HAL
open science

Practical Pairwise Testing for Software Product Lines

Dusica Marijan, Arnaud Gotlieb, Sagar Sen, Aymeric Hervieu

To cite this version:

Dusica Marijan, Arnaud Gotlieb, Sagar Sen, Aymeric Hervieu. Practical Pairwise Testing for Software Product Lines. SPLC 2013, Aug 2013, Tokyo, Japan. hal-00859438

HAL Id: hal-00859438

https://inria.hal.science/hal-00859438v1

Submitted on 8 Sep 2013

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.

Practical Pairwise Testing for Software Product Lines

Dusica Marijan, Arnaud Gotlieb and Sagar Sen
Certus Software V&V Center
Simula Research Laboratory
Email: dusica, arnaud, sagar@simula.no

Aymeric Hervieu
INRIA Rennes Bretagne Atlantique
Rennes, France
Email: Aymeric.Hervieu@inria.fr

***Abstract*—One key challenge for software product lines is efficiently managing variability throughout their lifecycle. In this paper, we address the problem of variability in software product lines testing. We (1) identify a set of issues that must be addressed to make software product line testing work in practice and (2) provide a framework that combines a set of techniques to solve these issues. The framework integrates feature modelling, combinatorial interaction testing and constraint programming techniques. First, we extract variability in a software product line as a feature model with specified feature interdependencies. We then employ an algorithm that generates a minimal set of valid test cases covering all 2-way feature interactions for a given time interval. Furthermore, we evaluate the framework on an industrial SPL and show that using the framework saves time and provides better test coverage. In particular, our experiments show that the framework improves industrial testing practice in terms of (i) 17% smaller set of test cases that are (a) valid and (b) guarantee all 2-way feature coverage (as opposite to 19.2% 2-way feature coverage in the hand made test set), and (ii) full flexibility and adjustment of test generation to available testing time.**

Software product lines (SPLs) allow companies to efficiently increase the range of products diversity imposed by a diverse user base. SPL represents a set of similar products developed from a common core components and with some variations in functionality. Therefore, instead of developing a collection of single customized (but similar) products, we can mass-customize products by exploiting their commonalities and maximizing reusable variation through a product line. This approach brings benefits in terms of higher productivity, shorter time to market and cost reduction, but also challenges in managing variability throughout the whole product line lifecycle. This paper focuses on managing variability in test case generation for SPL.

Some companies have structured parts of their software development following the SPL concept, with software testing activities still mainly based on traditional techniques that are suited for single product development. When employed in testing SPL, such techniques and tools reveal scalability problems and high inefficiency beyond a certain number of product variants. Experienced test engineers manually create test configurations of mutually consistent software components leading to a suite of functional software tailored for different user bases. While such manual process is highly time consuming, projects are normally planned with strict schedules, imposing a strict time bound for testing activities. Any test case exceeding this bound is missed. Therefore, the objective of test engineers is to specify the smallest number of test cases in available amount of time such that specific coverage criteria are satisfied (for example, all two software feature interactions are tested) and that all test configurations are valid (all dependencies between features are satisfied). Given a very large configuration space, this manual task is extremely tedious and unsystematic, leading often to missed deadlines, insufficient test coverage and redundancy in test cases.

In our industrial setting we have talked to engineers and have identified three main challenges for making SPL testing practical. First, the time that is provided for testing is inflexible - testing must be completed on time. Second, tests often contain invalid software configurations that cause failures in execution. Third, there is a lack of measurable test coverage criteria. In this work we provide a framework that addresses each of these limitations and is effective in practice. The framework (1) utilizes an anytime algorithm to balance the tradeoff between a test generation time and a number of test cases, (2) allows automated checking for validity of test configurations, (3) leverages combinatorial testing to increase test coverage, and (4) provides systematic training for engineers to make modeling easier and more accurate.

The first step is manual extraction of variability in SPL. We present a methodology to identify the variation points in the SPL and construct the model of variability called a feature model. An instance of the feature model corresponds to a configuration of the SPL. We use our algorithm to automatically generate the minimal set of configurations that cover all pairwise interactions and satisfy all interdependencies between features of the feature model. The algorithm utilizes constraint optimization techniques to generate the set of test configurations of minimal size for defined time interval. We experimentally evaluate the framework on the industrial SPL and demonstrate the improvement over industrial testing practice.

The contributions of the paper can be summarized as follows:

- We identify a set of issues that must be addressed to make SPL testing work in practice.
- We provide a SPL modeling methodology and training that helps practitioners to extract and abstract variability in a SPL and specify it formally as a feature model with

configurable features and their inter-dependencies.

- We leverage an anytime algorithm that generates the minimal set of test configurations covering all pairwise interactions between features in a feature model in a given amount of time, while satisfying all feature inter-dependencies.
- We evaluate our framework on an industrial SPL and present experiments that demonstrate the improvement in (i) smaller test set size, (ii) complete 2-way feature coverage, (iii) absolute test case validity, and (iv) conformance of the testing process to the project schedule constraints, comparing to the industrial practice.

The paper is organised as follows. In Section I, we describe our industrial case study of a video conferencing system and the SPL testing problem emerging from it. Background information and the related work are presented in Section II. We describe our framework for practical SPL testing in Section III. The experiments to validate our approach are presented in Section IV. Section V discusses some threats to validity of the approach, while section VI concludes the paper.

I. Industrial Case Study and Problem Description

In this section we present the industrial case study of testing a video conferencing SPL (VCSPL) that motivates our research and discussion on testing challenges that arise in industrial setting. We illustrate the case study and some of its variations in Fig. 1.

The VCSPL under test comprises core functionality (providing basic video-conferencing) common to all variants, and a set of features used to configure the variants according to user specific needs, requirements or preferences. VCSPL can be configured based on factors such as product price, localization or hardware limitations enforced by customers. In particular, VCSPL can be configured using 78 features, such as interface (telnet, snmp, ssh, http, on screen display, touch panel), call control (point-to-point, multisite, multiway), camera control modes (local, far end), audio/video resolution/protocol/standards, network types (IPv6, VLAN, 802.1x) or supported languages. However, not all possible features can co-exist in a single product. For example, the multi-site call feature cannot be pared with the point-to-point feature in one video conferencing software configuration. Similarly, video conferencing system cannot be configured for 1080p60 video resolution without high definition feature support. In order to generate valid test cases, all constraints among features must be satisfied in test generation.

A typical scenario for a test engineer is to manually combine software features paying attention to the validity of combinations. In the interviews conducted with test department, engineers admitted that they are aware of inefficiency of employed testing process, which takes a lot of time, is unsystematic, does not lead to sufficient test coverage (for example, all n-way feature interactions) and most of all, does not accommodate projects with different time bounds. In close dialogue with the test department, we identified several most important problems in manual test configuration generation to be addressed.

Strict and variable time bounds for testing activities. In the VCSPL testing, one core problem is adjustment of the manual process to the dynamics of product development and delivery. Due to rapid market growth for the networking equipment, our partner as a leader in this field has to follow market needs to stay competitive. This induces large amount of innovation and development offering users more and better functionality, followed by unpredictable and unstretchable time bounds for testing activities. Products are released for testing with different deadlines, often irrespective of complexity or amount of software to be tested. Following the manual (time consuming) practice, successfully time-aligning testing activities with delivery schedules is of core interest for the test department.

Invalidity of test configurations. For the VCSPL, test configurations are made by selecting features from a large space of constrained values. Manually satisfying all feature constraints for all configurations gives rise to missed deadlines. On the contrary, accommodating deadlines results in skipping a number of configurations in testing, due to their invalidity. Furthermore, invalid configurations will lead to failures and will make result reports difficult to interpret; the failures can be due to faulty software or invalid combination of features. In the VCSPL testing, manually checking configurations for validity is informal, as it is dependent on engineer's experience, his interpretation and understanding of the documentation. The objective of the test department is to automate and formalize checking the validity of test configurations.

Insufficient test coverage. Manual process of generating tests for VCSPL lacks systematics. Creating exhaustive test set for the VCSPL variability space is infeasible. The configuration set that test engineers use represents the subset of an exhaustive set, but there is no formal rationale used for reduction. Test engineers are guided by criteria such as covering configurations relevant in their opinion, covering the most common configurations (from their experience) or inheriting and adapting configurations from previous projects. This process results in insufficient coverage of certain features on one hand, and overabundant feature coverage on the other. The latter means that tests share commonalities and increase redundancy of the test set. Optimal testing needs to use systematic criteria to select configurations. To be able to measure the adequacy of the configuration set, the goal of the test department is to use n-way feature coverage as a test coverage criteria.

II. Background and Related Work

Variabilities and commonalities in a SPL can be efficiently represented by feature models [2]. By combining the features, we make test configurations for the SPL. However, in the presence of high variability, simply combining the features leads to unmanageably many test configurations. One popular way to decrease the testing effort and keep the balance between effort and quality of testing is combinatorial approach [3]. Still, optimal SPL testing will require not only a test set of

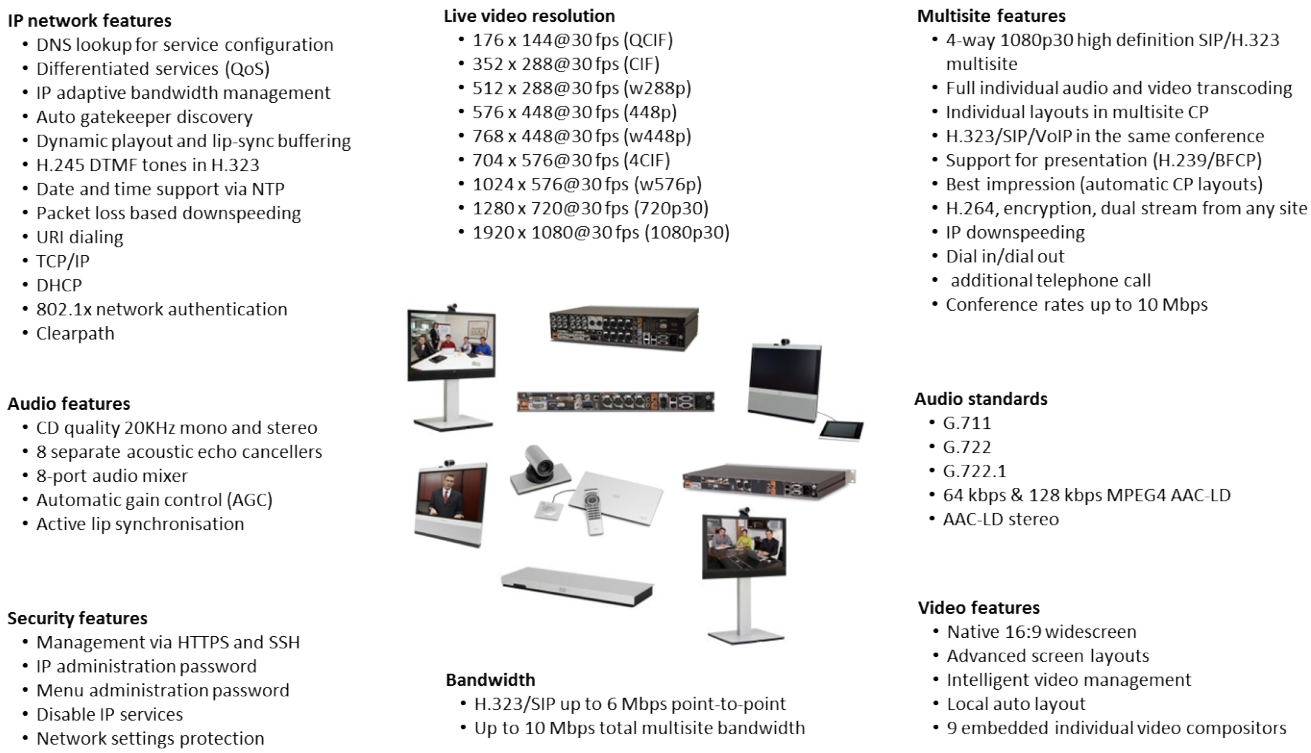

Fig. 1. Video conferencing software product line

manageable size, but the one that satisfies all dependences between features. An efficient way to address that problem is to use constraint programming techniques. Now we describe these three concepts in more detail, along with an overview of other approaches to SPL/variability testing proposed in the literature.

A. Feature modelling

Feature modelling has been introduced by Kang [4] as a compact and hierarchical representation of products in a SPL. The representation captures features that are related using mandatory, optional and alternative relations and that can be combined to build a product. Dependencies among the features are represented by require and exclude relations. These dependencies (constraints) will restrict invalid combinations in test generation. Fig. 2 shows a sample feature diagram of the VCSPL feature model. The model specifies that the VCS supports making calls, which can be either $P2P$ or $Multisite$ calls. For the $Multisite$ calls, possible resolutions are either $3x1024x576Max$ or $3x720p30Max$ or $1080p30/720p60Max$. Optionally, the VC software can support $720p60$ premium resolution, $1080p30$ premium resolution, or both resolutions. The configuration that supports $3x720p30Max$ or $1080p30/720p60Max$ resolution multisite calls must have $Premium\ resolution$ feature.

B. Combinatorial interaction testing

Combinatorial interaction testing (CIT) is a recognized software testing technique introduced by Cohen [5], used to test interactions between software parameter values. The effectiveness of CIT is based on the observation that software failures are often due to interactions between only few (t) software parameters [6], [7], [8]. As reported in the NIST study of the fault databases for several real-life systems, all faults were caused by no more than six factors [3]. A t-way testing covers all t-way combinations of input parameters and can detect faults caused by interactions of t or less components. The most often used CIT application in practice is pairwise or 2-way testing. Pairwise testing requires that every pair of values is present at least once in a set of test configurations. It was shown to be both time efficient and effective for most real case studies [9]. Besides the test coverage requirement from our industrial partner, this reason motivated us to focus on pairwise feature interactions in designing our framework for SPL testing.

One well-known approach proposed in literature for generating test configurations that satisfy pairwise coverage criterion is AETG tool [5]. Although AETG is able to generate t-wise covering arrays, it is based on greedy approach and thus does not guarantee reaching the global minimum number of test configurations with pairwise coverage. CTE-XL tool [10], based on classification tree method, allows users to generate pairwise and three-wise covering test sets, while handling constraints among input parameters. However, constraints are handled in a passive way, by checking generated test configurations and possibly refuting inconsistent combinations. This approach is insufficient for larger number of variables

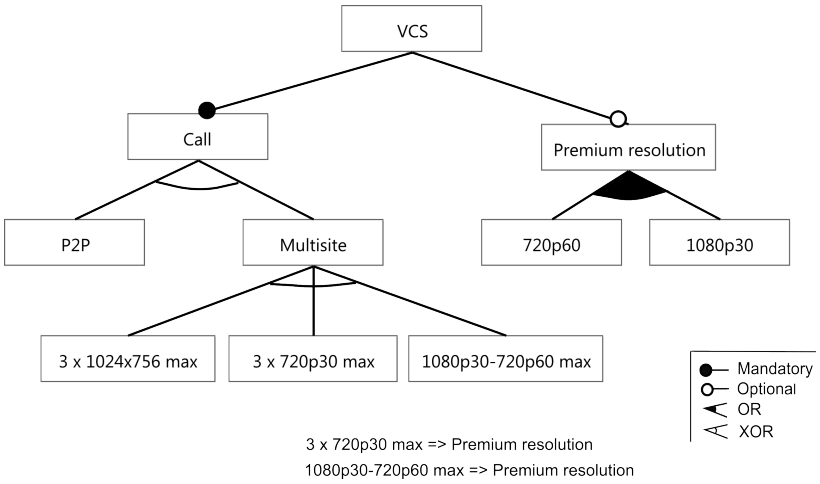

Fig. 2. Video conferencing SPL feature diagram

and constraints. Perrouin proposes transforming the feature models into Alloy declarative programs, in order to select valid configurations, with respect to the initial model [11]. This approach faces scalability issues, since it uses generate-and-test approach to select valid pairs that must be covered (generate all possible pairs, then filter all invalid pairs). Besides, the Alloy model needs to be transformed in CNF form before it can be solved by the underlying SAT solver used by Alloy. Oster uses a greedy and ad-hoc algorithm based on the maximum number of valid pairs within each configuration, where dependencies between features are handled by the flattening transformation over the feature model and AETG algorithm [12]. His work represents the extension of the method implementing greedy approach to solve the problem of dependencies [13]. Johansen generates test configurations with 1-3-way coverage from large feature models using greedy algorithm to enforce all pairs in a set of configurations [14]. However, none of these approaches does not guarantee the selection of the minimal number of combinations to cover pairwise interactions, as opposite to our approach. Furthermore, they do not allow specifying upper time bound for generating test configurations. Our framework is designed to generate the minimal number of pairwise-covering valid test configurations from a feature model with constrained relations between configuration parameters in a given amount of time. Specifying a (reasonably-long) time interval for test generation has high practical value, as it allows adapting the process to the available testing time.

C. Constraint Programming

Constraint programming is an artificial intelligence paradigm describing the relations between variables in the form of constraints and automatically solving these constraints in a declarative way. Modelling and solving a constraint problem requires defining variables and specifying their domain, identifying constraints between the variables, and enumerating values (labelling). The rest of the problem is handled by the constraint solving algorithms. In addition, this paradigm can deal with cost optimization functions and search for an optimal solution in a large search space. Another benefit of constraint programming is the ability of defining new special-purpose constraints (global constraints) for fine-tuned modeling and time-aware optimization, useful for problems such as finding the minimal set of valid test configurations with pairwise feature coverage in a given time interval from a FM.

III. Framework for practical pairwise testing of SPL

We propose a tool-supported framework for practical SPL testing, designed to solve the realistic problem that emerged from our collaboration with industry; the problem of variability in test configuration generation for SPL. The framework combines feature modelling, combinatorial interaction testing and constraint programming, as well-recognized techniques used for automatic test generation in the presence of variability [11]. The main idea consists in extracting variability in a SPL that is specified in the form of feature model with feature interdependencies. The feature model is explored by a tool-supported algorithm PACOGEN [1], to generate the minimal set of test configurations covering all valid pairwise interactions between the features in a given time interval. The configurations are fed into the test case generator to generate the executable tests cases. The test case generator represents the integration layer between PACOGEN test configuration generation algorithm and VCSPL test execution framework.

The general overview of the framework is given in Fig. 3. The methodology underlying the framework is logically divided in two parts: SPL modeling methodology and PACOGEN test configuration generation algorithm.

A. SPL Modeling Methodology

Our modeling methodology originates from the industrial case study introduced in Sec. I. Nevertheless, the methodology is generic and can be applied to other domains dealing with variability in software systems.

Extracting variability in a SPL by means of a FM means first identifying features, the building blocks of the FM. One

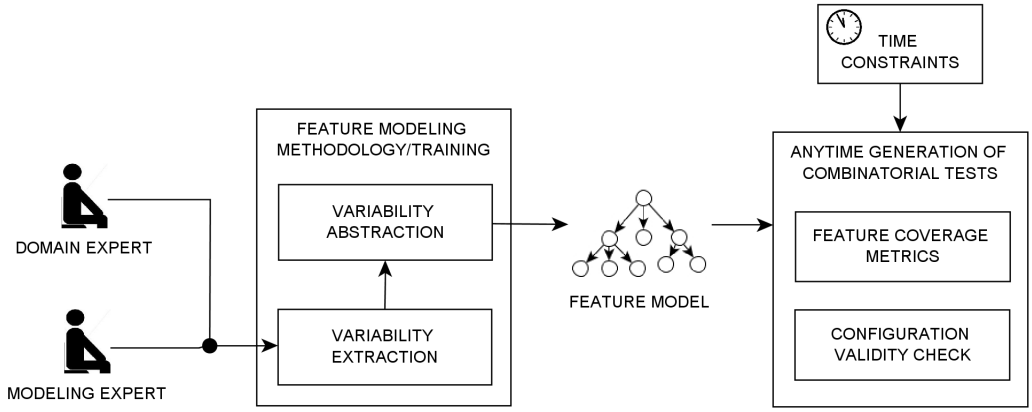

Fig. 3. Framework overview

of the early definitions defines a feature as a prominent and distinctive aspect or characteristic that is visible to various stakeholders [4]. In practice, however, definition of a feature comes from a common view of software variable characteristics for a test engineer and a modeling expert. With respect to SPL feature modeling, we define a feature (and related concepts, including a feature model, constraint and configuration), based on the extension of the metamodel [15] and the Free Feature Diagram [16] as follows:

- *Feature* is a representation of software variable characteristics in SPL;
- *Primitive feature* is a specific value of a feature (concrete artefact in SPL);
- *Feature model (FM)* is a hierarchical representation of a set of features;
- *Constraint* is a relation over several features in a FM. We distinguish between different types of constraints, such as hierarchical constraints, cross-tree constraints and CNF constraints:
  1) *Hierarchical constraints* are relations among father and child features, based on the operators AND, OR, XOR, OPT, CARD;
  2) *Cross-tree constraints* are binary relations among any pair of features in a FM, based on the operators REQUIRE, MUTEX;
  3) *CNF constraints* are non-binary relations among any subset of features that can be expressed as a boolean formula in Conjunctive Normal Form (CNF);
- *Configuration* is a combination of primitive features from a FM;
- *Valid configuration* is a configuration that satisfies all FM constraints;

SPL variability modeling is accomplished in three steps. First, identifying the features, followed by defining primitive features for the identified features and last, specifying the relations between the features (constraints). For optimal SPL variability modeling, this process requires the combination of domain and modeling expertise. A domain engineer (often a test engineer) provides contextual information of the variability, including the features, their primitive features and relations between the features. The modeling expert provides modeling principles; how to formally represent the features and their constraints. Besides, the modeling expert helps to determine the right FM abstraction level, at which SPL variable characteristics will be specified. The challenge is to choose a good balance between the precision of a FM and its complexity. Usually, a domain expert tends to use too low abstraction level, what would make the model complex for processing. Besides, large number of features in a FM potentially leads to a large number of test configurations. Therefore, the goal is to keep the FM size and complexity within manageable bounds, yet sufficiently detailed to generate valid test configurations. Our good-practice modeling guidelines recommend, when possible, splitting complex FM into logically divisible smaller FMs and combining them in processing. This way, maintaining the FM (done manually) is lightened.

In our proposed methodology, FM is built manually. The issue with manual SPL variability modeling in practice is that it requires specific knowledge that test engineers rarely poses. A way to overcome that problem is to provide a training on modeling for practitioners. Effective training should be well suited to the modeled context and based on realistic examples. It should, for example, help test engineers to define a feature in the given context easier. When the test engineers are familiar with modeling, manually building a FM has a low impact of their productivity. The FM is usually made once and updated as the SPL evolves. In fact, a positive effect of making a FM manually is that it produces a better understanding of the SPL for the engineers, for example, how are particular components related or what components are mutually dependent. Often, the engineers developing/testing a single system component miss a wider picture of the system. In order to make a FM, these experts have to make a common understanding on the system structure and the relations between its components.

domain([A,B,C],0,1), or(A,[B,C]), B = 1.

Fig. 5. A simple illustration of constraint propagation

B. Test Configuration Generation Algorithm

Generating the minimal set of test configurations from a FM, enforcing pairwise feature coverage and satisfying feature constraints can be seen as a constraint optimization problem. Relations between the features are captured by the constraint model, while minimizing the number of test configurations is implemented through our *time-aware constraint solving and minimization* technique. First, the algorithm takes a FM as input and transforms it into an internal constraint model. The model is represented as a matrix where columns specify features and rows represent configurations. The matrix is dynamic, since the minimal number of test configurations is unknown at the time of matrix allocation. The constrained matrix for the VCSPL is shown in Fig. 4.

To ensure pairwise coverage, each pair of features must appear in the matrix. This is implemented through a special global constraint (GC) that enforces the specific pair of values to be included in the vectors. For example, $\mathsf{GC}(I, ([X_1, X_2, X_3], [Y_1, Y_2, Y_3]), (0, 0))$ constrains the unknown row I of the matrix to contain the pair $(0, 0)$, meaning that the corresponding features should be included within the test configuration of rank I. During the solving process, if I is instantiated to value 2 then $(X_2, Y_2) = (0, 0)$, while if X_3 is instantiated to value 1, then value 3 will be removed from the domain of I. The pair value (X_3, Y_3) cannot be equal to $(0, 0)$ in this case. A detailed implementation of our global constraint (along with detailed PACOGEN algorithm description) is given in [1]. To filter out the invalid pairs PACOGEN uses the proposed constraint model and the constraint propagation mechanism. Particularly, the constraint propagation mechanism enables propagating the impact of a feature selection or de-selection from one feature to another. An example of constraint propagation mechanism is shown in Fig. 5. Request, $domain([A, B, C], 0, 1)$ specifies that A, B and C features are integer variables with possible values of 0 and 1, while $Or(A, [B, C])$ is a hierarchical constraint denoting the link between a father A and its two children B and C. $B = 1$ constraints feature B to be selected. The result of constraint propagation is $A = 1, C$ in 0..1. Feature A is thus automatically selected, while feature C is left unbounded.

Given the constraint matrix, constraint optimization algorithms are used to generate the set of test configurations of minimal size for the specified time interval. We call this property a *time-aware constraint solving and minimization* [1]. It is based on the combination of constraint propagation, search space exploration and anytime algorithms. A timeout value is set for the algorithm, after which the minimization process will be stopped. This means that given a test generation time, the algorithm generates a near-optimal set of test configurations. Near-optimal solution is found quickly and most of the remaining time is used to prove that no better solution exists. This is a property common to anytime algorithms [17]. It has a high practical value, as it allows test engineers to specify how much time they want to spend generating tests. In this way, test engineers can balance between the minimal size of the test set and time spent in test generation.

IV. Experimental evaluation

Our goal in this section is to evaluate the proposed framework in terms of its ability to improve industrial practice in SPL testing with respect to the main three identified problems: (i) strict and variable time bounds for testing activities, (ii) invalidity of test configurations, and (iii) insufficient and informal test coverage. The evaluation of the PACOGEN test configuration generation algorithm, compared to the existing similar solutions was reported in [1] and is out of scope of this paper. The experiments in this section will address the following questions:

- Does the proposed framework help test engineers to accommodate testing deadlines?
- Does it ensure test configuration validity?
- How much it improves test coverage?
- Does it reduce test configuration set size?

A. Experiment Setup

We consider VCSPL described in Sec. I as the SPL under test. We apply our framework to testing VCSPL and analyse the effects on the current industrial practice. The FM of the VCSPL and time value that specifies how long the algorithm will run are inputs to the framework, and the minimized set of valid test cases with pairwise feature coverage is generated as an output.

B. Experiment Procedure

To capture software variable characteristics in VCSPL and represent them in the form of features and feature interdependencies, it was necessary to deeply understand VCSPL structure and functionality. We built the VCSPL FM in small increments, together with domain experts (test engineers). We used Pure::Variants tool [18] to build the FM. We specified feature interdependencies using OR, XOR, $mandatory$ and $optional$ operators. Given as input to PACOGEN algorithm, the FM is transformed into the constraint models. The constraint models are explored to find the minimal number of test configurations with pairwise feature coverage and satisfying all FM constraints for the specified running time value. Test configurations are automatically supplemented with the test input and output data to form executable test cases.

To answer the first experimental question, we measured the performance of the framework over time, as VCSPL was evolving. There were five major changes in the VCSPL, what corresponds to five sub-projects, each with different schedule constraints. On every VCSPL change, we would update the FM, generate the test cases and use them in test execution. The framework was used in parallel with manual test generation, so that we can analyse and compare the results. The changes in the FM consisted in adding and removing

$$\begin{pmatrix} VCS_1 & CALL_1 & P2P_1 & MULTISITE_1 & \dots & 720p60_1 & 1080p30_1 \\ VCS_2 & CALL_2 & P2P_2 & MULTISITE_2 & \dots & 720p60_2 & 1080p30_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ VCS_n & CALL_n & P2P_n & MULTISITE_n & \dots & 720p60_n & 1080p30_n \end{pmatrix}$$

Fig. 4. Constrained matrix for the VCSPL feature model

features and constraints between the features and updating existing constraints between the features. The size of the FMs varied from 78 to 83 features for these five sub-projects.

Regarding test configuration validity, one of the functionalities of our proposed framework is generating test configurations from a constrained FM such that all constraints between features are satisfied. This means that generated test cases represent valid SPL configurations and any failure in test case execution denotes a fault in software. On the contrary, satisfying FM constraints manually is tedious and as a consequence, test sets often contain test cases that fail due to invalid software configurations. To answer the second experimental question, we analysed test reports after executing two different manually specified test sets for the VCSPL to identify how many failures were caused by test configuration invalidity.

Regarding the test coverage, the algorithm generates the minimal set of valid test configurations to cover all 2-way feature interactions in a FM. This means that all pairs of software features are included in the test set at least once. To answer the third experimental question, we examined the level of 2-way feature interactions in the manually generated test set and compared it to the 100% pairwise feature coverage provided by the automatically generated set.

To answer the fourth experimental question, we compared the size of automatically generated configuration set that covers all 2-way feature interactions and satisfies all FM constraints with the size of manually generated test set for the five VCSPL versions.

C. Experimental Results

1) Accommodating testing deadlines: When employing the proposed framework, total time to generate test cases comprises time to build the FM and time required to generate executable test cases. For the VCSPL v1, time taken to build the FM of 87 features equals 57 man-hours. This value does not include time spent to learn feature modeling; it is supposed that the test team is familiar with modeling principles. For all subsequent FM, time was significantly less, from 9 to 15 hours, as all successive VCSPL models were made by updating the models from the previous version. For the VCSPL v1 FM, test case generation took 12.3 hours. For all subsequent FM, test case generation required approximately the same amount of time, as the FMs had similar number of features and constraints. Fig. 6 compares time spent generating test cases for five versions of VCSPL using the proposed framework and the manual approach.

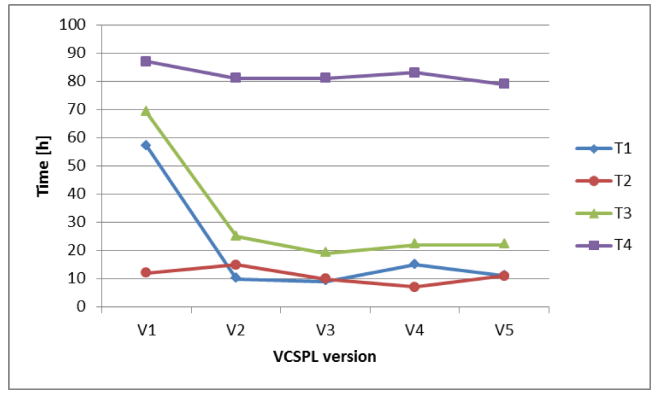

Fig. 6. Comparison of the framework running time (including feature modeling and test case generation) and time required for manual test case generation for five subsequent versions of VCSPL. T1 = Time to build FM; T2 = Time given to the test configuration generation algorithm; T3 = Total running time of the framework; T4 = Time required by manual test case generation

Time to build the FM (T1) has high value for the VCSPL v1 FM. T1 significantly decreases and remains almost constant for all subsequent FM. It represents the concept of upfront investment that pays off by reusing created model. Time for test case generation algorithm (T2), given as input to the framework, was chosen by experienced test engineers based on available total testing time. For all five cases, in average, time given to the framework was 7 times less than time taken by manual test generation. After obtaining the test sets for the given time, we analysed whether the test sets would be further minimized if more time was allocated for the algorithm. In one out of five cases the test set was reduced for only 6 % after the algorithm run longer. Such results are due to the fact that a near-optimal solution is found quickly and most of the remaining time is spent proving that there is no better solution for the given problem. These experiments show that the framework can be efficiently used to generate test configurations for the available testing time, not compromising the quality of the configuration set (in terms of its size). Similarly to T1, total running time of the framework (T3) is high in beginning and decreases to almost constant value with FMs reuse. Unlike T3, time required by the manual test case generation (T4) keeps similar high value over all five VCSPL versions.

2) Test configuration validity: We analysed test reports after executing two manually generated test sets for the VCSPL and found that 11 % of failures in one case and 10% in another came from invalid test configurations represented by the test

sets. On the contrary, automatically generated configurations fully satisfied the validity criterion.

3) Test case coverage: By analysing manually generated test set for the VCSPL, we extracted the feature pairs covered by the set and found that they satisfy 19.2% of total VCSPL pairwise feature coverage, what corresponds to 5 times lower pairwise coverage compared to automatically generated test set. These results show that many interactions between two features are missed by test engineers. Besides, there is uncertainty on the coverage provided by the manually specified test set. Usually due to time constraints, test engineers make tests that include most important interactions in their opinion, irrespective of the fact how many two-way or more-way feature interactions are tested, since there is no support for such analysis in the manual practice.

4) Configuration set size: Compared to the manually generated test configuration sets, automatically generated ones were 17% smaller in average, for the five VCSPL versions. The results are shown in Fig. 7. The reduction of 17% in configuration set size is not big, taking into account effort needed to make the FM. However, these results are expected, since automatically generated configuration sets satisfy all FM constraints and cover 5 times more 2-way feature interactions than the manually generated ones. The benefit of the proposed framework is more valuable in terms of higher quality and adequacy of test sets than their size. The objective of the test department is not only to generate the smallest test set, but the test set that will in addition guarantee complete pairwise feature coverage and the validity of configurations.

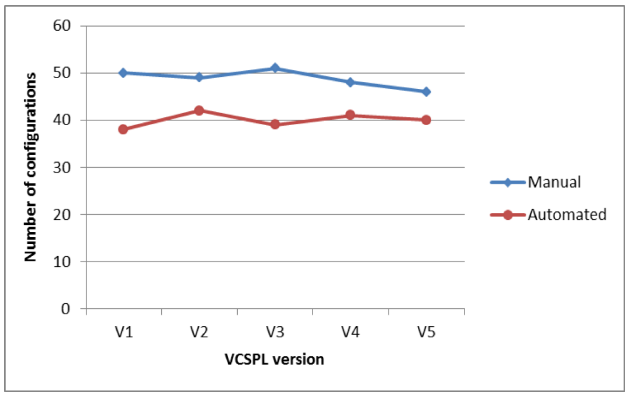

Fig. 7. Comparison of the test configuration set size for manual and automated generation approach, for five subsequent versions of VCSPL.

V. Threats to Validity

Our framework for testing SPL embraces two steps: modeling the variability in a SPL and generating valid test configurations from the variability model. While the latter step is completely automatic, SPL variability modeling requires manual work. Domain and modeling experts work together to specify the feature model that is detailed enough to support test case generation, yet maintainable and not too complex for processing. Although modeling requires some time and expertise, on the other hand it increases understanding of the modeled SPL for domain experts. Besides, if the framework is used continuously as the SPL evolves, building a FM takes constant low effort after the first SPL version FM has been built, as shown in our experiments, since the successive FMs can be built by modifying the previous SPL FMs.

VI. Conclusions

We proposed an industry-strength framework for automated pairwise testing of SPL, with an objective to generate the minimal set of test configurations that are valid and cover all pairwise feature interactions. A special feature of the framework allows specifying a time interval for the configuration generation algorithm, seen from an industrial perspective as a way to balance the quality and effort of testing. The framework is based on feature modelling, combinatorial interaction testing and constraint programming techniques. It consists of the modeling methodology that explains how to formally specify variability in a SPL and the test configuration generation algorithm. Variability in a SPL is extracted and specified as a feature model with mandatory, optional, and inter-dependent software features. The feature model is analysed to automatically generate a minimized set of valid test configurations covering all 2-way feature interactions. The approach is mainly automatic, with an exception of collaboratively specifying the feature model with domain experts. We evaluate the framework on an industrial video conferencing SPL and present the experiments showing the improvement over the industrial testing practice in terms of (i) 17% smaller set of test cases that are valid and guarantee all 2-way feature coverage (as opposite to 19.2% 2-way feature coverage in the hand made test set), and (ii) full flexibility and adjustment of test generation to available testing time.

As a future work, we plan tuning the constraint optimization algorithm to enable generating more balanced test configurations with respect to feature pair occurrence. Currently, the algorithm enforces at least one occurrence of each feature pair in the set of test configurations, while it is desirable to enforce several pair occurrences to improve the quality of test configurations. This would require designing more complex cost functions and introducing random choices in the search space heuristics for the test configuration generation algorithm.

Acknowledgment

We thank to our industrial partner for their feedback and valuable and interesting discussions. This research is supported by the Research Council of Norway.

References

[1] A. Hervieu, B. Baudry, and A. Gotlieb, "Pacogen: Automatic generation of pairwise test configurations from feature models," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium*, 2011, pp. 120–129.

[2] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Techniques, and Applications*. AddisonWesley, 2000.

[3] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," 2002.

[4] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[5] D. Cohen, S. Dalal, M. Fredman, and G. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, p. 437444, 1997.

[6] D. R. Kuhn and D. D. Wallace, "Software fault interactions and implications for software testing," *IEEE Trans. on Software Eng.*, vol. 30, no. 6, pp. 418 – 421, 2004.

[7] K. Bell and M. Vouk, "On effectiveness of pairwise methodology for testing network-centric software," in *Int. Conf. on Information and Communications Technology*, 2005, pp. 221–235.

[8] M. Grindal, J. Offutt, and S. Andler, "Combination testing strategies: a survey," *Software Testing, Verification, and Reliability*, vol. 15, pp. 167–199, 2005.

[9] [Online]. Available: http://www.pairwise.org/results.asp

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

[11] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Automated and scalable t-wise test case generation strategies for software product lines," in *International Conference on Software Testing (ICST'10)*, Paris, France, 2010.

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

[13] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[14] M. F. Johansen, O. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference - Volume 1*, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 46–55. [Online]. Available: http://doi.acm.org/10.1145/2362536.2362547

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

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

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

[18] D. Beuche, "Modeling and building software product lines with pure::variants," in *Software Product Line Conference, 2008. SPLC '08. 12th International*, sept. 2008, p. 358.