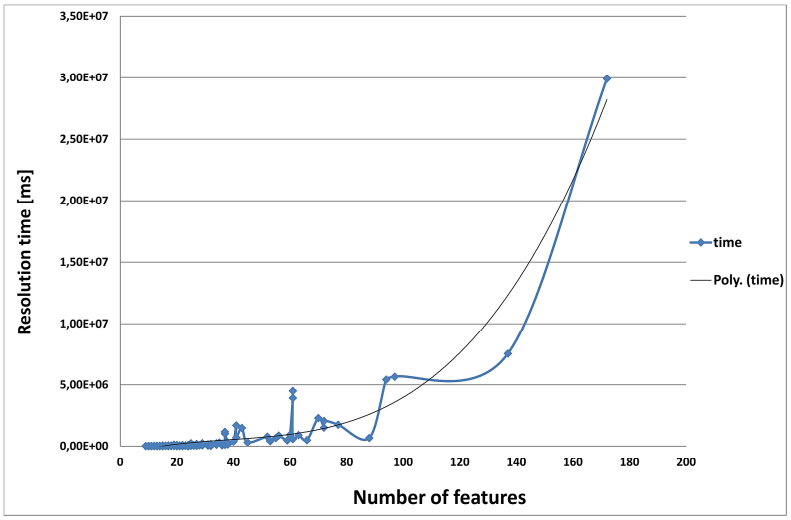

Fig. 11. Minimisation time estimation

Table IV. PACOGEN internal heuristics

Sort heuristics	Labelling heuristics	Minimisation cost functions
$depth\_father$	$first\_fail$	$f_1 = \mathsf{Max}_{k \in 1..4n^2} I_k$
$depth\_features$	$left\_most$	$f_2 = \sum_{k \in 1..4n^2} I_k$
$common\_operator$		

Table V. Pairs-ordering heuristics evaluation. Settings: number of FM = 224; labelling = first_fail; no minimisation; matrix size = 200.

Heuristic	$common\_operator$	$depth\_features$	$depth\_father$	no_sort
Average number of configurations	12.93	13.12	12.82	13.427

6.3.1. Pairs-ordering heuristics. In this experiment, we evaluated the three pairs-ordering heuristics (sort heuristics) on 224 FMs, to sense which one gives the smallest number of test configurations. For each considered FM, we fixed the labelling heuristic to $first\_fail$ and ran the process without minimisation, while changing the three pairs-ordering heuristics. The results for all 224 FMs were compared with the results obtained when PACOGEN is run without sorting. Table V shows the average number of test configurations computed for the three pairs-ordering heuristics for 224 FMs and the case without sorting. The results show that sorting allows us to generate up to 30% fewer test configurations than if no sorting is used (this result was obtained for 28% of 224 FMs). Further, comparing the sorting heuristics individually, the results show that the $depth\_father$ heuristic gives the smallest number of test configurations, followed by the $common\_operator$. The worst performance was observed for the $depth\_features$ heuristics.

6.3.2. Variable-selection heuristics. We evaluated two variable-selection heuristics on 224 FMs, in terms of their ability to minimise the number of test configurations. For each FM, we fixed the pairs-ordering heuristic to $depth\_father$ and ran the process without

Table VI. Labelling heuristic evaluation. Settings: number of FM = 224; sort = depth_father; no minimisation; matrix size = 200.

Model	$first\_fail$	$left\_most$
Aircraft PL	9	9
Arcade game	13	35
Car PL	6	7
Coche Ecologico	90	93
Connector PL	14	14
Doc generation	12	18
Fame DBMS	10	10
Inventory	12	14
Model Trans.	23	49
Movie App PL	6	6
Search Engine	11	13
Sienna	20	22
Smart Home	11	18
Stack PL	9	14
Web Portal	15	20

minimisation, while we were changing the labelling heuristic. The results show that the $first\_fail$ heuristic consistently outperforms the $left\_most$ heuristic. For 83% of the FMs, the $first\_fail$ heuristic provided up to 73% fewer configurations. For 17% of the FMs, these two metrics obtained the same number of configurations. The results for 15 FMs are shown in Table VI. We explain the better performance of the $first\_fail$ heuristic by its dynamic variable selection. This heuristic examines the domains of all unlabelled variables before making the selection, and then fills the matrix with the most constraint-sensitive pairs first.

6.3.3. Cost-functions. In this experiment, we evaluated how the cost-functions $f1$ and $f2$ impact the performance of the minimisation process. The evaluation was performed on 224 FMs. For each FM, we fixed the pairs-ordering heuristic to $depth\_father$ and the labelling heuristic to $first\_fail$, while we were changing the cost functions. The overall results show that using the cost function $f1$ gives from 31% to 46% fewer configurations compared with using the cost function $f2$.

Figure 12 shows the comparison of the cost functions f_1 and f_2 on the $Applications$ FM from SPLOT. X-axis represents time in seconds, while the Y-axis represents the number of remaining pairs that have to be set in the matrix in order to cover all feature interactions pairwise. For this FM, there are 1131 pairs initially. The solution of the constraint optimisation problem is found when the number of remaining feature pairs reaches 0. Figure 12 shows that PACOGEN found 9 different solutions using the cost function f_1, the smallest solution with 8 test configurations. Using the cost function f_2, PACOGEN found 2 solutions, the smallest with 16 configurations. The shape of the curves illustrates the backtracking used in the labelling process.

From this experiment, we concluded that the cost function $f1$ performs better in the minimisation process, compared to the cost function $f2$.

PACOGEN best configuration. Based on the experiments discussed above, we propose the following PACOGEN configuration: $depth\_father$ as a pairs-orderings heuristics, $first\_fail$ as a variable-selection heuristic, and $f1$ as a cost function. Using a PACOGEN methodology with these parameter values provides the most efficient usage of the tool in practice.

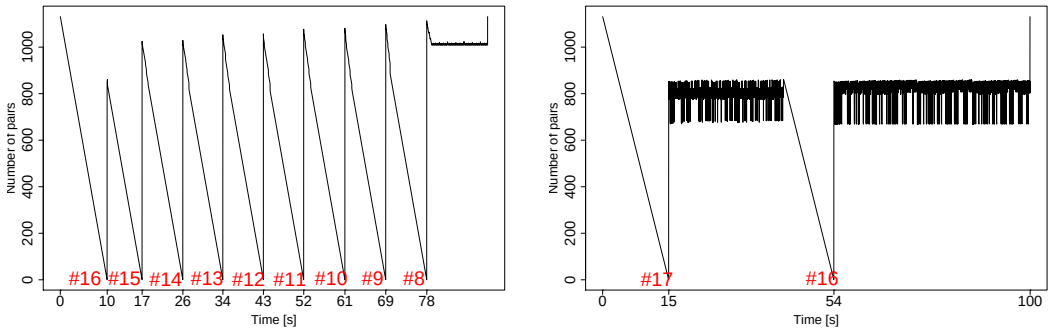

Fig. 12. Performance of minimisation for the $Applications$ feature model using: (a) cost function $f1$, and (b) cost function $f2$.

7. THREATS TO VALIDITY

Combinatorial pairwise testing can be helpful to decrease the effort of testing, but at the same time it can weaken the error-detection capability of a test suite, comparing to higher interaction strength techniques. It can bring false confidence in the quality of tested software, since it only protected against pairwise software errors. In this paper, we do not advocate unconditional usage of pairwise testing in practice. The benefit of pairwise testing is clearly context-dependent. Our motivation for the work on pairwise testing comes from our observation of the state of the practice. Our experience in working with industry shows that there are cases where approaches to generating tests are still mainly manual and unsystematic, with no strict quality requirements. Our experience also shows that in almost all cases, test managers are motivated to cut down testing costs. In combination, these two arguments produce a reasoned basis for applying pairwise testing.

PACOGEN time-out policy represents a capability of a tool to continue looking for a minimum set of configurations after the first solution has been found, until a timeout occurs. However, if the allocated timeout is too short, PACOGEN will produce suboptimal solution or, in a worst case, it will not be able to find a solution. On the other hand, a timeout too long will increase the size of the constraint model and will degrade the performance of the algorithm. Practising PACOGEN with feature models with different properties (e.g. size, constraints) will enable more reliable estimate of a timeout value.

8. CONCLUSIONS

Testing highly-configurable software is constrained by various factors, such as large configuration space or multiple dependencies between configuration elements that can make the generation of test configurations an inefficient process. In this paper, we propose a method and tool PACOGEN that helps validation engineers to manage these factors in automated, scalable and efficient manner. We model software variability using a feature model and specify the dependencies between features of the model. From the feature model, using the constraint programming techniques, our approach generates the exact minimum number of valid test configurations satisfying 2-way feature coverage.

We experimentally validated PACOGEN by comparing to the existing approaches and by applying it to an industrial context. The experiments reveal that PACOGEN outperforms the existing greedy approaches in minimizing the number of test configu-

rations. In industrial application, PACOGEN proved to decrease the time required by validation engineers in manual configuration by 85%, increasing the pairwise feature coverage, compared with the manually specified test set. Validation engineers evaluated the time-aware functionality of PACOGEN as beneficial, stating that it helps them to control and manage delivery deadlines easier.

In our experiments, we have observed that PACOGEN is less time-effective than the competing approaches, SPLCAT and MosoPolite, in finding the minimum set of test configurations. In fact, unlike the other two approaches, PACOGEN uses a constraint model where variables are closely linked together with constraints, and special constraints are used to enforce pairwise coverage. As a consequence, setting up a value for a given variable triggers numerous constraint propagations and value-assignment processes. However, PACOGEN often returns the true minimum solution, which is one of its main advantages over the other two approaches. This capability of PACOGEN is greatly useful when testing a configuration takes a lot of time and therefore it is important to generate as few configurations as possible. PACOGEN is also well suited to testing domains where variability models can be built early in the development process, so that longer test data generation time has low impact on the testing process.

In future work, first, we plan to extend PACOGEN to N-wise feature coverage, allowing the generation of test configurations with higher or lower interaction strength (e.g., $N = 3$ and $N = 1$). This involves the development of an algorithm to generate N-tuples in addition to a special additional constraint to enforce the coverage of N-wise interactions. Second, we plan to analyse the correlation between pairwise-coverage and error-detection effectiveness of a test suite. Third, we plan to integrate PACOGEN with the industry-strength variability management technology we are currently developing. We envision to fully integrate PACOGEN in our industrial partner's testing framework that runs in a continuous integration environment. However, this process is challenging, as it requires not only technical adoption, but also organisational and cultural changes at different levels in the company.

ACKNOWLEDGMENTS

We are immensely grateful to Marius Liaaen and the engineers from the QA department of Cisco Systems Norway, who provided us with the feedback on this work and participated in our experimental evaluation study. This work has been supported by the Research Council of Norway through the Certus SFI project.

REFERENCES

BATORY, D. 2005. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*. SPLC'05. Springer-Verlag, Berlin, Heidelberg, 7–20.

BELL, K. AND VOUK, M. 2005. On effectiveness of pairwise methodology for testing network-centric software. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*. 221 –235.

BENAVIDES, D., TRINIDAD, P., AND RUIZ-CORTS, A. 2005. Using constraint programming to reason on feature models. In *Seventeenth international conference on software engineering and knowledge engineering*.

BRYCE, R. C. AND COLBOURN, C. J. 2007. 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. ACM, New York, NY, USA, 1082–1089.

BRYCE, R. C. AND COLBOURN, C. J. 2009. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliab. 19,* 1, 37–53.

BRYCE, R. C., COLBOURN, C. J., AND COHEN, M. B. 2005. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th international conference on Software engineering*. ICSE '05. ACM, New York, NY, USA, 146–155.

CHVATAL, V. 1979. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research 4,* 3, 233–235.

COHEN, D., DALAL, S., FREDMAN, M., AND PATTON, G. 1997. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on 23,* 7, 437 –444.

COHEN, M., DWYER, M., AND SHI, J. 2007a. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*. 121–132.

COHEN, M., DWYER, M., AND SHI, J. 2008. 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.

COHEN, M. B., DWYER, M. B., AND SHI, J. 2007b. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis (ISSTA'07)*. London, UK, 129–139.

COHEN, M. B., GIBBONS, P. B., MUGRIDGE, W. B., AND COLBOURN, C. J. 2003. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. IEEE Computer Society, Washington, DC, USA, 38–48.

CZARNECKI, K. AND EISENECKER, U. 2000. *Generative Programming: Methods, Techniques, and Applications*. AddisonWesley.

CZARNECKI, K., HELSEN, S., AND EISENECKER, U. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice 10,* 1, 7–29.

ERIKSSON, M., BRSTLER, J., AND BORG, K. 2005. The pluss approach - domain modeling with features, use cases and use case realizations. In *Software Product Lines*, H. Obbink and K. Pohl, Eds. Lecture Notes in Computer Science Series, vol. 3714. Springer Berlin Heidelberg, 33–44.

GARVIN, B., COHEN, M., AND DWYER, M. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering 16,* 1, 61–102.

GRINDAL, M., OFFUTT, J., AND ANDLER, S. 2005. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability 15*, 167–199.

HARTMAN, A. 2005. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms*, M. Golumbic and I.-A. Hartman, Eds. Operations Research/Computer Science Interfaces Series Series, vol. 34. Springer US, 237–266.

HENTENRYCK, P. V. AND DEVILLE, Y. 1991. The cardinality operator: A new logical connective for constraint logic programming. In *ICLP* (2002). 745–759.

HERVIEU, A., BAUDRY, B., AND GOTLIEB, A. 2011. Pacogen: Automatic generation of pairwise test configurations from feature models. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. 120 –129.

HNICH, B., PRESTWICH, S. D., SELENSKY, E., AND SMITH, B. M. 2006. Constraint models for the covering test problem. *Constraints 11,* 2-3, 199–219.

JOHANSEN, M., HAUGEN, Y., FLEUREY, F., CARLSON, E., ENDRESEN, J., AND WIEN, T. 2012a. A technique for agile and automatic interaction testing for product lines. In *Testing Software and Systems*, B. Nielsen and C. Weise, Eds. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 39–54.

JOHANSEN, M. F., HAUGEN, O., AND FLEUREY, F. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*. MODELS'11. Springer-Verlag, Berlin, Heidelberg, 638–652.

JOHANSEN, M. F., HAUGEN, O., AND FLEUREY, F. 2012b. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC '12. ACM, New York, NY, USA, 46–55.

KANG, K., COHEN, S., HESS, J., NOWAK, W., AND PETERSON, S. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*.

KARATAS, A. S., OGUZTÜZÜN, H., AND DOGRU, A. H. 2010. 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*. 537–551.

KLAIB, M., ZAMLI, K., ISA, N., YOUNIS, M., AND ABDULLAH, R. 2008. G2way a backtracking strategy for pairwise test data generation. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*. 463 –470.

KUHN, D. R., WALLACE, D. R., AND GALLO, JR., A. M. 2004. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng. 30,* 6, 418–421.

LAWRENCE, J., KACKER, R., LEI, Y., AND KUHN, D. 2011. A survey of binary covering arrays. *Journal Of Combinatorial Designs 18*, 37–53.

LEI, Y., KACKER, R., KUHN, D. R., OKUN, V., AND LAWRENCE, J. 2008. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Softw. Test., Verif. Reliab. 18,* 3, 125–148.

LOPEZ-HERREJON, R., CHICANO, F., FERRER, J., EGYED, A., AND ALBA, E. 2013. Multi-objective optimal test suite computation for software product line pairwise testing. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. 404–407.

MEMON, A. M. AND SOFFA, M. L. 2003. Regression testing of guis. In *Proceedings of the 9th European software engineering conference*. ESEC/FSE-11. ACM, New York, NY, USA, 118–127.

MENDONCA, M., WKASOWSKI, A., AND CZARNECKI, K. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. SPLC '09. Carnegie Mellon University, Pittsburgh, PA, USA, 231–240.

MENDONÇA, M., BRANCO, M., AND COWAN, D. D. 2009. S.p.l.o.t.: software product lines online tools. In *OOPSLA Companion*. 761–762.

OSTER, S., MARKERT, F., AND RITTER, P. 2010. Automated incremental pairwise testing of software product lines. In *Software Product Lines: Going Beyond*. Lecture Notes in Computer Science Series, vol. 6287. Springer Berlin Heidelberg, 196–210.

PERROUIN, G., KLEIN, J., GUELFI, N., AND JÉZÉQUEL, J.-M. 2008. Reconciling automation and flexibility in product derivation. In *Software Product Line Conference (SPLC'08)*. IEEE Computer Society, Limerick, Ireland, 339–348.

PERROUIN, G., OSTER, S., SEN, S., KLEIN, J., BAUDRY, B., AND TRAON, Y. L. 2012. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal 20,* 3-4, 605–643.

PERROUIN, G., SEN, S., KLEIN, J., BAUDRY, B., AND TRAON, Y. L. 2010. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*. ICST '10. IEEE Computer Society, Washington, DC, USA, 459–468.

POHL, K., BÖCKLE, G., AND LINDEN, F. J. V. D. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

QU, X., COHEN, M. B., AND ROTHERMEL, G. 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ISSTA '08. ACM, New York, NY, USA, 75–86.

SALINESI, C., DIAZ, D., DJEBBI, O., MAZO, R., AND ROLLAND, C. 2009. Exploiting the versatility of constraint programming over finite domains to integrate product line models. In *Requirements Engineering Conference, 2009. RE '09. 17th IEEE International*. 375 –376.

SCHOBBENS, P.-Y., HEYMANS, P., TRIGAUX, J.-C., AND BONTEMPS, Y. 2007. Generic semantics of feature diagrams. *Computer Networks 51,* 2, 456–479.

STEFFENS, M., OSTER, S., LOCHAU, M., AND FOGDAL, T. 2012. Industrial evaluation of pairwise spl testing with moso-polite. In *Sixth Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12), Leipzig, Germany, January 25-27, 2012. Proceedings*. 55–62.

WILLIAMS, A. 2000. Determination of test configurations for pair-wise interaction coverage. In *Testing of Communicating Systems*, H. Ural, R. Probert, and G. Bochmann, Eds. IFIP Advances in Information and Communication Technology Series, vol. 48. Springer US, 59–74.

WILLIAMS, A. 2002. *Software Component Interaction Testing: Coverage Measurement and Generation of Configurations*. University of Ottawa.

YILMAZ, C., COHEN, M. B., AND PORTER, A. A. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on 32,* 1.

ZILBERSTEIN, S. 1996. Using anytime algorithms in intelligent systems. *AI Magazine 17,* 3, 73–83.