Software Qual J (2012) 20:605–643
DOI 10.1007/s11219-011-9160-9

Pairwise testing for software product lines: comparison of two approaches

Gilles Perrouin · Sebastian Oster · Sagar Sen · Jacques Klein · Benoit Baudry · Yves le Traon

Published online: 31 August 2011

Abstract Software Product Lines (SPL) are difficult to validate due to combinatorics induced by variability, which in turn leads to combinatorial explosion of the number of derivable products. Exhaustive testing in such a large products space is hardly feasible. Hence, one possible option is to test SPLs by generating test configurations that cover all possible *t* feature interactions (*t*-wise). It dramatically reduces the number of test products while ensuring reasonable SPL coverage. In this paper, we report our experience on applying *t*-wise techniques for SPL with two independent toolsets developed by the authors. One focuses on generality and splits the generation problem according to strategies. The other emphasizes providing efficient generation. To evaluate the respective merits of the approaches, measures such as the number of generated test configurations and the similarity between them are provided. By applying these measures, we were able to derive useful insights for pairwise and *t*-wise testing of product lines.

G. Perrouin (✉)
University of Namur, PReCISE, B-5000 Namur, Belgium
e-mail: gilles.perrouin@fundp.ac.be

S. Oster
Real-Time Systems Group, Technische Universität, Darmstadt, Germany
e-mail: sebastian.oster@es.tu-darmstadt.de

S. Sen
INRIA Sophia Antipolis, 2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex, France
e-mail: sagar.sen@sop.inria.fr

J. Klein · Y. le Traon
University of Luxembourg, SnT and LASSY, Campus Kirchberg, Luxembourg-Kirchberg, Luxembourg
e-mail: jacques.klein@uni.lu

Y. le Traon
e-mail: yves.letraon@uni.lu

B. Baudry
Triskell Team, IRISA/INRIA Rennes Bretagne Atlantique, Rennes, France
e-mail: bbaudry@irisa.fr

Keywords Model-based engineering and testing · Test generation · *t*-wise and pairwise · Software product lines · Alloy

1 Introduction

When a company rapidly derives a wide range of different products, a key challenge is to ensure correctness and safety of most of these products (if not all) at a low cost. Software Product Line (Pohl et al. 2005; Clements and Northrop 2001a) (SPL) techniques and tools allow to engineer such families of related products. However, they rarely focus on testing the SPL as a whole. A software product line is usually modeled with a feature diagram (Kang et al. 1990), describing the set of features in the SPL and specifying the constraints and relationships between these features. For example, mandatory features as well as mutually exclusive ones can be described. As a result, from a feature diagram, it is possible to derive products by selecting a set of features that satisfy all the constraints. The product is a software system built by composing the software assets that implement each feature (Perrouin et al. 2008).

Product line testing consists in deriving a set of products and in testing each product. Although required to achieve 100% SPL coverage, testing each product individually is rarely feasible in practice. In the automotive domain, each car of a certain brand may have a different software configuration induced by different choices made in the feature diagram. Generally, the number of possible configurations induced by a given feature diagram grows exponentially with the number of features quickly leading to millions of possible configurations to test. As a result, test engineers are seeking solutions to reduce the number of configurations to test in order to meet release deadlines and cost constraints.

Previous work (Cohen et al. 1997; Kuhn et al. 2004) has identified combinatorial interaction testing (CIT) as a relevant approach to reduce the number of products for testing. CIT is a systematic approach for sampling large domains of test data. It is based on the observation that most of the faults are triggered by the interactions between a small numbers of variables. This has led to the definition of pairwise (or 2-wise) testing. This technique selects the set of all combinations so that all possible pairs of variable values are included in the set of test data. Pairwise testing has been generalized to *t*-wise testing, which samples the input domain to cover all *t*-wise combinations (Lei et al. 2008; Bryce and Colbourn 2009). In the context of SPL testing, this consists of selecting a small set of products in which all *t*-wise feature interactions occur at least once.

Such algorithms enable to drastically reduce the number of configurations to test from millions or billions to a few dozens or hundreds, making the testing effort tractable. However, questions remain with respect to the merits of CIT for everyday SPL testing practice. CIT algorithms require the use of constraint solvers to generate pairwise configurations. Constraint satisfaction problems (CSP) are known to be NP-complete in the general case. This inevitably leads to scalability issues. These issues have to be handled pragmatically because the “phase transitions” which distinguish tractable problems from untractable ones are not known *à priori* (Monasson et al. 1999). Also, these algorithms do not natively consider constraints between features. As such constraints are common in SPL modeling, extensions of CIT algorithms are needed (Calvagna and Gargantini 2009, 2008; Bryce and Colbourn 2006; Cohen et al. 2007). Furthermore, SPL engineers are used to design feature diagrams but not to write CNF clauses—inputs of CSP solvers—that are numerous for any realistic case. Hence, solutions have to be proposed to automatically derive such inputs from feature diagrams handled by modeling tools. Finally, to increase confidence of test engineers in the viability of such techniques for SPL testing, considerations about the efficiency, quality, and flexibility of the generation approach are important. These considerations lead to questions about test diversity, size of test suites, or computation time.

1.1 Contribution

In this paper, we report on our efforts toward solving the aforementioned questions. In particular, we describe two approaches developed by the authors (Oster et al. 2010; Perrouin et al. 2010), exhibiting different concerns and choices in the implementation of the pairwise algorithms for testing software product lines. Our goal is to provide decision criteria to the software tester willing to apply *t*-wise testing for SPL. To support these criteria in an objective way, we generalize measures, initially presented in Perrouin et al. (2010) to qualify any *t*-wise generation algorithm. For example, we are able to characterize whether and how optimally the *t*-wise criteria are met by analyzing the number of times a given interaction appears in a generated test suite. Such a value can be used by testers to gain confidence that their tests will cover the same interaction in various cases. Providers of *t*-wise toolsets can also use such a measure to improve their implementations. Another important criterionis the similarity of generated test configurations: depending on their needs, testers validate small variations in important products or test the SPL broadly. To assess this, we form the concept of test configuration similarity based on a distance metric. We present these measures in Sect. 4, and this is the first contribution of this paper.

The second contribution is formed by the lessons we have learned in applying these measures on several feature diagrams with our two approaches. By applying our measures, we are able to highlight the particular impact the choice of implementation technologies (and the theories underlying them) has on generated test suites. We confirm here previously identified tendencies in our previous work (Oster et al. 2010; Perrouin et al. 2010). Our conviction is that what we have learned with these approaches is transferable to other approaches as well, serving as an evaluation framework for pairwise and *t*-wise testing of software product lines.

1.2 Outline

The remainder of this paper is structured as follows: Section 2 provides the background of our approach. There, we first introduce the context of our contribution together with our running example, which we use throughout this paper. Furthermore, those preliminaries give a short introduction to feature modeling and SPL testing and define the vocabulary used to introduce and compare both approaches. A problem statement describing the challenges of our contribution is provided at the end of Sect. 2. Both approaches are described in Sect. 3 using our running example. To compare both approaches, we define a comparison framework in Sect. 4 by defining criteria for comparison. The actual experimentation is presented in Sect. 5. Section 6 deals with experimental results and summarizes pros and cons of each approach in order to assist the tester in his choice. Section 7 discusses related work. Finally, Sect. 8 concludes this paper and discusses the ongoing research and open research questions.

2 Background

In this section, we provide the preliminaries of our contribution describing the use of feature models within SPL engineering and how it can be related to SPL testing purposes.

Furthermore, this section provides the detailed problem description we address in this paper.

2.1 Context and example

In this paper, we address the problem of testing software developed according to the SPL paradigm— “a set of software intensive systems, that share a common, managed set of features satisfying the specific needs of a particular market segment or mission, and that are developed from a common set of core assets in a prescribed way” (Clements and Northrop 2001b)—to effectively address differences required by each product while reusing common parts to increase productivity. Hence, the key to success in any SPLE approach is the sensible management of commonalities and differences or *variability management* (Metzger et al. 2007). One of the most practical techniques is *feature modeling* (Kang et al. 1990) that aims at representing the common and variable *features*[1] of a product family. Feature modeling can be used to document and analyze variability during any phase of the SPL development lifecycle. Hence, every stakeholder can manipulate features “as is”, independently of the kind of variability and the level of abstraction. Moreover, feature models (FMs) encourage defining a standard vocabulary for a domain language and are ideal abstractions which customers, experts, and developers can easily understand. FMs hierarchically structure domain concepts into multiple levels of increasing detail, thus proposing a taxonomy. The upper most feature is called the *root*. Root feature is then decomposed in sub-features (children), and when a feature has no child, it is denoted as a *leaf*. (On the contrary, the root feature has no parent.) When decomposing a feature into sub-features, the sub-features may be optional or mandatory or may form alternative (XOR), or (OR), or and (AND) groups. We can also denote the fact that a feature requires another one or excludes it. A particular product is formed according to the *valid* selection of features in the feature model. Such a valid selection of features is called a *configuration* of the feature model. The process of actually building the corresponding software on the basis of a configuration is called *product derivation* (Ziadi and Jézéquel 2006).

Figure 1 depicts the feature model for a smartphone-based SPL based on the Google Android operating system. This example will be used as a running example throughout the paper to illustrate and compare our two approaches for pairwise testing. The FM was added to the FM repository on the SPL Online Tool website (Mendonca et al. 2009) in order to provide it to the community.

The features **Basic Functions, Messages, Voice Call**, and **SMS** are mandatory and part of every product derived from the cell phone SPL. The feature **MMS** is optional for product instantiation. **Communication** and its subfeatures: **WLAN, Bluetooth**, and **UMTS** are optional as well. The feature **Extras** is mandatory and the underlying or-group demands that at least one element of the or-group (**MP3** or **Camera**) has to be selected. It is also possible to select both **MP3** and **Camera** within the same product. Either the **3MP** (3 megapixel) or the **8MP** (8 megapixel) has to be chosen if **Camera** is included. As this example illustrates, there are certain rules to fulfill in order to correctly select features for a given configuration of the feature model; (1) The root feature has to be in the selection, (2) The selection should evaluate to true for all operators referencing them, (3) All constraints (require and exclude) must be satisfied, and (4) For any feature that is not the root, its

[1] Defined by Pamela Zave as “An increment in functionality”. See http://www.research.att.com/pamela/faq.html and Batory et al. 2006).

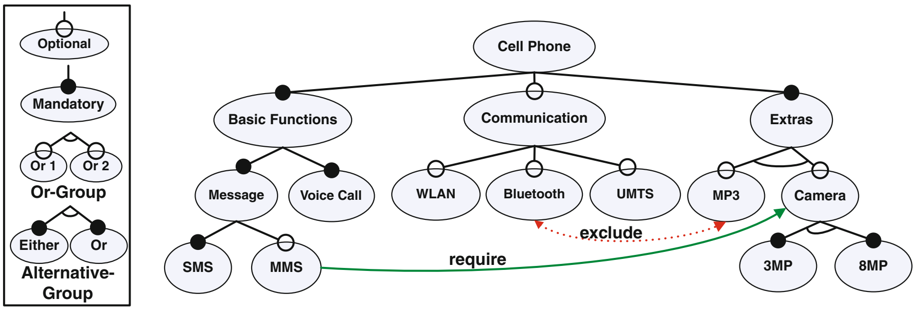

Fig. 1 Feature model of our AndroidSPL running example

parent(s) have to be in the selection. Considering this, 61 valid products can be derived on the basis of this FM.

2.2 Feature modeling and SPL testing

Due to its intuitiveness and conciseness, feature modeling has become a *de facto* standard to represent and analyze SPL. Indeed, feature models have to be considered for integration with the concrete syntax of current attempts to standardize a common variability modeling language at the object management group (OMG)[2] However to be a suitable for automated SPL testing (and verification), feature models have to satisfy two requirements: (1) be precise enough so that automatic extraction of test configurations can be performed against well-defined criteria and (2) be able to relate "concrete" assets of the SPL. Regarding this last requirement, it is important to note that the notion of "feature" can have many different meanings depending on the context (Classen et al. 2008). To preserve concision, a feature should abstract the details of its realization while efficiently associating them to allow product construction. Providing such solutions (Czarnecki and Antkiewicz 2005; Perrouin et al. 2008) are out of the scope of this paper as we focus on the problem of generating abstract test cases and not executing them. Regarding the first requirement, feature models have been equipped with formal semantics (Schobbens et al. 2006, 2007; Czarnecki et al. 2005; Batory 2005; Czarnecki and Wasowski 2007) and automated analysis (Benavides et al. 2010), techniques, and tools. There are three main benefits of formal semantics for SPL testing:

– *Notation-independent toolsets*. Since their original proposal 20 years ago, a plethora of different notations (Czarnecki et al. 2005; Griss et al. 1998; Kang et al. 1998) to name a few) have been developed. Indeed, feature models can be considered as a product line of notations sharing commonalities and exposing syntactical and semantic differences which were not always explicitly motivated. In such a context, it is not obvious for modelers to choose a specific notation on objective grounds. Furthermore, similar tool support performing analysis and derivation has to be developed for each notation. Based on a formal evaluation framework to evaluate the expressiveness of feature models, we were able to define a generic metamodel (Perrouin et al. 2008) for feature modeling, independent of any concrete syntax, and able to capture various feature modeling approaches. This metamodel is used to characterize the inputs of (Perrouin et al. 2010) testing approach and broaden its applicability.

[2] See document ad/09-12-03 on the OMG website for the full request for proposals.

- *Test configuration generation*. Automated test case generation requires the ability to form automatically valid configuration of the feature model. As we have seen, this implies satisfying all the constraints of the feature model. Formally, this can be seen as a constraint satisfaction problem (CSP). Formalization in terms of propositional logic (Batory 2005; Schobbens et al. 2007) helps encoding the problem in terms of inputs processable by CSP or SAT solvers. The testing approaches described in this paper make use of these solvers.
- *Test metrics and coverage*. Formalization of feature models also permits to define metrics for testing and assess coverage. For example, it is possible to compute all the possible valid configuration of a feature model as we have done for our example above. Even if we usually do not build all the possible members of a product line, this is an important metric to evaluate the efficiency of test case reduction. Another interesting figure is the number of core or common features (Mendonça et al. 2009). This helps to characterize the distribution of feature in the generated test configurations. This contribution focuses on feature model–based metrics and coverage criteria and does not take into account metrics and coverage addressing code or models for test or implementation purposes linked to the features of the feature model.

Therefore, we can use feature models as a relevant artifact to generate test configuration suites for SPLs. We introduce some vocabulary to establish a mapping between feature modeling concepts and testing ones.

2.2.1 SPL test case

A *SPL test case* is one valid product of the product line. Therefore, a test case is formed by a valid configuration of the feature model and its appropriate derivation. Once this test case is generated from a feature diagram, its behavior has to be tested. This is the goal pursued by the MoSo-PoLiTe (Oster et al. 2010) approach. However, in this article, we focus only on the first step, obtaining a set of abstract test cases with respect to a given criteria.

2.2.2 SPL test configuration

A *SPL test configuration* is one *valid* configuration of a feature model. This configuration is then used to form a test case. In the following, we will simply refer to SPL test configuration as “test configuration”.

2.2.3 SPL test configuration suite

A *SPL test configuration suite* is a set of SPL test configurations. We will refer this term to as “test suite”.

2.2.4 Valid/Invalid t-Tuple

A *t*-Tuple (where *t* is a natural integer giving the number of features present in the *t*-Tuple[3]) of features is said to be *valid* (respectively *invalid*), if it is possible (respectively impossible)

[3] In general, we will use the term “tuple” to mention a *t*-Tuple when *t* does not matter. In the special case of pairwise, i.e., when $t = 2$, we denote a 2-tuple by the term “pair”.

to derive a product that contains the pair (*t*-Tuple) while satisfying the feature model's constraints.

2.2.5 SPL test adequacy criterion

SPL variability represented in feature models can induce billions of possibilities, making any attempt of exhaustive testing unfeasible. Thus, to determine whether a test suite is able to cover all the SPL configurations represented by the feature model, we need to express test adequacy conditions that will allow reducing the number of test configurations to handle. In this paper, we use the combinatorial interaction testing techniques that were successfully applied to test software where multiple combinations are possible such as medical systems (Kuhn et al. 2004) or web browsers on multiple platforms (Kuhn et al. 2008). In particular, we consider the "t-wise" (Kuhn et al. 2004; Cohen et al. 2006) adequacy criterion (all-*t*-Tuples) where each valid *t*-Tuple of features is required to appear in at least one test case.

2.2.6 Test generation

In our context of SPL testing, test generation consists of analyzing a feature model in order to generate a test configuration suite that satisfies pairwise coverage of features.

Pairwise (and more generally t-wise) is a set of constraints over a range of variables [mathematically defined as *covering arrays* (Phadke 1995)]. Thus, it is possible to use SAT-solving technology (Torlak and Jackson 2007; Mahajan and Fu 2004; Niklas Een and Niklas Sorensson 2005) to compute such arrays. In our case, variables are the features of a given feature model. As we have seen, feature models can be formalized in terms of propositional logic which enable to see the problem "*t*-wise generation for feature models" as constraint satisfaction problem (CSP). Another possibility besides SAT-solving is to apply another well-known CSP solver: forward checking (Haralick and Elliott 1980).

Extensions of original CIT techniques have been proposed to handle constraints. Calvagna and Gargantini (2008) generate pairwise test sets on abstract state machines and propositional formulas representing constraints over the variables. A satisfiability modulo theory (SMT) solver is employed to verify consistency of the test configuration to include in the suite. This approach is very close to one of strategies developed in Perrouin et al. (2010) though the models and technologies employed differ. Cohen et al. (2007) examine the need for mixing pairwise algorithms with SAT solvers to handle constraints and present possible extensions of AETG in this respect. Bryce et al. (2006) distinguish different kinds of constraints and assign priorities to pairs. However, this last method is not directly applicable to feature models since "hard constraints" (constraints that prevents unfeasible combination of pairs to occur in a test configuration) are not covered by the approach.

3 Two approaches for *t*-wise testing

In this section, we present the toolsets developed by the authors to address pairwise (and more generally *t*-wise) testing of software product lines. The first one has been developed by Perrouin et al. (2010) and called "alloy-based toolset/approach" in the reminder of this paper. The second one has been developed by Oster et al. (2010) and called "dedicated CSP-based toolset/approach".

3.1 Alloy-based approach (Perrouin et al. 2010)

In the following, we describe the automatic generation of test products from a feature diagram that satisfy the *t*-wise SPL test adequacy criteria. Our tool support has been designed to support any value of *t*. The toolset has been implemented mostly in JAVA (approximately 2.3 KLOC) for *t*-wise generation and metrics computation and Kermeta (Muller et al. 2005) for transforming feature diagrams into alloy specifications. The methodology consists of five key steps shown in Fig. 2.

The generation is based on Alloy as the underlying formalism to formally capture all dependencies between features in a feature diagram as well as the interactions that should be covered by the test configurations. Alloy is a formalism dedicated to lightweight formal analysis (Jackson 2006). Alloy provides a set of concepts allowing to specify elements and constraints between them. The first construct is *Signature* (`sig`). A signature defines a set of elements and possibly the relationships with other elements. Signatures are similar to type declarations in an object-oriented language. Facts (`fact`) are axioms that specify constraints about elements and relationships. These axioms must always hold, and they are close to the concept of invariants in other specification languages. Predicates, (`pred`), as opposed to facts, define constraints which can evaluate to true or false. With these constructs, it is possible to build various kinds of alloy models and to ask alloy whether it is possible to find instances that satisfy all constraints and evaluate one predicate to true. The *scope* is an integer bound on the maximum number of instances for each signature (Jackson 2006). This allows the limitation of the search space in which alloy looks for a solutions, and this is a way to finely tune how alloy builds instances satisfying a model.

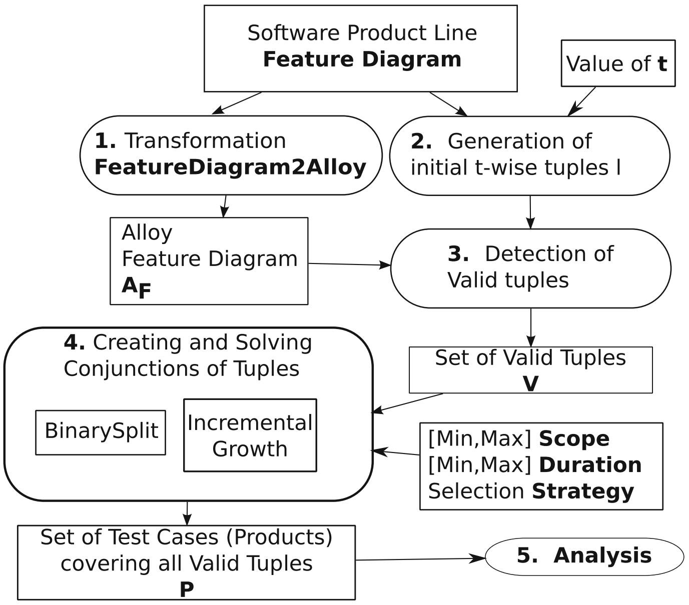

Fig. 2 Product Line Test Generation Methodology

3.1.1 Step 1: transforming feature diagrams to alloy

In order to generate valid test configurations directly from a feature diagram, we need to transform the diagram in a model that captures constraints between features. The *FeatureDiagram2Alloy* transformation automatically generates an alloy model A_F from any feature model F expressed in our generic feature diagram formalism (Perrouin et al. 2008) (Listing 1).

The A_F model captures all features as *alloy signatures* and a set of *alloy signatures* that capture all constraints and relationships between features. This model also declares two signatures that are specific to test generation: *configuration* that corresponds to a test configuration and that encapsulates a set of features (Listing 2); *ProductConfiguration* (Listing 3) that will encapsulate a set of test cases.

Example In the cell phone SPL, shown in Fig. 1, we have 15 features $f_1, f_2, \ldots, f_{15}$. The transformation *FeatureDiagram2Alloy* generates 15 signatures to represent these features shown in Listing 1. Signatures representing mandatory features are preceded by the alloy keyword one stating that their valuation is always one. Signatures representing variable features are preceded by the alloy keyword lone (meaning zero or one).

The *FeatureDiagram2Alloy* transformation generates *Alloy facts* in A_F.

```
sig SMS{}
sig MMS{}
sig VoiceCall{}
sig WLAN{}
sig Bluetooth{}
sig UMTS{}
sig MP3{}
sig ThreeMP{}
sig EightMP{}
sig BasicFunctions{}
sig Message{}
sig Communication{}
sig Extras{}
sig Camera{}
sig CellPhone{}
```

Listing 1 Generated signatures for features for the cell phone SPL

```
sig Configuration
{
f1 : one SMS,   //mandatory
f2 : lone MMS,  // variable
...
f13 : one Extras,
f14 : lone Camera,
f15 : one CellPhone, //mandatory
}
```

Listing 2 Generated signature for configuration of features for the cell phone SPL

```
one sig ProductConfigurations
{
  configurations : set Configuration
}
```

Listing 3 Generated signature for set of configurations

3.1.1.1 Example In the Listing 4, we present two generated Alloy facts corresponding to the XOR and AND operators. These facts must be true for all configurations. The first constraint states that if *Camera* (f_{14}) is selected, then the sum resulting from the selection of children features (*ThreeMP* and *EightMP*, respectively, f_8 and f_9) cannot be greater than 1.

The *FeatureDiagram2Alloy* transformation has been implemented as a model transformation in the Kermeta metamodeling environment (Muller et al. 2005). Since our feature diagram formalism is generic (Perrouin et al. 2008) various kinds of feature diagrams can be automatically transformed, e.g., FODA FMs (Kang et al. 1990) or the orthogonal variability model (OVM) proposed by Pohl et al. (2005).

3.1.2 Step 2: generation of tuples

In Step 2, we automatically compute the set I of all possible tuples from feature diagram AF and the number t. The tuples enumerate all t-wise interactions between all selections of features in AF.

Example The 3-tuple $t = <\#f_{15} = 1, \#f_2 = 0, \#f_{13} = 1>$ for the value $t = 3$ contains 3 features and their valuations. In the tuple, we state that the test suite must contain at least one test configuration that has features f_{15} (*CellPhone*), not f_2 (*MMS*) but f_{13} (*Extras*).

The initial set of tuples I is the set of tuples that cover all combinations of t features taken at a time. For example, if there are N features, then the size of I is ${}_{2N}C_T$ minus all tuples with repetitions of the same feature (e.g. $<\#f_{15} = 1, \#f_{15} = 1>$). In the case of the cell phone SPL and considering pairwise (or 2-wise), there are 435 possible combinations of features. As there are 15 repetitions of the same feature, we consider only 420 tuples in our set I.

Each tuple t in I also has an alloy predicate representation. An alloy predicate representation of a tuple t is *t.predicate*.

Example The tuple $t = <\#f_{15} = 1, \#f_2 = 0, \#f_{13} = 1>$ is shown in Listing 5.

```
// Camera XOR Operator =>  ThreeMP XOR  EightMP XOR
fact Invariant_Operator_12
{
  all c:Configuration | #c.f14==1 implies ( ( #c.f8 +  #c.f9 )==1)
}
// CellPhone And Operator =>  BasicFunctions AND  Extras AND
fact Invariant_Operator_13
{
  all c:Configuration | #c.f15==1 implies ( #c.f10=1 and  #c.f13=1)
}
```

Listing 4 Generated facts for XOR and AND operators

```
pred t
{
  some c:  Configuration | #c.f15=0 and #c.f2=0 and #c.f13=1
}
```

Listing 5 Example tuple predicate

3.1.3 Step 3: detection of valid tuples

In this third step, we use the predicates derived from each possible tuple in order to select the valid ones according to the feature model. We say that a tuple is valid if it can be present in a valid instance of the feature diagram F.

Example Consider our running example, $t = <\#f_2 = 1, \#f_{14} = 0>$ is not a valid tuple, as the feature f_2 (*MMS*) required the existence of feature f_{14} (*Camera*) and hence we neglect it. On the other hand, the 3-tuple $t = <\#f_1 = 1, \#f_2 = 0, \#f_4 = 1>$ is valid since all feature selections hold true for F. We determine the validity of each such tuple t by solving $A_F \cup$ *t.predicate* for a scope of exactly 1. This translates to solving the alloy model to obtain *exactly one product* for which the tuple t holds true. For the cell phone case study, we have 420 tuples for pairwise ($t = 2$) interactions in the initial set I. We select 257 valid tuples in the set V.

3.1.4 Step 4: creating and solving conjunctions of multiple tuples

Once we have a set of valid tuples, we can start generating a test suite according to the t-wise SPL adequacy criteria. Intuitively, this consists in combining all valid tuples from V with respect to A_F in order to generate test products that cover all t-wise interactions.

Example For pairwise testing in the case of cell phone SPL, this amounts to solving a conjunction of 257 tuple predicates $t_1.predicate \cap t_2.predicate \cap \ldots \cap t_{257}.predicate$ for a certain scope.

Though the number of tuples to solve in this example is reasonable, it changes rapidly with the value of t. For instance, computing 3-wise on the same example, would require solving 1639 tuples instead of 257. If the number of tuples can be evaluated quickly, the difficulty of solving them over a given alloy model is impossible to guess *á priori*. As a result, depending on the number of tuples and the “solving complexity” (driven by the number of operators and cross-tree constraints) of the feature model, solving all these tuples at once may fail. A pragmatic approach is to divide the solving phase in sets that the solver can process more easily. Hence, we derived two “divide-and-compose” strategies to breakdown the problem of solving a conjunction of tuples to smaller subsets of conjunction of tuples. The strategies we present are *binary split* and *incremental growth*. Each strategy is parameterized by intervals of values defining the scope of research for each (sub)-conjunction of tuples, the duration in which alloy is authorized to solve the conjunction as well as a strategy defining how features are picked in a tuple. We describe these strategies in more detail below. The combination of solutions is a test suite *TS* that covers all tuples.

3.1.4.1 Binary split The *binary split* strategy shown in Algorithm 1 is based on splitting the set of all valid tuples V into subsets (halves) until all subsets of tuples are solvable. We first order the set of valid tuples based on the strategy *Str*. The strategy can be *random* or based on *distance* measure. In this paper, we consider a random ordering. The *Pool* is set of sets of tuples. Initially, *Pool* contains the entire set of valid tuples V. If each set of tuples $Pool[i]$, $0 \leq i \leq Pool.size$ in *Pool* is not solvable in the given range of scopes *mnSc* and *mxSc* or within the maximum duration *mxDur* then *result* is *False* for $Pool[i]$. A single value of $result = False$ renders $AllResult = False$. In such a case, we select the *largest set* in $Pool[i]$ and split it into halves $\{H1\}$ and $\{H2\}$. We insert the halves $\{H1\}$ and $\{H2\}$ into $Pool[i]$. The process is repeated until all sets of tuples in *Pool* can be solved given the time limits and *AllResult* = *True*. In the worst case, halves are made with one tuple, by definition solvable.

3.1.4.2 Incremental growth The *incremental growth* strategy is shown in Algorithm 2. In the algorithm, we incrementally build a set of tuples in the conjunction *CT* and add it to the *Pool*. The *select* function based on a strategy *Str* selects a tuple in *V* and inserts it into *CT*.

Algorithm 1 binSplit(A_F, *V*, *mnSc*, *mxSc*, *mxDur*, *Str*)

```
$AllResult \leftarrow True$
$V \leftarrow order(V, Str)$
$Pool \leftarrow \{\{V\}\}$
repeat
    $result \leftarrow False$
    $i \leftarrow 0$
    repeat
        $\{result, Pool[i].solution\}$
        $\leftarrow solve(A_F, Pool[i], mnSc, mxSc, mxDur)$
        $i \leftarrow i + 1$
        $AllResult \leftarrow AllResult \wedge result$
    until $i = \ = Pool.size$
    if $AllResult = \ = False$ then
        $\{L\} = max(Pool)$
        $\{\ \{\ H1\ \}, \{\ H2\ \}\ \} = split(\{\ L\ \}, 2)$
        $Pool.add(\{\ H1\ \})$
        $Pool.add(\{\ H2\ \})$
until $AllResult = false$
Return $Pool$
```

Algorithm 2 incGrow(A_F, *V*, *mnScp*, *mxScp*, *mxDur*, *Str*)

```
$Pool \leftarrow \{\}$
repeat
    $CT \leftarrow \{\}$
    repeat
        $tuple \leftarrow V.select(Str)$
        $CT.add(tuple)$
        $\{\ result, CT.solution\ \}$
        $\leftarrow solve(A_F, CT, mnSc, mxSc, mxDur)$
        if $result = \ = False$ then
            $CT.remove(tuple)$
            $V.add(tuple)$
    until $result = \ = False$
    $Pool.add(CT)$
until $V.isEmpty$
Return $Pool$
```

The strategy *Str* can be *random* or based on a *distance* measure between tuples. In this paper, we consider only a random strategy for selection. We select and remove a tuple from *V* and add it to *CT* until the conjunction cannot be solved anymore, i.e. $result = False$. We remove the last tuple and put it back into *V*. We include *CT* into *Pool*. In every iteration, we initialize a new conjunction of tuples until we obtain sets of tuples in *Pool* that contain all tuples initially in *V* or when *V* is empty.

3.1.5 Step 5: analysis

Once the solutions have been generated, we can perform some analyses to assess the quality of the generated test suites. In Perrouin et al. (2010), we have defined a set of metrics to compare our two strategies. We will reuse and extend some of these metrics with the aim of comparing the two approaches for *t*-wise generation dealt with in this paper.

3.2 Dedicated CSP-based approach (Oster et al. 2010)

The second approach applies graph transformation, combinatorial testing, and forward checking for the test suite generation. The goal is to apply pairwise algorithms similar to AETG (Cohen et al. 1997) and IPO (Lei and Tai 1998) to feature models.

To apply combinatorial testing to feature models, we either have to adapt an existing combinatorial algorithm so that it can handle the hierarchical structure, the different node notations, and constraints of the feature model or have to change the structure of the feature model so that it can be processed using existing pairwise algorithms.

We combine both ideas: First, the structure of the feature model is changed so that it is processable by combinatorial algorithms. This flattening translates a feature model into a binary constraint solving problem (CSP), extracting parameters and parameter values. The second step realizes pairwise combination by integrating a pairwise algorithm and standard constraint solving techniques such as forward checking. A subset extraction algorithm generates all valid pairwise combinations of features regarding cross-tree dependencies, the hierarchical structure, and the different feature notations in the feature model.

3.2.1 CSP Translation

A so-called CSP translation algorithm reduces the depth of the feature model to extract parameters with corresponding values. This translation can easily be adapted to be applied to different kinds of feature models or to an OVM (Pohl et al. 2005).

The algorithm consists of two steps:

1. Every feature with its associated notation and dependencies is iteratively pulled up until it is placed directly beneath the root node. Every feature then serves as a parameter.
2. The algorithm assigns every parameter its correspondent parameter value.

Several model transformation rules control the CSP translation; they are iteratively applied to a subtree of a feature model. A subtree always consists of three levels: the grandparent node, the parent node, and the child node. Different rules are required for the translation process depending on the notations of the involved features. We currently support four different node notations: mandatory, optional, or, and alternative. For every possible combination of parent and child notation, a separate transformation rule is required: $4 \times 4 = 16$ rules are needed. As examples, we depict three rules to describe our flattening approach. For a complete description of all the rules, refer to (MoSo-PoLiTe 2011).

Figure 3 depicts three transformation rules: (1) pulling up a mandatory and (2) an *optional* node beneath a *mandatory*-parent node and (3) pulling up an *alternative*-group of child nodes with a parent node placed in an *or*-group.

Figure 3a shows the transformation rules 1 and 2. A mandatory child node is always included within its parent node. Thus, *SMS* and *Message* are combined to be one feature, because it is not possible to select a configuration without *SMS* when *Message* is selected. An optional child node (*MMS*) stays optional and is pulled up besides the parent node.

Figure 3b shows rule number 3. The parent *or*-group stays unchanged, and the *alternative*-group is pulled up beside the parent. Because the features **3MP** and **8MP** can only be chosen if **Camera** is selected, we have to add require dependencies. Furthermore, an additional feature is added into the alternative group: the ¬**Camera** feature which is required for the situation that **Camera** is not selected. Without adding this feature, either **3MP** or **8MP** is always selected and, therefore, **Camera** is always required. Selecting ¬**Camera**, the feature **Camera** is excluded, and we preserve the semantic equivalence between both FMs.

After the first step of the translation algorithm, all features are placed directly beneath the root node serving as parameters. Figure 4 depicts the flat feature model.

In the next step, we extract the corresponding values. Again, different rules are applied to extract the values of the features.

– **optional:** An optional feature is changed to a mandatory feature with two child nodes. The optional feature **MMS** turns into a mandatory node with an alternative-child group containing a feature **MMS** and ¬**MMS**. For product instantiation, the feature **MMS** is selected and one element of the alternative group has to be chosen as well. Therefore, either the feature **MMS** or the feature ¬**MMS** is selected.
– **mandatory:** Mandatory nodes stay mandatory and obtain an additional child node with the same notation and name. (e.g. **Extras**)

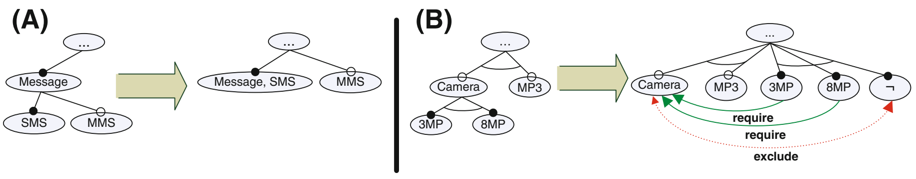

Fig. 3 Transformation rule pulling up an alternative-child with an or-parent

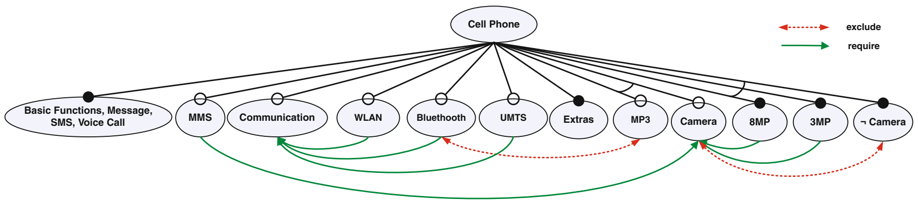

Fig. 4 Flat feature model of our case study

- **or:** Extracting the parameter values of an **or**-group is the most complex rule. Each feature of the **or**-group is handled like an optional feature. To ensure that a least one element of the **or**-group has to be chosen within a product, the values for not including the features within a product exclude each other.
- **alternative:** An alternative group stays unchanged, but we add a single placeholder feature in-between the alternative group and the root node representing the parameter (**ALT_F**).

Figure 5 shows the flat feature model of our running example including feature values. This flat feature model exhibits the following variability: $1^1 2^2 2^3 2^4 2^5 2^6 1^7 2^8 2^9 3^{10}$.

A valid pair is a combination of two features not violating cross-tree dependencies, the hierarchical structure, and the different feature notations in the FM (cf. lines 2–3). Then, the algorithm incrementally combines those pairs of features to create valid test configurations (cf. lines 5–12). The algorithm starts with the first pair and iteratively adds pairs of the remaining parameters (cf. line 8). For each step, forward checking (Haralick and Elliott 1980) is applied to determine whether the selected pair can be combined with remaining pairs of parameters to create a valid test configuration (cf. line 9). If a certain pair results in such a deadlock, another pair is selected instead (cf. line 11). The algorithm continues until all pairwise combinations are covered by at least one configuration and will return the list of selected configurations.

Compared to AETG and IPO, we adopted the following ideas for our algorithm:

- Building product by product as in AETG.
- Using a list of pairs that need to be covered as in IPO.
- Using a weighting/priority function to decide which value to select within a certain configuration similar to AETG. This function calculates the priority of a certain value according to its occurrence within the list of pairs that need to be covered. The value which has the most required combinations obtains the highest priority.

We applied our algorithm to the presented running example. The algorithm identified 8 test configurations which are listed in Fig. 6 covering all pairwise interactions of features.

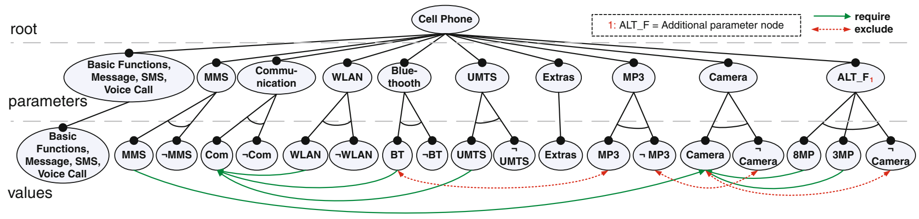

Fig. 5 Flat feature model with parameters and values

P1	CellPhone	B, M, S, V	Extras	Comm.	¬MMS	¬UMTS	¬WLAN	¬BT	Camera	8MP	¬3MP	MP3
P2	CellPhone	B, M, S, V	Extras	¬Comm.	MMS	¬UMTS	¬WLAN	¬BT	Camera	¬8MP	3MP	MP3
P3	CellPhone	B, M, S, V	Extras	Comm.	¬MMS	UMTS	WLAN	¬BT	¬Camera	¬8MP	¬3MP	MP3
P4	CellPhone	B, M, S, V	Extras	Comm.	MMS	UMTS	WLAN	BT	Camera	8MP	¬3MP	¬MP3
P5	CellPhone	B, M, S, V	Extras	¬Comm.	¬MMS	¬UMTS	¬WLAN	¬BT	Camera	8MP	¬3MP	¬MP3
P6	CellPhone	B, M, S, V	Extras	Comm.	¬MMS	¬UMTS	WLAN	BT	Camera	¬8MP	3MP	¬MP3
P7	CellPhone	B, M, S, V	Extras	¬Comm.	¬MMS	¬UMTS	¬WLAN	¬BT	¬Camera	¬8MP	¬3MP	MP3
P8	CellPhone	B, M, S, V	Extras	Comm.	MMS	UMTS	¬WLAN	BT	Camera	¬8MP	3MP	¬MP3

Fig. 6 The resulting test suite covering all valid pairs of features of the running example

Furthermore, the subset extractor can handle seeds to be provided by the user. To realize this functionality, the pairs of these seeds are extracted and stored. When generating the set of pairs to cover, these pairs are marked as already covered and the algorithm uses the remaining pairs.

4 Comparison framework

In this section, we introduce some measures to evaluate the quality of our test generations approaches. These measures are inspired from earlier work (Perrouin et al. 2010).

4.1 Performance

Concerning performance, time required for the toolset to perform the computation of a solution is the most obvious metrics. We therefore use execution time to measure the performance of the compared toolsets and give values for examples of the SPLOT repository in Sect. 5.

4.2 Test suite size

One of the simplest metrics to characterize generation is the number of test configurations generated by the *t*-wise toolset:

Definition 1 *Test Suite Size*. The number of test configurations composing it gives the size of a test suite.

As discussed in Perrouin et al. (2010), there is a trade-off to find between two antagonist goals, *optimality* and *coverage*. Optimality requires the minimum number of test configurations meeting the *t*-wise criteria. In the cell phone SPL, this can be obtained with only 8 test configurations over the 61 possible ones induced by the feature model. Thus, having more test configurations than absolutely necessary implies a greater testing effort but also to the benefit of a greater coverage. This metric is also an indirect indicator of the degree of "repetition" a given *t*-wise strategy may produce by splitting tuple conjunction and composing results. This "repetition" issue is more finely captured in the following metrics.

4.3 *t*-Tuple occurrence and frequency

The *t*-wise criterion states that every valid *t*-Tuple must be present in a least one test configuration of the test suite (exactly one being the optimum in this respect). However, this optimum is barely achieved. There are three main reasons for this:

– *Mandatory and Common Features*. The occurrence of a given tuple is strongly influenced by the nature of the features composing it. A *common feature* (or *core feature*) (Benavides et al. 2010; Mendonça et al. 2009) has to be present in all valid configurations of the feature model. This comprises mandatory features but also their dependencies (parents, require/exclude constraints…). Therefore, a non-mandatory feature may be always included to satisfy complex combinations of constraints and operators. Such features are therefore considered as "false-optional" (Benavides et al. 2010). As noticed by Mendonça et al. (2009), this can be an undesirable design flaw.

Therefore, tuples that are composed only of common features will appear in every test configuration of the suite; their occurrence will correspond to the size of the test suite.
- *Constraints*. Cross-tree constraints (such as require/exclude) by enforcing relationships between features are likely to increase the number of times a given tuple appears in the test configurations.
- *Generation Algorithm/Strategy*. The generation algorithm or the "divide-and-compose" strategy used to incorporate tuples in test configurations may deterministically or randomly deviates from the optimum.

These reasons motivated our will to measure by "how much" the *t*-wise criteria was over met. We define two related metrics.

Definition 2 *t-Tuple Occurrence*. *t*-Tuple occurrence is the number of times a given *t*-Tuple appears in a test suite.

Example The pair $<\neg 3MP, MP3>$ appearing three times in Fig. 6 representing a pairwise compliant test suite for the cell phone feature model has a tuple occurrence of 3. The mandatory pair <*CellPhone*, *BasicFunctions* > has tuple occurrence of 8.

We initially used this metric to assess the optimality of *t*-wise generation by measuring the number of repetitions of a *t*-Tuple in a suite. Yet, as the number of generated products may vary depending on the algorithm or strategy used, the raw occurrence is difficult to comment without information on the test suite size. Furthermore, for same reason, *t*-Tuples composed of only common features, may be harder to detect. Hence, we take into account the number of test configurations in this related metric:

Definition 3 *t-Tuple Frequency*. *t*-Tuple frequency is the ratio between the *t*-Tuple occurrence and the size of the test suite.

Example The pair $<\neg 3MP, MP3>$ has a tuple frequency of 0.375, while the mandatory pair <*CellPhone*, *BasicFunctions* > has a tuple frequency of 1.

As a result, *t*-Tuple frequency is a value in the [0,1] interval. A value of 0 for a given *t*-Tuple means that there is no occurrence of this tuple in test suite. This cannot normally happen: we are dealing only with valid *t*-Tuples needing to appear at least once to meet the *t*-wise criteria. This can be used as a sanity check to exhibit bugs in the *t*-wise generation algorithm. A value of 1 means that the *t*-Tuple appears in all generated test configurations, implying that the *t*-Tuple is comprised of common features. This also can be used as a conformity check: If one or more *t*-Tuples consisting of common features has a frequency less than 1, then test configuration generation is invalid with respect to the feature model.

4.4 Test configuration similarity

The objective of this metric is to answer the question: "How similar are my generated test configurations ?". In fact, *t*-wise generation techniques rearrange *t*-Tuples in test configurations in different ways (as we have seen this can be done by splitting the *t*-Tuple subset or by incrementally constructing them). This results in some test configurations that cover "almost the same" product or very different ones. Furthermore, "divide-and-compose" strategies allow by construction that identical test configurations are generated. These points form the main motivation of proposing a similarity metric (Cartaxo et al. 2011). A few similarity functions have been proposed in the literature in the context of model-based testing (Cartaxo et al. 2011; Hemmati and Briand 2010). However, to our knowledge, none has been proposed to compare test configuration generated from a feature model. Ours is based on the Jaccard index (Tan et al. 2006), which is devoted to the comparison of two sample sets:

$$Jac(A,B) = \frac{\|A \cap B\|}{\|A \cup B\|}$$

Here, the sample sets are the sets of variants features (all features that are not common (Benavides et al. 2010)) of the SPL. Thus, variant features represent the possible decisions (to select a feature or not) one can make on the feature model.

Definition 4 *Test Configuration Similarity*. Test configuration similarity is defined between two test configurations as the Jaccard index of the number of identical variant features (i.e. identical decisions) over the possible number of variants features.

Hence,

$$Sim(tc_i, tc_j) = \frac{\|Tci_v \cap Tcj_v\|}{\|Tci_v \cup Tcj_v\|}$$

where tc_i, tc_j are test configurations, Tci_v , Tcj_v sets of their variants features.

Example The SPL test configurations $P1_1$ and P_2 illustrated Fig. 6 have 5 variants features in common out of 9, this $Sim(P_1,P_2) = 0.55$.

4.5 Test suite similarity

After having introduced the notion test configuration similarity, we generalize it to define *test suite similarity*:

Definition 5 *Test Suite Similarity*. Test suite similarity is the arithmetical mean of test configuration similarities computed over the Cartesian product of the test suite by itself.

More precisely, for any test suite *ts*, we have:

$$Sim_{ts} = \frac{\sum_{i=1}^{t} \sum_{j=1}^{t} Sim(tc_i, tc_j)}{t^2}$$

where tc_i, tc_j are test configurations, $Sim(tc_i, tc_j)$ their similarity and $t = | ts |$ i.e., the number of test configurations present in the test suite.

5 Experimentation

In this section, we apply the measures defined in the previous section on the toolsets developed by the authors. In particular, we compare the alloy-based approach (with its two strategies: BinarySplit and IncrementalGrowth) with the dedicated CSP-based approach for pairwise testing on examples present in the SPLOT repository for feature models.

5.1 Case studies validation

As we have seen, the alloy-based approach and the dedicated CSP-based approach have different inputs and model-driven transformation chains. Therefore, there is a risk that the source models (created by the designer either in PureVariants for the CSP approach or using an EMF compliant[4] tool for the *t*-wise approach) are not semantically equivalent, and therefore, we do not generate comparable results for pairwise. To eliminate this risk, we cross-checked our feature model implementations. Indeed, we ensured that all invalid pairs generated by dedicated CSP-based approach are also invalid when applied on the generated alloy model. We used pairs generated by the alloy-based approach (Perrouin et al. 2010) with the generated test configurations from the dedicated CSP-based approach (Oster et al. 2010). We also inspected manually generated test configurations for the examples considered.

5.2 Experiment design

As discussed in Perrouin et al. (2010) and as for any solution based on Alloy, the choice of the scope is a very important parameter to set. As we have shown, there is an optimal value for the scope that minimizes the number of generated test configurations and similarity. This cannot be determined in advance and depends of the case study. The toolset automatically generates sets of test suites in order to study the effects of random ordering of tuples (Perrouin et al. 2010). When it was possible, we therefore generated 10 test suites for each strategy and we report on the measures performed using "boxplot and whiskers" to illustrate the results distribution. As the Dedicated CSP-based approach does not have this kind of setting, varying the scope cannot be taken into account in the comparison. One big difference between the alloy-based approach (with its 2 strategies) and dedicated CSP-based approach (incremental pairwise) is that the latter is deterministic; it generates the same set of test configurations. As a consequence, one test suite is sufficient to compare the generation behavior with other strategies.

5.3 2-wise testing

5.3.1 Execution times

We report execution times for examples taken in the SPLOT online repository in Table 1 above. Figures such as >32400000 indicate that we stopped the alloy-based framework from running after more than nine hours of computation, either having partial results (may not fully respect the pairwise criterion) for the incremental growth strategy or with no result at all for the binary split strategy.

An obvious observation one can make from this table is the CSP-dedicated approach is at least 1,000 times faster than any of the strategies of the alloy-based solution. Furthermore, the CSP-dedicated approach execution times grows gently with the feature model complexity while the alloy-based strategies execution times follows a steeper increasing curve. It is not that surprising as the strategies decompose the problems in hundreds or thousands of solving steps. Yet, what matters to the tester is that the overall computation time may be judged unreasonable for large feature models.

As initially stated (Perrouin et al. 2010), we confirm here that the binary split strategy is faster than the incremental growth one. However, we observed a greater stability of the incremental growth strategy that may be used when the binary decomposition fails to give a result.

[4] EMF (Budinsky et al. 2003) is an Eclipse framework dedicated to the manipulation of models, on which we based our generic feature modeling approach (Perrouin et al. 2008).

Table 1 Execution times for pairwise generation on feature models

	CP	SH	AG	MT	ES
Features	19	35	61	88	287
Possible products	61	10,48,576	$3.3 * 10^9$	$1.65 * 10^{13}$	$2.26 * 10^{49}$
Cross-tree constraints (%)	26	0	55	0	11
CSP-dedicated (ms)	0	0	32	46	797
BinarySplit (ms)	11,812	11,457	33,954	>3,24,00,000	>3,24,00,000
IncrementalGrowth (ms)	56,494	13,72,094	1,38,47,835	>3,24,00,000	>3,24,00,000

Key: *CP* cell phone, *SH* smart home, *AG* arcade game, *MT* model transformation, *ES* electronic shopping

5.3.2 Test suite size

Table 2 below shows the number of products obtained by pairwise testing the considered feature models.

In the following, we focus on our running example (cell phone) and a larger feature model, the arcade game feature model.

5.3.2.1 Cell phone Figure 7 shows the boxplots for the two approaches. For BinarySplit, the size of the test suite varies between 12 and 20 test configurations, with an average of 15.6 and a standard deviation 2.7. Regarding IncrementalGrowth, we compute as less as 15 test configurations and as much as 18, with a mean of 15.7 with a standard deviation of 1.5. While these two strategies are comparable in the number of generated test configurations, we observe a greater stability of IncrementalGrowth with respect to BinarySplit. This confirms our initial assumption (Perrouin et al. 2010); the incremental way is more accurate in finding an extremum (whether local or global) and reproducing this extremum while BinarySplit will be more or less "lucky" while distributing halves of tuples. This trend is confirmed with the dedicated CSP-based approach; it always generate 8 test configurations for the suite which is the exact minimum for satisfying the pairwise criteria.

As we have discussed it above, BinarySplit and IncrementalGrowth can generate redundant test configurations. This fact can also be revealed by similarity computation. IncrementalGrowth can generate as many as 4 duplicates (with a minimum of 0) with a mean of 2.0 and a standard deviation of 1.15. BinarySplit can generate as many as 5.0 duplicates and as less as 0 with a mean of 2.3 and a standard deviation of 1.5. The greater diversity of BinarySplit in the generation is also confirmed here. The dedicated CSP-based approach has reached the minimum in computing the solution "all-at-once": there are no redundant test configurations in the test suite.

Table 2 Test suites sizes obtained for pairwise generation

	CP	SH	AG	MT	ES
CSP-dedicated	8	40	46	92	215
BinarySplit	12	92	514	N/A	N/A
IncrementalGrowth	15	28	74	N/A	N/A

Key: *CP* cell phone, *SH* smart home, *AG* arcade game, *MT* model transformation, *ES* electronic shopping

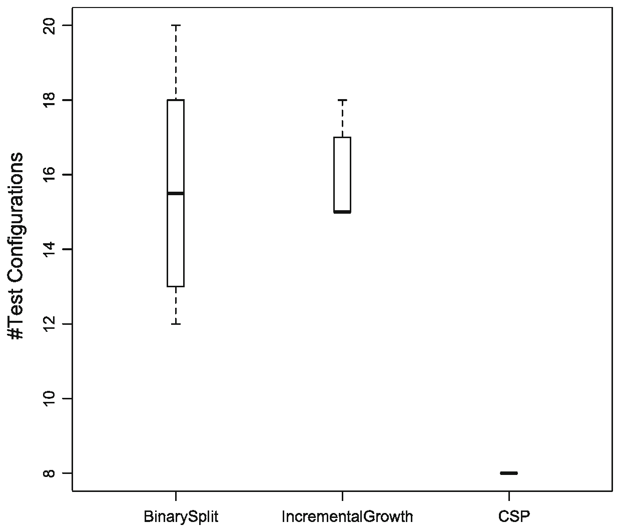

Fig. 7 Number of generated test configurations for the cell phone feature model (pairwise)

5.3.2.2 Arcade game It is interesting to see whether these tendencies are confirmed for larger examples. We therefore decided to report test suite size obtained for the arcade game feature model. However, due to important execution times (see Table 1), it was not possible to generate set of 10 solutions for the alloy-based strategies. We thus adopted a "best guess" approach in which we report one solution for each of the strategies. Since there is only one value for the size of test suite for all CSP-based and alloy-based toolsets, we report obtained results in Table 3.

A first observation is that the test suite size varies in a large extent between CSP-dedicated and alloy-based toolsets and within strategies themselves. This observation can be explained by the fact that the strategies need many more steps to compute the test suite yielding more test case configurations. Another important observation is that on this example there is no duplicate. It can be surprising as BinarySplit and IncrementalGrowth strategies produce duplicates on smaller examples and more "divide-and-compose" steps can mean more chances of deriving redundant test case configurations. In fact, the arcade game model allows a significant number of variants ($3.3*10^9$) to be derived from the model implying that the probability of twice the same test case configurations decrease with the number of possible variants.

5.3.3 t-Tuple occurrence and frequency

In the next paragraphs, we provide the computed tuple occurrences for the Cell Phone and Arcade Game feature models.

Table 3 Test suite size and duplicates for the arcade game feature model

	Test suite size	Duplicates
CSP-dedicated	46	0
BinarySplit	514	0
IncrementalGrowth	74	0

5.3.3.1 Cell phone *t*-Tuple occurrence is depicted in Fig. 8. As it can be seen, the pairwise criterion is satisfied as no *t*-Tuple appears less than once in any test suite. As there are more than two common features (6), there are necessarily 2-tuples that are composed of common features. Therefore, their occurrences correspond to the size of the generated test suite, represented in the box plot as outliers. As result of generating less test configurations, the dedicated CSP-based approach has a lower number of tuple occurrences. We also observed a remarkably stable frequency distribution. On average, a tuple is appearing in 41% of all the generated test configurations.

5.3.3.2 Arcade game Regarding *t*-Tuple occurrences for the arcade game feature model shown Fig. 9, the pairwise criterion is met as well as the minimal occurrence of a tuple is 1. However, due to the fact of generating more test case configurations, BinarySplit

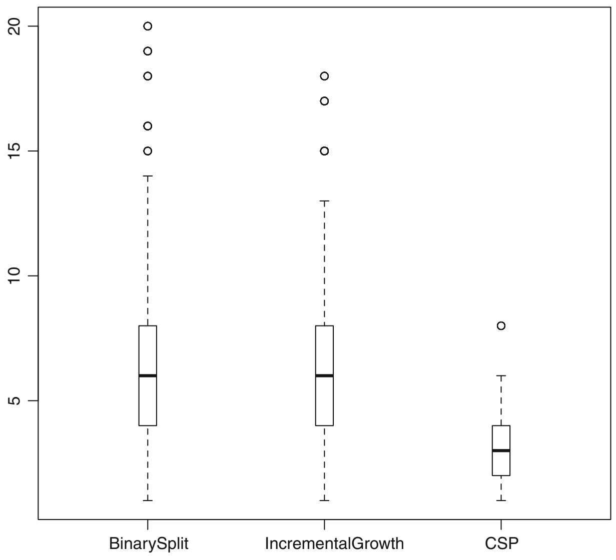

Fig. 8 Tuple occurrences for the cell phone feature model (pairwise)

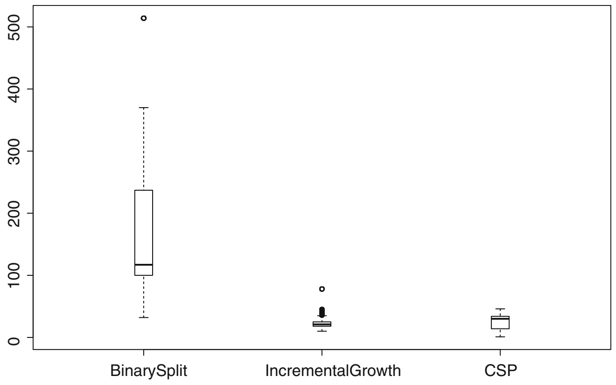

Fig. 9 Tuple occurrences for the arcade game feature model (pairwise)

and IncrementalGrowth strategies have a tendency to over meet this criterion: the minimal occurrence of a tuple is 10 for IncrementalGrowth and 32 for BinarySplit. It is important to note that the frequency of apparition of a tuple is 58% for CSP-dedicated approach and 31% both for BinarySplit and IncrementalGrowth. As opposed to the cell phone example, the difference appears more clearly. High frequencies are to be looked for to test the same couple of features in various contexts which is essential for critical ones.

5.3.4 Similarity

The last measure is similarity that we provide for our two examples.

5.3.4.1 Cell phone Similarity is plotted in Fig. 10. What is important here are the median values. They are the same for the "incrementally driven" approaches (0.44 for CSP and IncrementalGrowth) while a little bit higher for BinarySplit. Hence, within a test suite, if diversity-based testing is an objective (Hemmati et al. 2010), testers should privilege an incremental approach to pairwise testing.

5.3.4.2 Arcade game Test suite similarity is depicted Fig. 11. The CSP-dedicated approach tends to produce more similar (mean = 0.66 compared to 0.44 for IncrementalGrowth or 0.52 for BinarySplit) results on average. Yet, it has to be noted that there is a few interesting outliers in which few decisions regarding feature selection are common. The alloy-based strategies are not able to reach such extremes while they maintain a good diversity on average despite a larger set of test configurations. This is made possible by the nature of the feature model. There are 43 variant features in the model which allows many more choices to pick a given test configuration.

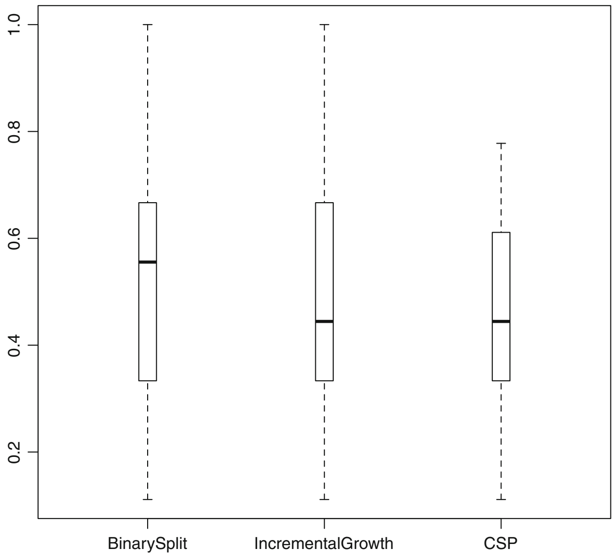

Fig. 10 Test suite similarity distribution for the cell phone feature model (pairwise)

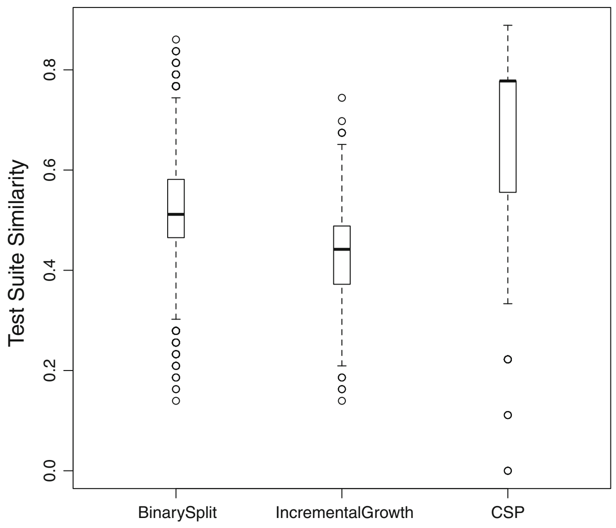

Fig. 11 Test Suite Similarity Distribution for the Arcade Game Feature Model (pairwise)

5.4 Beyond pairwise

As mentioned in Sect. 3, both approaches were meant to generate tests configurations for values of $t \geq 2$. In the following, we report on experimentations carried out for $t = 3$.

5.4.1 Execution times

Table 4 details execution times for 3-wise generation on SPLOT examples. Unsurprisingly, execution times are higher than those obtained for pairwise. While still low for CSP-dedicated, there are at least 10 times greater regarding alloy-based strategies. The tendency between IncrementalGrowth and BinarySplit is also confirmed, BinarySplit performing much faster than IncrementalGrowth.

We also observed scalability issues from the Smart Home feature model onwards. Issues encountered were linked to "out of memory" errors arising at two different steps of the computation: (i) during the solving, after several hours of computation and (ii) after the generation of valid tuples in the arcade game case. This last problem was generated by the fact that the alloy model representing the whole problem was too big (the model is several hundred of thousands lines) to be handled by the alloy API.

5.4.2 Test suite size

Table 5 below shows the number of products obtained by 3-wise testing the considered feature models.

An higher value of t also induces greater test suite sizes. The expected effects on test suite size of the "divide-and-compose" strategies appear clearly. As the problem is more difficult, the decomposition proceeds with more steps and adds more test configurations to the suite. Decomposition side effects are also revealed by the number of generated duplicated as illustrated in Table 6.

Table 4 Execution times obtained for 3-wise generation

	CP	SH	AG	MT	ES
CSP-dedicated (ms)	0	56	83	118	2,586
BinarySplit (ms)	5,84,893	>3,24,00,000	Fail	N/A	N/A
IncrementalGrowth (ms)	44,97,255	Fail	Fail	N/A	N/A

Key: *CP* cell phone, *SH* smart home, *AG* arcade game, *MT* model transformation, *ES* electronic shopping

Table 5 Test suites sizes obtained for 3-wise generation

	CP	SH	AG	MT	ES
CSP-dedicated	23	61	257	643	841
BinarySplit	207	N/A	N/A	N/A	N/A
IncrementalGrowth	133	N/A	N/A	N/A	N/A

Key: *CP* cell phone, *SH* smart home, *AG* arcade game, *MT* model transformation, *ES* electronic shopping

Table 6 Test suite size and duplicates for the cell phone feature model (3-wise)

	Test suite size	Duplicates
CSP-dedicated	23	0
BinarySplit	207	156
IncrementalGrowth	133	88

This increase in test suite size indicates that there is a trade-off to find between the value of *t* and the number of possible test configurations induced by the feature model. For example, using the alloy-based framework, even if we remove duplicates, leading to test suites of 51 test configurations for BinarySplit and 45 for IncrementalGrowth, reduction in the test suite size is small compared to exhaustive testing (61 possibilities). Naturally, this trade-off heavily depends on the *t*-wise generation framework used, as 3-wise testing is still valuable for the CSP-dedicated approach.

5.4.3 *t*-Tuple occurrence and frequency

5.4.3.1 Cell phone *t*-Tuple occurrence distribution for the cell phone feature model is depicted Fig. 12. Again, the *t*-wise criteria are satisfied with a minimum of 5 occurrences for IncrementalGrowth and 7 for BinarySplit, while the CSP-dedicated approach is getting the optimal value of 1. We also observe that there are tuples uniquely composed of common features, which appear on all the generated test configurations.

Regarding frequency distribution, which is depicted Fig. 13, as for pairwise we observe very similar results. On average, a 3-tuple is appearing in 28% of the generated test configurations. We explain this lower average frequency by the fact that 3-tuples are more difficult to place in test configurations due to dependencies and constraints.

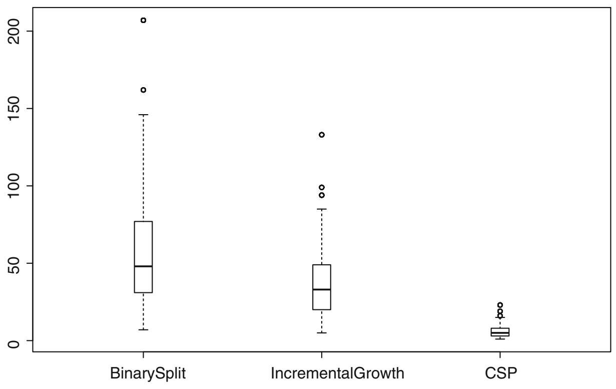

Fig. 12 t-Tuple Occurrence Distribution for the Cell Phone Feature Model (3-wise)

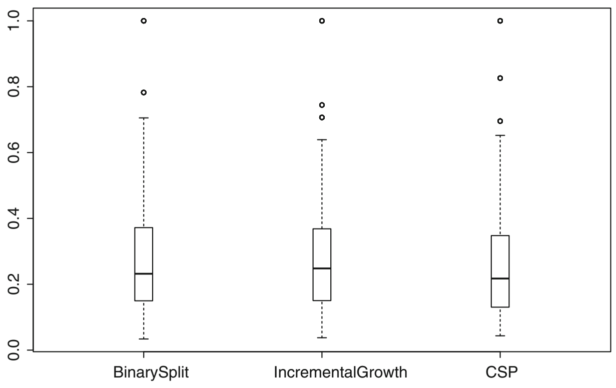

Fig. 13 t-Tuple Frequency Distribution for the Cell Phone Feature Model (3-wise)

5.4.3.2 Arcade game Since for 3-wise the alloy-based framework was not able to terminate computations, we only report obtained results by the CSP-dedicated approach in Table 7.

We observe that the CSP-dedicated approach is able to reach the optimality regarding *t*-Tuple occurrence (minimum 1) and that there are again 3-tuples composed of only common features forced to appear in every test configuration. Frequency values are lower than for the cell phone case as we can predict it, since there are much more possibilities ($3{,}3*10^9$) for the arcade game feature model, and therefore more ways to combine tuples.

Table 7 t-Tuple occurrence and frequency for the arcade game feature model (3-wise, CSP-Dedicated)

	Min	Max	Mean	SD
t-Tuple occurence	1	238	52.6	39.9
t-Tuple frequency	0.004	1	0.22	0.17

5.4.4 Test suite similarity

5.4.4.1 Cell phone Test suite similarity is depicted Fig. 14. While the distribution is similar for the CSP-dedicated approach (Mean = 0.53 for 3-wise and 0.44 for pairwise), both BinarySplit and IncrementalGrowth have much higher similarity (mean = 0.90 for IncrementalGrowth and BinarySplit with a very small standard deviation of 0.04). The great number of duplicates has played a major role toward the obtention of such high similarity scores.

5.4.4.2 Arcade game Table 8 details results for similarity. Interestingly, the choice of 3-wise is adapted for such feature model offering many possibilities. Indeed, similarity is on average two times smaller than for the pairwise case implying that chosen test configurations are more different, which can be regarded as an advantage for coverage.

6 Synthesis

In this section, we synthesize the findings of our comparison and give insights to help the software tester choosing the approach that best suits her or his needs.

6.1 Synthesis

Comparison of main test generation characteristics is presented Table 9.

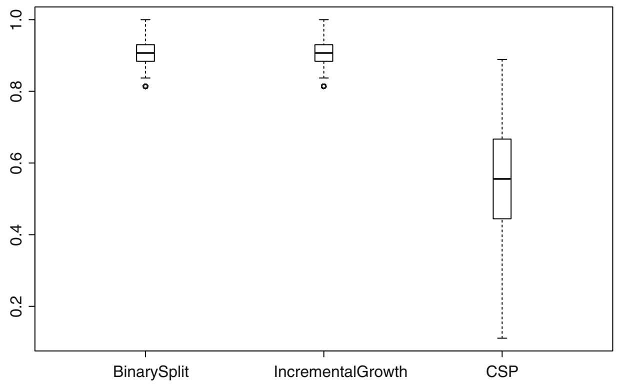

Fig. 14 Test suite similarity distribution for the cell phone feature model (3-wise)

Table 8 Test suite similarity for the arcade game feature model (3-wise, CSP-Dedicated)

	Min	Max	Mean	Std. Deviation
Similarity	0	0.88	0.32	0.25

Table 9 Test generation characteristics

	CSP-based	Alloy-based
# Number of products	+	–
Generation time	+	–
Determinism	+	–
t-wise support	+	+

Based on the above table, several points are worth detailing; *Execution times & Scalability, Quality of generated tests, measures,* and *threats to validity.*

6.1.1 Execution times and scalability

Regarding execution times, results are explicit: The dedicated CSP-based performs much faster on all the examples on which it was applied to for pairwise and 3-wise testing. This performance is also an indicator that the CSP-dedicated approach may scale to larger examples though we cannot provide such evidence.

Regarding "divide-and-compose" strategies, this article provides important results since their original publication in Perrouin et al. (2010). On the one hand, we provided evidence that these strategies may be employed on larger feature models than the 17 features representing the size of the original case study that initially motivated their definition. On the other hand, the price to pay to break complexity into pieces is rather high: this decomposition in a smaller set of problems that are easier to solve is associated with a performance degradation, and for even larger models/higher values of *t*, we still encounter scalability issues. In other terms, "divide-and-compose" strategies improved the scalability up to a certain limit, imposing a complete redesign of the approach and questioning technological choice if we want to push this limit further.

Thus, there is an important issue emerging from this comparison between a generic solution (using Alloy) and a dedicated one (CSP). The scalability dimension is managed either 'a posteriori' (Alloy-Based strategies) or 'a priori' with a dedicated SAT solution (CSP-Dedicated approach). In 'a posteriori' approach, the pairwise generation and the feature model are directly encoded into the generic alloy format: the price to pay for this apparent simplicity is scalability. Since we do not control finely how such an alloy specification is translated to a SAT input, the scalability issue has to be managed 'a posteriori' with divide-and-compose strategies. These strategies transform the scalability issue into a set of smaller problems. The scalability issue is changed into a performance one. (Each smaller problem requires a certain time to be solved.) Thus, solving the scalability issue 'a posteriori' leads to other problems such as performance. On the contrary, for the dedicated CSP-based approach, significant attention has been devoted 'a priori' to the flattening of the model in an efficient structure. Much more effort has been spent proving that the model flattening preserves the semantics of the input feature model, but the benefit of such a dedicated approach is avoiding both scalability and performance issues. This is certainly an important point explaining such a divergence in the results. Given the complexity of generating pairwise tests for feature models, instead of addressing the problem at a general level, one should delve into the details of the encoding and solving technology (SAT, CSP, SMT...) in order to develop 'a priori' scalable solutions. To summarize, the main lesson learnt from these two different approaches is that the simplicity of use of a generic approach does not compensate the efficiency of dedicated CSP-based approach. In practice, a generic approach is useful for prototyping, to precisely define how to solve the problem (e.g. generating pairwise tests), but should then be replaced by a dedicated solution.

6.1.2 Quality of generated tests

Going along the same considerations opposing generic versus dedicated SAT, the CSP approach performs particularly well in minimizing test suites. Although alloy-based strategies can compete on small examples, "divide-and-compose" strategies necessarily generate more test configurations due to the fact that they create test suites based on a subset of all valid tuples and merge them in a single test suite. The largest the examples are, the most likely non-minimal test suites are produced and higher the number of duplicates is, due to more steps required by the algorithms to terminate. Also regarding the size, incrementally driven approaches (IncrementalGrowth and CSP-dedicated) are doing better that the BinarySplit. Yet, more products may also mean more interactions and more chances to reveal complex or rare bugs. In that case, higher similarity degrees can be looked at for regression testing.

Another option is to use higher order *t*-wise which is necessary in some situations (Kuhn et al. 2008), as we demonstrated it in the Arcade Game model with the CSP-dedicated approach. However, in the case of software product lines, the number of generated test configurations can become huge: there are already 238 products for 3-wise for the arcade game model with respect to 46 for the pairwise case. Therefore, on small models, it is probably more fruitful to test exhaustively the SPL rather than using high values of interaction strength.

Determinism also influences the quality of generated tests. The CSP-dedicated approach behaved consistently with respect to the satisfaction of the *t*-wise criteria. On the contrary, the alloy-based approach is not necessarily reaching this optimum, but generates "extreme" test suites which can be sought after (e.g. highly similar test configurations) depending on tester needs.

6.1.3 T-wise support

Both approaches are dealing with *t*-wise generation, and we applied them for pairwise and 3-wise. The method followed by the alloy-based approach is generic, and the strategies do not depend on the value of *t*, which can be set at any arbitrary value. Yet, in practice, we run into scalability and performance issues for the pairwise and 3-wise cases, suggesting that higher value of *t* may not be practically supported. The CSP-dedicated approach performs well on pairwise and 3-wise but requires some adaptation for $t > 3$. The algorithm executes a lot of different operations on the list of parameter/value combinations and on the list containing the pairs of values that need to be covered. The algorithm with $T < 3$ operates on hashmap/hashset combinations, and the algorithm handling $T > 3$ operates on ordinary lists. Hence, the latter thus much slower. We are currently working on further optimizations for $T > 3$.

6.1.4 About measures

While we believe that measures are helpful to determine the merits and issues of test generation approaches, we should not forget that they also witness some specific characteristics inherent to the model under study and the coverage criteria. For example, tuple occurrence is a good indicator of the pairwise coverage criteria. Similarity measures have to be interpreted carefully. As shown by Hemmati et al. (2011), high similarity may both be considered negatively (dissimilar test cases detect different faults) or positively (for diagnosis purpose, similar test cases may help diagnose the location of an error), depending on the testing context (validation or diagnosis). So, when a test suite is said to be better than another because the generated products are dissimilar, we implicitly consider that these products are used in a validation context.

6.1.5 Threats to validity

We tried to be as "fair" as possible in this extension regarding the strategies and measures considering the original paper in which they were initially published (Perrouin et al. 2010).

Having a competing approach was fruitful in the sense that we could verify each other implementation on the same examples. This increases confidence in the trends initially sketched and confirms the applicability of the measures initially defined and generalized in this article.

We mitigated external validity threats by applying our toolsets on several examples and detailing two in this paper. However, performance issues of alloy-based strategies did not allowed us to perform experiments for the arcade game feature model as thoroughly as it was possible for the cell phone one. However, we believe the examples chosen are representative of typical feature models. For example, considering the SPLOT online repository statistics[5], the feature models chosen are balanced with respect to the mean number of features and constraints.

6.2 Additional comparison points

In the following, we discuss additional points that are related to our experience using the toolsets.

6.2.1 Expressivity

Table 10 summarizes the commonalities and differences regarding feature modeling support. The alloy-based approach is more expressive in the sense that it natively supports cardinalities, complex constraints, or the possibility for a feature to have multiple parents.

We are not claiming that this difference in the types of models the two approaches can handle are related to their underlying technologies. It is rather a choice derived from the generic against specific design philosophy.

6.2.2 Usability

Both approaches were designed with the same usability goal: *make CIT approaches accessible to the SPL tester who is not a CSP/SAT-solving specialist*. However, as mentioned in Sect. 5, the alloy-based strategies require to set value for the alloy scope and a timeout value to be used at each step of the "divide-and-compose" algorithm. The CSP-dedicated approach works fully automatically without having any parameter to set, which is better from a usability perspective.

6.2.3 Which approach to choose from ?

These characteristics witness two design philosophies. If a ready-to-use and predictable solution is needed for industrial purposes, then the CSP-dedicated approach is the best choice. If an academic is more interested by evaluating different strategies and see how the quality of the generated results evolves with respect to some parameters, then the alloy-based framework will provide support for such evaluations.

7 Related work

The work related to our research covers SPL testing approaches as well as combinatorial testing and transformations of the feature model.

[5] http://www.splot-research.org/.

Table 10 Expressivity support for input feature models

	CSP-based	Alloy-based
Cardinalities	−	+
Multiple parents	−	+
Binary constraints	+	+
N-ary constraints	−	+

7.1 SPL testing

Concerning test generation for PL (1), McGregor (2001) and Tevanlinna et al. (2004) propose a well-structured overview of the main challenges for testing product lines. SPL testing approaches can roughly be divided in two categories, *product-focused* testing and *SPL-focused* testing. The first category considers a bottom-up approach in which products derived from feature configurations are successively tested. The second category of approaches works top-down from the product line level to extract relevant configurations and derive test cases from them. We cover these two categories with a special emphasis for the latter, in which our research fits in.

7.1.1 Product-focused testing

Studying related work focusing on SPL testing, we identified two common practices:

"SPL-ignorant" techniques: These approaches do not take into account commonalities and variabilities between family members to perform testing. Rather, they consider testing each member individually in an independent way using general testing methods. In Tevanlinna et al. (2004), the authors refer to this approach as product-by-product testing. However, considering the number of derivable products of today's SPLs, this approach is unpractical. This expected result has been confirmed empirically (Ganesan et al. 2007).

Reuse-Techniques: Methods of this category utilize reuse-techniques to reduce the test effort. These approaches either make use of regression testing techniques to incrementally test products or realize the reuse of domain tests during application testing. Reusing domain tests created during domain engineering for product tests is a very popular approach especially in the model-based testing community. A summary of model-based testing approaches for SPLs can be found in Oster et al. (2011). Uzuncoava et al. (2008) use alloy to generate a test suite incrementally from the specification of a product, directly modeled as alloy formulae. The interesting point in this work is that tests are reused from one product to another in a cumulative way. Hence, such a product-focused approach allows to perform cumulative coverage as described in Cohen et al. (2006). However, even when they efficiently take the SPL's features to minimize the testing effort, they do require a particular product to start with. Our approaches do not require such an "initial" product to generate test configurations. Yet, the CSP-dedicated approach is able to take into account already tested configurations into account to complete them with t-wise based generation.

7.1.2 SPL-focused testing

Subset-Heuristics: This approach aims at reducing the effort for testing by extracting a subset of feature combinations or products. Instead of testing every product of the SPL, a subset for testing is created. We identified two different methodologies: methods generating a subset of representative products for testing purposes for the whole SPL, and approaches using combinatorial testing. In Scheidemann (2007), the author introduces an approach generating a representative set for each requirement. The major disadvantage of this approach is the fact that it does not scale with real-world SPLs and that the effort to set up the representative set is enormous. In Yoon et al. (2007), the authors propose a method to generate test plans covering user-specified portions of the huge number of possible configurations of a component-based software system.

7.2 Combinatorial testing

McGregor initially introduced combinatorial testing to SPLs in McGregor (2001). However, he neither describes how combinatorial testing may be applied to SPLs nor describes how SPL models like FMs or OVMs can be mapped onto an appropriate representation to apply existing combinatorial testing algorithms.

Cohen et al. use the OVM approach to model the variable and common parts of the SPL which are mapped onto a relational model. This relational model serves as a semantic basis for defining coverage criteria for the SPL under test (Cohen et al. 2006). Furthermore, Cohen et al. describe the development of combinatorial interaction testing (CIT) achieving a desired level of coverage. Kuhn et al. (2004) led to the definition of pairwise testing and, then, its generalization to t-wise testing. Cohen et. al. have applied CIT to systematically select configurations/products (2006) that should be tested. They consider various algorithms in order to compute configurations that satisfy pairwise and *t*-wise criteria (Cohen et al. 2007).

Our two implementations regarding combinatorial testing differ in the following ways:

– (*t*-wise:) goes along the same lines but deals with scalability of the test generation, noting that CIT+SAT approaches do not scale directly with real-case feature diagrams, such as the AspectOPTIMA SPL example.
– CSP-based: combines graph transformation, a well-known pairwise algorithm associated with forward checking, to generate a set of products achieving 100% pairwise interaction coverage in the whole SPL on the basis of the corresponding FM. The reason for choosing a CSP approach for pairwise testing is that we want to apply this approach to the FMT approach that utilizes large ranges of values. Especially for such problems, a CSP-based approach seems to be a natural choice (Bennaceur 2004; Westphal and Wölfl 2009).

7.3 Feature model translation

Since both approaches are based on feature modeling, we provide related work to our translation algorithms.

7.3.1 Cartesian flattening

In White et al. (2009), the authors realize a Cartesian flattening of FMs, which is a similar to our flattening algorithm. There, the motivation is to translate the FM into a knapsack problem which is then used to generate highly optimal architectural variants/products of the SPL. There are some significant differences to our flattening approaches: In White

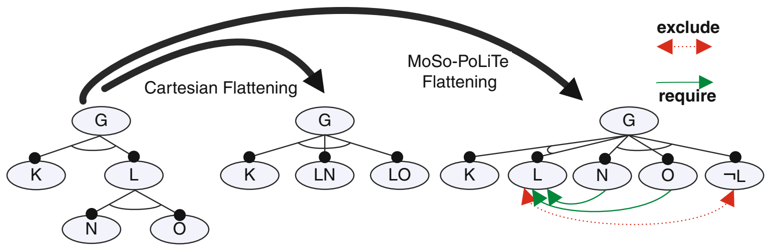

Fig. 15 Comparison of the flattening approaches of an alternative parent with alternative-child elements

et al. (2009), cardinality groups (or-groups in our approach) are translated into an XOR (alternative group in our approach) with a maximum number boundary.

For testing purposes, all valid feature combinations need to be identified. We lose semantic equivalence between the original FM and the flat FM if we use a boundary, limiting the maximum number of combinations. In White et al. (2009), a different rule for flattening an alternative group beneath an alternative parent node is presented. Figure 15 shows an abstract example used in White et al. (2009).

In the Cartesian flattening approach, the features **N** and **O** are merged with its parent node. Let us now assume that the parent feature **L** is required by some other feature **X**. The feature **X** would then require **L,N** XOR **L,O**. As you can imagine, this dependency relation cannot be captured using a binary constraint such as the ones we support in our subset extraction algorithm. Because of distinct needs, White et al. apply different transformation rules to prepare the FM for their algorithms. This approach offers additional evidence that it is possible to change the structure of the FM to apply well-known algorithms for different purposes. Unfortunately, due to fact that not all rules keep semantic equivalence, we cannot apply this method for our *t*-wise approaches.

7.3.2 Feature model into alloy

We choose a model-driven technique to automatically map a feature diagram into an alloy input format. The user of the approach can thus manipulate directly feature diagrams and transform them directly in alloy. A formalization for feature models in alloy can be found in Gheyi and Borba (2006), but is not dedicated to testing, and feature diagrams have to be written by hand. Our work focuses on testing the SPL as whole rather than individual products. Indeed, these techniques of SPL testing are complementary; our approaches focus on automated selection of products, which can then be individually tested.

8 Conclusion

As software product line engineering is taking momentum in software engineering, testing software product lines is of growing importance. A particular problem in SPL testing is the number of test cases to consider, which increases exponentially with the number of features the SPL owns. In this article, we focus on reducing the number of test cases in a product line context. In particular, we compared two approaches (CSP-dedicated, alloy-based) for test cases reduction both based on *t*-wise interaction testing.

At first sight, both approaches are functionally equivalent from the *t*-wise testing perspective, since they provide the same guarantee in terms of pairwise interaction coverage: they ensure that all valid pairs of features with regard to the feature model notation, dependencies, and hierarchy are tested together. Furthermore, both approaches have the main advantage that the test suite consists of configurations that can be tested using well-known single system test methods from the software engineering community. In order to make the comparison possible, we provide a set of metrics, capturing the number of generated test configurations (the test effort) and the similarity degrees between these test configurations.

However, while functionally equivalent, comparing the different philosophies using these metrics, some major differences were identified. Compared to alloy-based strategies, the CSP-dedicated philosophy:

- is deterministic and more stable and finds a better/smaller solution
- is focused on pairwise and 3-wise interaction testing, but could be adapted to higher interaction strengths. The alloy-based testing approach is independent of the value of *t*.
- currently only supports binary constraints between features. Thus, n-ary constraints cannot be solved by the dedicated CSP approach. This is a drawback compared to a more generic toolset, like alloy offers.
- is much faster, especially on large/highly constrained feature models. While pragmatic, alloy-based strategies failed to produce quality results in due time. Several improvements could be envisioned such as conversion of the feature model in alloy or the usage of atomic sets (Benavides et al. 2010). Yet, issues that emerged from the comparison are severe enough to require redesigning this alloy-based approach from the start.

This work opens two main research perspectives. First, we would like to extend our comparison approach and metrics in a fully-fledged evaluation framework to assess various CIT-based solutions for SPL testing. We are convinced that detailed evaluation of these techniques is the key to gaining confidence in CIT-based approaches and toolsets and so help such toolsets permeate SPL testing practice.

Second, we outlined two strategies to deal with scalability: one working a priori by optimizing the feature model and its flattening, and the other a posteriori by providing “divide-and-compose” strategies decomposing the problem in smaller solvable problems. Although such strategies can significantly degrade the quality of generated results, they may be the last option if a priori optimization failed. We would like to investigate the combination of a priori and a posteriori philosophies on very large feature models such as the linux kernel (Berger et al. 2010) with more than 6000 features or with additional elements, such as attributes or priorities.

Acknowledgments The authors would like to thank Professor Andy Schürr for his valuable comments on the paper. This research was partly funded by the NAPLES project funded by the Walloon Region (Belgium).

References

Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In: *Software product line conference (SPLC)* (pp. 7–20).

Batory, D., Benavides, D., & Ruiz-Cortés, A. (2006). Automated analysis of feature models: Challenges ahead. *Communications of the ACM*.

Benavides, D., Segura, S., & Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems, 35*(6), 615–636.

Bennaceur, H. (2004). A comparison between SAT and CSP techniques. *Constraints, 9*(2), 123–138.

Berger, T., She, S., Lotufo, R., Wasowski, A., & Czarnecki, K. (2010). Variability modeling in the real: A perspective from the operating systems domain. In: *Proceedings of the IEEE/ACM international conference on automated software engineering* (pp. 73–82). New York, NY, USA: ACM, automated software engineering conference (ASE) '10.

Bryce, R., & Colbourn, C. (2009). A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability, 19*(1), 37–53.

Bryce, R. C., & Colbourn, C. J. (2006). Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology, 48*(10):960–970, advances in Model-based Testing.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., & Grose, T. (2003). *Eclipse modeling framework*. The Eclipse Series, Addison Wesley Professional.

Calvagna, A., & Gargantini, A. (2009). Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In: *International conference on tests and proofs* (pp. 27–42). Berlin, Heidelberg: Springer.

Calvagna, A., & Gargantini, A. (2008). A logic-based approach to combinatorial testing with constraints. In Beckert, B., Hähnle, R., (Eds.), *Tests and proofs* (Vol. 4966, pp. 66–83). Berlin/Heidelberg: Springer, Lecture Notes in Computer Science.

Cartaxo, E. G., Machado, P. D. L., & Neto F. G. O. (2011). On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification & Reliability, 21*, 75–100.

Classen, A., Heymans, P., & Schobbens, P. (2008). What's in a feature: A requirements engineering perspective. In *Proceedings of the theory and practice of software, 11th international conference on fundamental approaches to software engineering* (pp. 16–30). Springer.

Clements, P., & Northrop, L. (2001a). *Software product lines: Practices and patterns*. Reading, MA, USA: Addison Wesley.

Clements, P., & Northrop, L. (2001b). Software product lines: practices and patterns. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Cohen, M., Dwyer, M., & Shi, J. (2007). Interaction testing of highly-configurable systems in the presence of constraints. In *International symposium on software testing and analysis* (Vol. 4961/2008, pp. 129–139).

Cohen, M. B., Dwyer, M. B., & Shi, J. (2006). Coverage and adequacy in software product line testing. In *ROSATEA@ISSTA* (pp. 53–63).

Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering, 23*(7), 437–444.

Czarnecki, K., Wasowski, A. (2007). Feature diagrams and logics: There and back again. In *11th software product line conference* (pp. 23–34). Kyoto, Japan: IEEE Computer Society.

Czarnecki, K., & Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *Generative programming and component engineering (GPCE)* (Vol. 3676, pp. 422–437). Springer, LNCS.

Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice, 10*(1), 7–29.

Ganesan, D., Knodel, J., Kolb, R., Haury, U., & Meier, G. (2007). Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. In: *11th International software product line conference* (pp. 74–83). Los Alamitos, CA, USA: IEEE Computer Society.

Griss, M. L., Favaro, J., & d' Alessandro, M. (1998). Integrating feature modeling with the RSEB. In *Fifth international conference on software reuse* (pp. 76–85). Washington, DC, USA.

Haralick, R., & Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*(3), 263–313.

Hemmati, H., & Briand, L. (2010). An industrial investigation of similarity measures for model-based test case selection. In International symposium on software reliability engineering (ISSRE) (pp. 141–150). Los Alamitos, CA, USA: IEEE Computer Society.

Hemmati, H., Arcuri, A., & Briand, L. (2010). Reducing the cost of model-based testing through test case diversity. In *22nd IFIP international conference on testing software and systems (ICTSS)— formerly TestCom/FATES* (Vol. 6435/2010, pp. 63–78).

Hemmati, H., Arcuri, A., & Briand, L. (2011). Empirical investigation of the effects of test suite properties on similarity-based test case selection. In 4th international conference on software testing, verification and validation (ICST) (pp. 327–336), Berlin, Germany.

Jackson, D. (2006). *Software abstractions: Logic, language, and analysis*. MIT Press: Cambridge.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, S. (1990). *Feature-oriented domain analysis (FODA) feasibility study*. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering, 5*, 143–168.

Kuhn, R., Lei, Y., Kacker, R. (2008). *Practical combinatorial testing: Beyond pairwise*. *IT Professional, 10*, 19–23. http://doi.ieeecomputersociety.org/10.1109/MITP.2008.54.

Kuhn, D. R., Wallace, D. R., & Gallo, A. M. (2004). Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering, 30*(6), 418–421.

Lei, Y., & Tai, K. (1998). In-parameter-order: A test generation strategy for pairwise testing. In *IEEE high assurance systems engineering symposium* (pp. 254–261).

Lei, Y., Kacker, R., Kuhn, D., Okun, V., & Lawrence, J. (2008). IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability, 18*(3), 125–148.

Mahajan, Y. S., Fu, Z. S. M. (2004). Zchaff2004: An efficient sat solver. In *SAT 2004* (pp. 360–375).

McGregor, J. (2001). *Testing a software product line*. Tech. Rep. ESC-TR-2001-022, CMU/SEI.

Mendonça, M., Wasowski, A., & Czarnecki, K. (2009). Sat-based analysis of feature models is easy. In *13th international software product line conference (SPLC)* (pp. 231–240). San Francisco, CA, USA.

Mendonca, M., Branco, M., & Cowan, D. (2009). SPLOT: Software product lines online tools. In *Proceeding of the 24th ACM SIGPLAN conference companion on object oriented programming systems languages and applications* (pp. 761–762). ACM.

Metzger, A., Pohl, K., Heymans, P., Schobbens, P. Y., & Saval, G. (2007). Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *IEEE conference on requirements engineering* (pp. 243–253). Delhi, India: IEEE Computer Society.

Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., & Troyansky, L. (1999). Determining computational complexity from characteristic phase transitions. *Nature, 400*(6740), 133–137.

MoSo-PoLiTe (2011). http://www.sharq.tu-darmstadt.de/projects/mosopolite/. Accessed 8th April 2011.

Muller, P. A., Fleurey, F., & Jézéquel, J. M. (2005). Weaving executability into object-oriented metalanguages. In *MODELS/UML*. Springer.

Niklas, E., & Niklas, S. (2005). MiniSat: A SAT solver with conflict-clause minimization, poster. In *SAT 2005*.

Oster, S., Markert, F., & Ritter, P. (2010). Automated incremental pairwise testing of software product lines. In Bosch, J., & Lee, J. (Eds.), *Software product line conference (SPLC)* (Vol. 6287, pp. 196–210). Springer, Lecture Notes in Computer Science.

Oster, S., Wübbeke, A., Engels, G., & Schürr, A. (2011). Model-based software product lines testing survey. In Zander, J., Schieferdecker, I., & Mosterman, P. (Eds.), *Model-based testing for embedded systems*. CRC Press Taylor & Francis, to appear on September 9th, 2011.

Perrouin, G., Klein, J., Guelfi, N., & Jézéquel, J. M. (2008). Reconciling automation and flexibility in product derivation. In *Software product line conference (SPLC)* (pp. 339–348). Limerick, Ireland: IEEE Computer Society.

Perrouin, G., Sen, S., Klein, J., Baudry, B., & le Traon, Y. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *International conference on software testing, verification, and validation (ICST)* (pp. 459–468). IEEE Computer Society, Paris, France.

Phadke, M. (1995). *Quality engineering using robust design*. Upper Saddle River, NJ, USA: Prentice Hall PTR

Pohl, K., Böckle, G., & van der Linden. F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Gheyi, R. T. M., & Borba, P. (2006). A theory for feature models in alloy. In *First alloy workshop* (pp. 71–80).

Scheidemann, K. (2007). *Verifying families of system configurations*. Doctoral Thesis TU Munich.

Schobbens, P. Y., Heymans, P., Trigaux, J. C., & Bontemps, Y. (2006). Feature diagrams: A survey and a formal semantics. In *Requirements engineering, IEEE international conference on* (pp. 139–148).

Schobbens, P., Heymans, P., Trigaux, J., & Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks, 51*(2), 456–479.

Tan, P., Steinbach, M., Kumar, V., et al. (2006). *Introduction to data mining*. Boston: Pearson Addison Wesley

Tevanlinna, A., Taina, J., & Kauppinen, R. (2004). Product family testing: A survey. SIGSOFT Software Engineering Notes, 29(2), 12–12.

Torlak, E., & Jackson, D. (2007). Kodkod: A relational model finder. In *Tools and algorithms for construction and analysis of systems* (Vol. 4424/2007, pp. 632–647).

Uzuncaova, E., Garcia, D., Khurshid, S., & Batory, D. (2008). Testing software product lines using incremental test generation. In *ISSRE* (pp. 249–258). IEEE Computer Society.

Westphal, M., & Wölfl, S. (2009). Qualitative csp, finite csp, and sat: Comparing methods for qualitative constraint-based reasoning. In *IJCAI'09: Proceedings of the 21st international jont conference on artificial intelligence* (pp. 628–633). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

White, J., Dougherty, B., & Schmidt, D. C. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software, 82*(8), 1268–1284.

Yoon, I., Sussman, A., Memon, A., & Porter, A. (2007). Direct-dependency-based software compatibility testing. In *Automated software engineering conference (ASE)* (pp. 409–412). Atlanta, Georgia, USA.

Ziadi, T., & Jézéquel, J. M. (2006). Product line engineering with the UML: Deriving products. In *Families research book*. Springer.

Author Biographies

Gilles Perrouin is currently a postdoctoral researcher at the Faculty of Computer Science, University of Namur (FUNDP), Belgium. He is a member of the PReCISE research centre, in which he explores Software Product Lines, Modelling, Requirements Engineering and Testing. Dr. Gilles Perrouin also serves regularly as a (co)-referee for journals and conferences. Dr. Gilles Perrouin holds a joint PhD Degree from the University of Luxembourg and Namur.

Sebastian Oster was born in Hamburg, Germany in November, 1981. He studied Computer Science at the University of Duisburg-Essen and received his Diploma in August, 2007. Currently, he is a research associate at the Technische Universität Darmstadt finishing is doctoral thesis. His main research activities are in the field of Software Product Line engineering, model-based testing and quality assurance.

Sagar Sen obtained a PhD from the University of Rennes 1, France and M.Sc. from McGill Univeristy, Montreal, Canada as a Commonwealth Fellow. He has a Postdoc from INRIA Sophia-Antipolis, France on applying MDE techniques and formal methods to computer vision systems. Here he supervised projects to develop software prototypes for self-adaptive computer vision. Currently, he is a Research Fellow at Ecole des Mines, Nantes. His research interests are in model-driven software development/validation, software product lines, scaling formal methods, and their application to other domains such as computer vision.

Jacques Klein In 2003, Dr. Jacques Klein received an engineering degree in Computer Science from the ESSAIM (ENSISA) and a Master in Computer Science from the University of Haute-Alsace, France. He received a PhD degree in Computer Science from the University of Rennes, France in 2006 for a dissertation on the weaving of behavioral models (UML) in a Model-Driven Engineering and Product Line contexts. Part of his doctorate work has been to propose new software engineering tools to reduce the cost and the risk of software development by adapting software systems to wide ranges of new contexts. In 2007/2008, he worked for two years at the University of Luxembourg as a scientific collaborator. He participated in the SPLIT project to propose new transformation languages to support software product lines. He is also participating in the VERITY project to ease the design of reliable software systems. Finally, he successfully proposed two research projects, one on the security testing of resilient systems and one on the mix of SPL and AOSD. In 2009, he moved to a full time position at the CRP-Gabriel Lippmann, Belvaux, Luxembourg, to work on several IT projects as project manager. He worked in the ISC department of the CRP-Gabriel Lippmann. One of the main research topics of the research centre is to foster the development of high-value added business services by using Service-Oriented Architecture (SOA). In March 2010, he came back to the University of Luxembourg to work in the team of the Prof. Yves Le Traon.

Benoit Baudry received his PhD in computer science from the University of Rennes, France in 2003. He first worked at CEA (French government nuclear agency) before joining INRIA in 2004. He is now a researcher in software engineering in the Triskell team at INRIA Rennes Bretagne Atlantique. In 2008 he was an invited scientist at Colorado State University. His research interests include software testing, aspect-oriented software development, model transformation and model-driven development. He is the vice-chair of the steering committee of the International Conference on Software Testing Verification and Validation. He is a member of the IEEE and the IEEE Computer Society.

Yves le Traon is professor at University of Luxembourg in the domain of software engineering, reliability, validation and security. He is also a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT), where he leads the joint research group SERVAL (SEcuRity and VALidation of services and networks). His research interests include OO testing, design for testability, model-driven validation, model based testing, evolutionary algorithms and software measurement. Professor Le Traon received his engineering degree and his PhD in Computer Science at the "Institut National Polytechnique" in Grenoble, France, in 1997. From 1998 to 2004, he was an associate professor at the University of Rennes, in Brittany, France. He is the co-founder of the Triskell INRIA team, which focuses on innovating design, modeling and testing techniques, such as Model-driven Engineering. During this period, Professor Le Traon studied design for testability techniques, validation and diagnosis of object-oriented programs and component-based systems. From 2004 to 2006, he was an expert in Model-Driven Architecture and Validation in the EXA team (Requirements Engineering and Applications) at "France Télécom R&D". In 2006, he became professor at Telecom Bretagne (Ecole Nationale des Télécommunications de Bretagne) and led the SERVAL team (Validation and Security of Services and Networks), where he pioneered the application of testing for security assessment of web-applications, P2P systems and the promotion of intrusion detection systems using contract-based techniques. He is author of more than 90 publications in international journals and conferences.