# Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage

Beatriz Pérez Lamancha, Macario Polo Usaola

# Testing product generation in Software Product Lines using pairwise for features coverage

Beatriz Pérez Lamancha[1], Macario Polo Usaola[2]

[1]Software Testing Centre, Republic University, Montevideo, Uruguay
bperez@fing.edu.uy
[2]Alarcos Research Group, UCLM, Ciudad Real, Spain
macario.polo@uclm.es

**Abstract.** A Software Product Lines (SPL) is *"a set of software-intensive systems sharing a common, managed set of features that satisfy 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"*. Variability is a central concept that permits the generation of different products of the family by reusing core assets. It is captured through features which, for a SPL, define its scope. Features are represented in a feature model, which is later used to generate the products from the line. From the testing point of view, testing all the possible combinations in feature models is not practical because: (1) the number of possible combinations (i.e., combinations of features for composing products) may be untreatable, and (2) some combinations may contain incompatible features. Thus, this paper resolves the problem by the implementation of combinatorial testing techniques adapted to the SPL context.

**Keywords:** testing, software product lines, combinatorial testing, feature coverage, pairwise

## 1 Introduction

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features which satisfy the specific needs of a particular market segment or mission and which are developed from a common set of core assets in a prescribed way [1]. Products in a line share a set of characteristics (commonalities) and differ in a number of variation points, which represent the variabilities of the products. Software construction in SPL contexts involves two levels: (1) Domain Engineering, which refers to the development of common features and the identification of the variation points; (2) Product Engineering, where each concrete product is built. At this second level, commonalities must be included in the products, and the corresponding variation points must be adequately managed.

Traceability and reuse are fundamental aspects in SPL development and, thus, testing is an essential task in this kind of software development paradigm. In fact, and error introduced in a common part which remains undetected may affect all the products in the line; in the same way, an error in a variation point will be propagated to all the products which include that variation.

In previous works [2], a framework for model-driven testing in SPL was defined. The framework includes a methodological approach to automate the generation of test models from SPL design models, and specifies a way to deal with variability: given a SPL design, the approach produces a test model which includes enough information to build specific test cases both for the common features of the line, as well as for the specific characteristics of the variation points finally implemented in each product.

However, just as the execution of integration testing is required after unit testing in a classic testing process, features of a SPL must be also tested when they are integrated into a single product; finding no faults in core assets at the Domain Engineering level does not mean that its transformation into a concrete product (generated at the Product Engineering level) does not introduce defects. In the same way, the fact of not discovering errors when an isolated feature is tested does not guarantee that a given product with that very same feature, together with others, will be free of defects, even in those features which, apparently, were previously error-free.

From a testing point of view, testing all the possible feature combinations in a SPL is unfeasible. In a SPL with just 5 features and 4 variants, the number of products that can be generated is $4^5$=1024. Testing each possible product is expensive and unrealistic for software industry.

This paper defines a strategy for testing products proceeding from SPL feature models. The strategy uses pairwise as its covering criterion, in the sense that all the pairs of features must be included and tested in at least one product. The Orthogonal Variability Model (OVM, [3]) is used to represent the variation points and its variants. This does not mean any loss of generality in the proposal, since any other metamodel can be used to represent the feature model. In fact, the same rules would be applied to obtain the test suite of products to test.

One of the most widely-used strategies to obtain pairwise coverage is the AETG algorithm [4], which works in polynomial time. In the SPL context, the algorithm must be modified to deal with *requires* and *excludes* relationships between features. If a variant in a feature excludes a variant in another feature, then the pair between both variants must not be present in any product. One of the SPLs we use as a case study consists of a system to play board games over the internet. Thus, we may be dealing with four variation points (Game, Dice, Opponent and Number of Players) and several variants in each ({Ludo, Trivial, Chess, Checkers}, {Dice, No-dice} {Person, Computer}, {2, >2}). *Ludo* or *Trivial* with *No-dice* make no sense, and neither do *Chess* or *Checkers* with *Dice* or with more than two players (*>2*). Restrictions between pairs such as these are not contemplated by AETG and, therefore, the algorithm has been modified to not consider undesired pairs.

This change to the algorithm is not restricted to SPL testing, since it is common to test systems excluding invalid combinations of test values. Pairwise assumes that many errors only arise from the specific interaction of certain values of two or more parameters [5], but in the actual practice of software testing, test cases containing

undesired pairs are often removed from the final test suite. For these situations, the improved version of AETG can be also used.

## 2 Representing variability in SPL

Variability is a central concept in product family development. It allows for the generation of different products in the family by reusing *core assets*. Variability is captured through *features*. A feature can be a specific requirement, a selection amongst optional or alternative requirements, or can be related to certain product (functionality, usability, performance, etc) or implementation characteristics (size, execution platform, standards compliance, etc)[6].

Domain engineering techniques are used to systematically extract features from existing or planned members of a product line. Feature trees are used to relate features to each other in various ways, showing sub-features, alternative features, optional features, dependent features or conflicting features [6]. Examples of these methods are FODA [7], FORM [8], FeatuRSEB [9], among others. Figure 1 shows a feature model example.

In this work, the proposal by Pohl et al. [3] is used to manage the variability, defined in their **Orthogonal Variability Model** (OVM). In OVM, variability information is saved in a separate model containing data about variation points and variants. A *variation point* may involve several *variants* in, for example, several products. OVM allows the representation of dependencies between variation points and variable elements, as well as associations among variation points and variants with other software development models (i.e., design artifacts, components, etc.). Variation points and variants are the core concepts of the OVM language. Each variation point offers at least one variant. Additionally, the constraints-associations between these elements describe dependencies between variable elements [3].

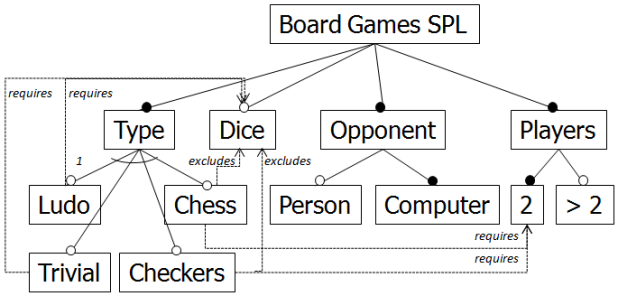


**Figure 1 – Feature Model for Board Game SPL**

OVM includes a graphical notation: Variation Points are represented by triangles and their variants with a rectangle. Dotted lines represent *optional* variants (i.e., they can be omitted in some products), whereas solid lines represent *mandatory* variants (they are present in all products). The associations between variants may be *requires_V_V* and *excludes_V_V*, depending on whether they denote that a variation *requires* or *excludes* another variation. In the same way, associations between a

variation and a variation point may be *requires_V_VP* or *excludes_V_VP*, also to denote whether a variation requires or excludes the corresponding variation point.

Figure 2 shows the OVM model for the board game SPL. The board games share a wide set of characteristics, such as the existence of a board, one or more players, the use of dice, possibility of taking pieces, presence or absence of cards, policies related to the assignment of the turn to the next player, etc. As we showed in the previous section, there are 4 four variation points (Game, Dice, Number of players and Opponent) and 4, 2, 2, and 2 variants respectively.

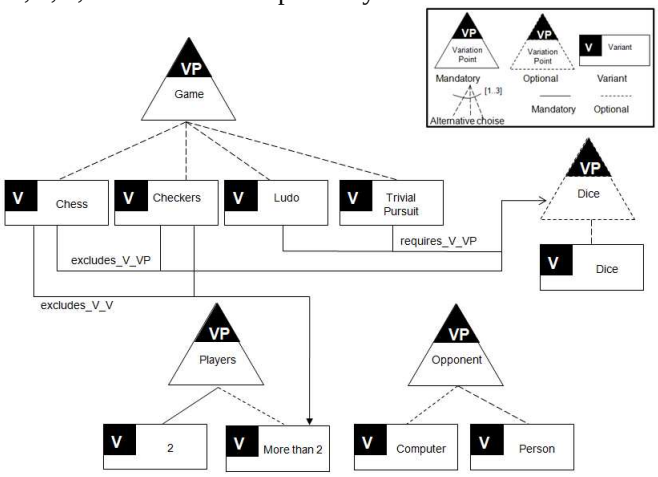


**Figure 2 – OVM model for Board Game SPL**

In previous works, a specific UML profile to represent OVM models was defined[10]. Figure 3 shows the same information as in Figure 2 but using the profile.

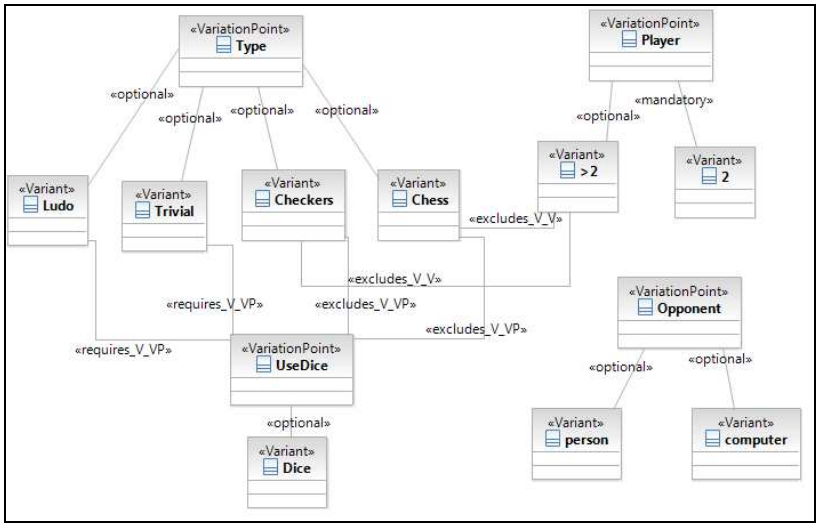


**Figure 3 – Board Games SPL using UML profile for OVM**

The use of one or another metamodel is independent for the process: Roos-Frantz, Benavides and Ruiz-Cortés[11] have shown that it is possible to use model-to-model transformation in order to generate a target model conforming to an OVM metamodel, preserving all the semantics in the source models.

## 3. Combination testing strategies and related works

Combination strategies are a class of test-case selection methods where test cases are created by the combination of "interesting values", which have been previously identified by the tester. The input of all these testing strategies is a set of sets (parameters), each with some elements (values). The output is a set of combinations, all of them composed of one element from each input set.

Like many test-case selection methods, combination strategies are based on coverage. In the case of combination strategies, coverage is determined with respect to the use of the parameter values that the tester decides are interesting. Thus, for example, a test suite satisfies *Each-use* (also known as *1-wise*) coverage when each test value is included in at least one test case in the test suite. *Pairwise* (also known as *2-wise*) coverage requires that every possible pair of interesting values of any two parameters be included in some test case. Note that the same test case may cover more than one unique pair of values. A natural extension of pairwise coverage is *t-wise*, which requires that every possible combination of interesting values of *t* parameters be included in some test case in the test suite.

Different test generation strategies have been published for pairwise testing, some of them collected in a survey article by Grindal, Offut and Andler [12]. Since the problem of generating minimum pairwise test sets is NP-complete, different researchers have developed strategies to generate near-minimum pairwise test sets, such as algorithms based on orthogonal arrays [13] or covering Arrays [14]. One important drawback to these two methods is that they can only be applicable to sets (parameters) with the same number of elements (test values), which restricts the actual application of these techniques.

One very interesting approach for pairwise coverage was proposed by Cohen et al. [4], who developed the AETG algorithm for *t-wise* coverage (Figure 4).

Considering the combinatorial strategies in SPL context, Perrouin et al. [15] uses t-wise for feature coverage using SAT solvers, dividing the set of clauses (transformed from a feature diagram) into solvable subsets. They use the features as parameters; each parameter may receive two values (*true* or *false*) to represent the presence or absence of the feature: thus, the combination strategy followed up by these authors may produce much more products to be tested that those required. Actually, the feature model should have information enough to consider the relationship between a variation point and its variants. The authors take into account *mandatory* and *optional* features, the *requires* relationship, but no the *excludes* one. In our approach, variation points are considered as the parameters: instead of having two boolean values for each feature, we process the feature model to consider that each feature variant is a parameter value. Moreover, all the relationships defined in OVM are processed to include or exclude pairs. The use of combinatorial testing to cover features in SPLs

has also been the focus of previous works by McGregor[16] and Cohen et al.[17], who address the issue of pairwise testing through orthogonal arrays and covering arrays respectively. However, we consider that these approaches have a key limitation in that they that require all of the features to have the very same number of variants. In our experience (also corroborated by examples found in the literature[18]), this is unrealistic, since features very rarely offer the very same number of variants. Moreover, these works neither consider the *excludes* relationship between features. Indeed, we decided to improve the AETG algorithm because the number of values in each parameter can be variable.

Assume that we have a system with $k$ test parameters and that the *i-th* parameter has $li$ different values.

Assume that we have already selected $r$ test cases. We select the $r + 1$ by first generating $M$ different candidate test cases and then choosing one that covers the most new pairs.

Each candidate test case is selected by the following greedy algorithm:

1. Choose a parameter $f$ and a value $l$ for $f$ such that that parameter value appears in the greatest number of uncovered pairs.

2. Let $f_1 = f$. Then choose a random order for the remaining parameters. Then, we have an order for all $k$ parameters $f_1, ..., f_k$.

3. Assume that values have been selected for parameters $f_1, ..., f_k$. For $1 \leq i \leq k$, let the selected value for $f_i$ be called $v_i$. Then, choose a value $v_{k+1}$ for $f_{k+1}$ as follows.

   For each possible value $v$ for $f_k$, find the number of new pairs in the set of pairs $\{f_{k+1} = v \text{ and } f_i = v_i \text{ for } 1 \leq i \leq k\}$. Then, let $v_{k+1}$ be one of the values that appeared in the greatest number of new pairs.

   Note that, in this step, each parameter value is considered only once for inclusion in a candidate test case. Also, that when choosing a value for parameter $f_{j+1}$, the possible values are compared with only the $k$ values already chosen for parameters $f_1,..., f_k$.

**Figure 4. Original explanation of the AETG algorithm for covering pairwise [4]**

## 4 Selection of products to test in SPL

Testing all the existing combinations in a feature model is similar to exhaustive testing in traditional software development and is economically unviable. The objective, then, is to select a testing strategy to decide what products will be tested, assuming that these products are representative of the set of all the possible products in the line.

Obviously, if the core assets are tested in isolation, it is less likely to find defects when they are assembled in a product. However, it is necessary to ensure that there are no undesired results when the product is generated. Rather than exhaustive testing,

a combinatorial approach can help SPL engineers to decide what combinations of features are more interesting to test, based on feature coverage and feature dependencies.

In our proposal, the variation points are the parameters for the pairwise, whereas variations are the values of the parameters. First, we define how pairs between features are generated from the OVM model and second, how the test cases are selected using the modified version of the AETG algorithm. Each test case is a combination of features, i.e., a product of the line.

### 4.1 Building the pairs set

We use the OVM model to describe the features and the relationships between features. As described in Section 2, OVM is used to exemplify the proposal, since the results of this study can be extrapolated to any other representation of feature models.

There are four parameters in the example of the Board Games SPL: Game, Dice, Players and Opponent. The values for the parameters are the Variations for each Variation Point. Table 1 shows the parameters and its values for the Board Games SPL.

| | features | | | |
|---|---|---|---|---|
| variants | type | dice | opponent | players |
| | ludo | dice | person | 2 |
| | chess | | computer | moreThan2 |
| | trivial | | | |
| | checkers | | | |

**Table 1 – Features and variants**

Actually, the information in Table 1 is incomplete, as it is necessary to add the information about the relations between the parameters and their values. Table 1 is augmented with the following information:

- **Variation Point**: If the variation point is optional, then a new value is added. This value states that the entire variation point is not selected for the product. The rule is:

  | If VP is an optional Variation Point with *n* variants, then the VP parameter has *n+1* values: one for each variant and one more for the value "no". |
  |---|

  In the example, the variation point Dice is optional and the "no" value is added.
- **Variants**: In OVM the relationship between a Variation Point and a Variant can be optional, mandatory or alternative. For each case:
  - Optional: The optional variability dependency states that a variant can (but does not need to) be part of a product line application [3]. No values are added for this relationship.
  - Mandatory: The mandatory variability dependency states that a variant must be selected for an application if and only if the associated variation point is part of the application [3]. For example, variant 2 in Figure 2 for the Players variation point is mandatory: then, value *2* can be present in all products of the line (because the Player variation point is also mandatory) and the variant *More than 2* is optional. The rule is:

If VP is a variation point with *n* variants, being *k* mandatory and *n-k* optional, then the parameter VP has *(n-k)+1* values, where the first value is the selection of all the *k* mandatory values together, and the *n-k* remaining values are pairs of each optional value with the first value.

For the example, since value 2 is mandatory, it must be added to the other values: i.e., MoreThan2 and (2,MoreThan2), which is the second value for the parameter Player.

- Alternative: The alternative choice groups a set of variants that are related through an optional variability dependency to the same variation point and defines the range for the amount of optional variants to be selected for this group [3]. The alternative contains two attributes: *min* and *max*. The rule is:

If VP is a variation point with *n* optional variants, where the alternative dependency is *[j, k]*, the values for the parameter VP are the result of *Comb(n,i)* where *Comb* is the combinatorial function of *i* values taken from *n* values, with *i = j..k.*

With this information, the table of parameters is built as shown in Table 2.

| | features | | | |
|---|---|---|---|---|
| variants | type | dice | opponent | players |
| | ludo | dice | person | 2 |
| | chess | no | computer | 2, moreThan2 |
| | trivial | | | |
| | checkers | | | |

**Table 2 – Parameters for pair-wise**

The next step is to build the tables of pairs between the parameters shown in Table 3.

| type-dice | type-opponent | type-players | dice-opponent | dice-players | opponent-players |
|---|---|---|---|---|---|
| ludo-dice | ludo-person | ludo-2 | dice-person | dice-2 | person-2 |
| chess-dice | ludo-computer | ludo-2,more2 | dice-computer | dice-2,more2 | person-2,more2 |
| trivial-dice | chess-person | chess-2 | no-person | no-2 | computer-2 |
| checkers-dice | chess-computer | chess-2,more2 | no-computer | no-2,more2 | computer-2,more2 |
| chess-no | trivial-person | trivial-2 | | | |
| checkers-no | trivial-computer | trivial-2,more2 | | | |
| ludo-no | checkers-person | checkers-2 | | | |
| trivial-no | checkers-computer | checkers-2,more2 | | | |

**Table 3 – Pairs between parameters**

The OVM model also states the relationship between variation points or variants belonging to different variation points. The relationship can be:

- **Variant requires variant (requires_V_V):** The selection of one variant v1 in the variation point VP1 requires the selection of another variant vk in the variation point VPk, without taking into account the variants associated. The rule is:

For each pair *(v1, vj)*, where *vj* is different from *vk*, the value *vk* is added to the pair, thus getting *(v1,vj,vk).*

- **Variant excludes variant (excludes_V_V):** The selection of one variant v1 in the variation point VP1 excludes the selection of another variant vk in the variation point VPk, without taking into account the variants associated. The rule is:

  | The *(v1, vk)* pair is deleted from the corresponding pairs table. |
  |---|

  In the example in Figure 2, the *Chess* variant excludes the *MoreThan2* variant (the same occurs with the *Checkers* variant). Thus, the pairs *(Chess-2,More2)* and *(Checkers-2,More2)* are deleted from the *(Type-Players)* pair table.
- **Variant requires Variation Point (requires_V_VP):** The selection of one variant v1 in the variation point VP1 requires the consideration of a variation point VPk. The rule is:

  | If the variation point *VPk* is optional, the value "no" was added as value for the parameter VPk. The *(v1, no)* pair is deleted from the pairs between VP1 and VPk |
  |---|

  In the example in Figure 2, the *Ludo* variant requires *Dice* (the same occurs with the Trivial variant). The pairs *(Trivial,no)* and *(Ludo, no)* are deleted from the pairs between type and dice.
- **Variant excludes Variation Point (excludes_V_VP):** The selection of one variant v1 in the variation point VP1 excludes the consideration of variation point VPk. The rule is:

  | If the variation point VPk is optional, the value "no" was added as value for the parameter VPk. All pairs between (v1, vk) are deleted from the pairs between VP1 and VPk except the pair (v1, no) |
  |---|

  In the example of Figure 2, the *Chess* and *Checkers* variants exclude *Dice*: thus, *(Chess, dice)* and *(Checkers, dice)* are deleted from the pairs between type and dice.
- **Variation Point requires Variation Point (requires_VP_VP):** The selection of one variation point VP1 requires the consideration of variation point VPk. The rule is:

  | If the variation point VPk is optional, the value "no" was added as value for the parameter VPk. The pair (vi, no) is deleted from the pairs between VP1 and VPk where vi represents all values of VP1 |
  |---|
- **Variation Point excludes Variation Point (excludes_VP_VP):** The selection of one variation point VP1 excludes the consideration of variation point VPk. The rule is:

  | If the variation point VPk is optional, the value "no" was added as value for the parameter VPk. All pairs between (vi, vk) are deleted from the pairs between VP1 and VPk except the pair (v1, no), |
  |---|

Table 4 shows the resulting pairs between the parameter values, excluding the relationships between features.

Once the pairs table is built, the AETG algorithm must be modified to remove the undesired pairs from the final products.

| type-dice | type-opponent | type-players | dice-opponent | dice-players | opponent-players |
|---|---|---|---|---|---|
| ludo-dice | ludo-person | ludo-2 | dice-person | dice-2 | person-2 |
| trivial-dice | ludo-computer | ludo-2,more2 | dice-computer | dice-2,more2 | person-2,more2 |
| chess-no | chess-person | chess-2 | no-person | no-2 | computer-2 |
| checkers-no | chess-computer | trivial-2 | no-computer | no-2,more2 | computer-2,more2 |
| | trivial-person | trivial-2,more2 | | | |
| | trivial-computer | checkers-2 | | | |
| | checkers-person | | | | |
| | checkers-computer | | | | |

**Table 4 - Pairs between parameters excluding relationships between features**

### 4.2 Modifications to the AETG algorithm

The next step is to calculate the test cases using pairwise. Achieving pairwise coverage requires each pair to be covered by at least one test case. The AETG heuristic algorithm must be adapted to consider feature dependencies.

AETG selects the value for each parameter that appears in most unvisited pairs. The problem in this case is that, after removing the undesired pairs, not all pairs are present in the final set of pairs. Therefore, the algorithm must find the value in each parameter that appears in most unvisited pairs, but taking into account that the pairs between the selected values exist. Considering, for example, the pairs in Table 4, the execution of the original AETG algorithm selects *{ludo, dice, person, 2}* as first test case. The second test case selected will be *{trivial, no dice, computer, 2-moreThan2}*; however, the *(trivial, no dice)* pair is not present in the set of pairs. The original AETG algorithm (Figure 4) is improved in step 3: instead of leaving "the pair selected appears in the greatest number of new pairs", adding "and the pair exists in the pairs set" is required.

The stop condition for the algorithm also must be changed. The original AETG algorithm stops when all pairs in the pairs set have been visited. In our case, pairs may exist that are unreached. This is the case for the pair *(no dice, 2-moreThan2)*, which is never visited because is not possible to find a combination of feature values where this pair is valid. Then, this pair remains unvisited at the end of the algorithm. The stop condition is changed and the algorithm stops when the test case selected does not visit any unvisited pair. We have called the AETG algorithm with these improvements *Customizable AETG*.

### 4.3 Implementation of a *Customizable AETG* algorithm

Previously, a framework for combinatorial testing called Combinatorial Testing for Software Product Lines (CTSPL) was implemented as a web application[1]. Any of the testing strategies supported by the framework can be resumed as an algorithm which takes a set of sets as input ($S=\{S_1, S_2, \ldots, S_n\}$, which correspond to the parameters or variation points) and produces a set of combinations of the elements in the sets (which

[1] http://161.67.140.42/CombTestWeb/

correspond to the parameter values, or products in the SPL context). Thus, the algorithm implementing each strategy can be seen as a specialization of an abstract *Algorithm* (Figure 5), which builds its corresponding collection of elements by means of an abstract operation (*buildCombinations*), which is implemented in each specialization.

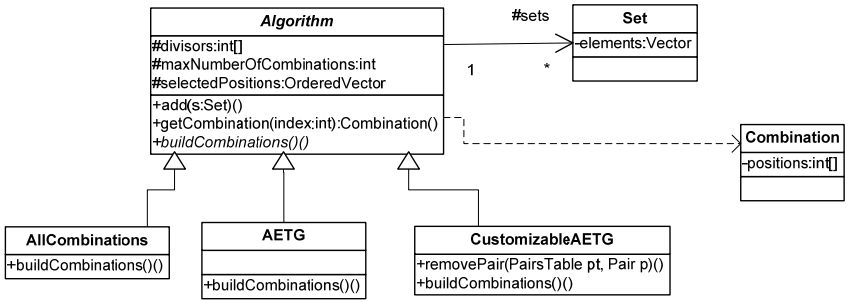


**Figure 5. Partial view of the hierarchical structure of Customizable AETG**

As seen in the figure, each algorithm holds a collection of sets, which represent the parameters. Moreover, each algorithm has a collection of integers (*selectedPositions*), which hold the positions of the selected combinations.

1. Build *pairTables* for *S*, the set of parameters ( *pairTables* does not includes the unrequired pairs).
2. let *c*=combination #0
3. Add *c* to the *selected* set
4. Update *pairTables* with the pairs visited by *c*
5. while there are unvisited pairs in *pairTables and continue*
    1. initialize *c* putting the value which visits more unvisited pairs in *pairTables*
    2. complete *c* with the values of the remaining sets **in such way most pairs are jointly visited and the pairs selected exists in *pairTables***
    3. *if c covers some unvisited pair*
        3.1 Add *c* to the *selected* set
        3.2 Update *pairTables* with the pairs visited by *c*

    *else continue* := false

**Figure 6. Pseudocode of the Customizable AETG algorithm**

Each *Combination* keeps an array of as many integers as there are sets in its *positions* field. Each integer in *positions* represents the index of the selected element

from the corresponding set. Given a combination, the *algorithm* extracts the parameter values by visiting its collection of *sets*.

Figure 6 shows a pseudocode of this new version of AETG. Note the changes introduced in the stop condition (step 5) and in the selection of values (step 5.2).

### 4.3 Description of the web application

The web application accepts the description of the elements in the sets (sets are distributed in columns; their elements in rows) and allows the application of any of the implemented combination algorithms. Moreover, the application also accepts xmi files representing the feature model of the SPL. In Figure 7, the user has selected and is ready to submit the xmi file corresponding to the feature model of the Board Games SPL.

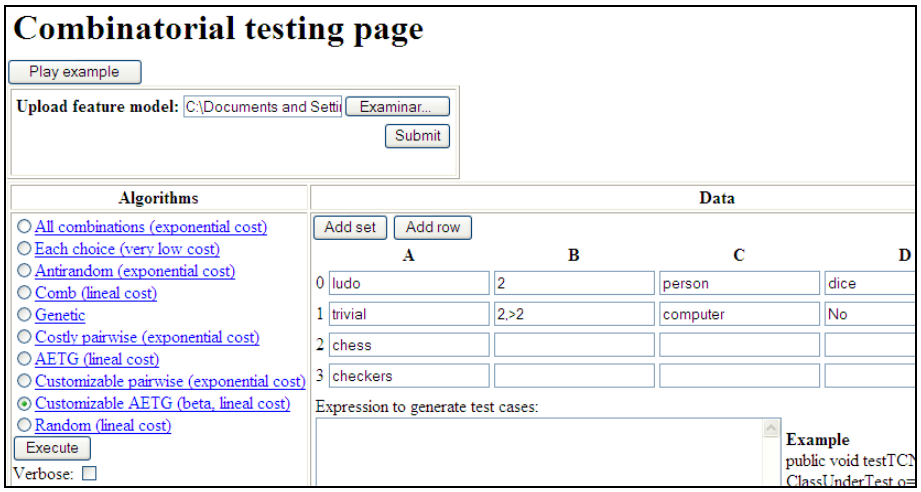


**Figure 7. Uploading the feature model shown in Figure 3**

Once the application has received the feature model with the xmi file, it analyzes it and shows the pairs tables (Figure 8) leaving the user to select those that should not be included in the final suite. At this time, we are modifying the code of the subsystem in charge of processing the xmi file to detect, via the relationships defined in the model (excludes and requires), which pairs should be removed.

Then, the user is ready to select any of the provided algorithms (left side of Figure 7) and obtain the results. If s/he selects the Customizable Pair AETG algorithm, the algorithm shows the results.

We will illustrate how the results are reached describing the steps followed by the *Customizable AETG* in Figure 6. The first step in the algorithm is "Build pairTables for S, the set of parameters, the pairTables does not include the restricted pairs", the pairTables is shown in Table 6. At the beginning, the column corresponding to the test case that visits this pair is blank. Table 5 shows the visited pairs in each step of the algorithm. In the first step, the ludo value appears in 5 unvisited pairs (see Table

6). When the combination # 0 ={ludo, dice, person,2}, is selected, Table 6 is updated and for step 2, the ludo value appears now in 2 unvisited pairs.

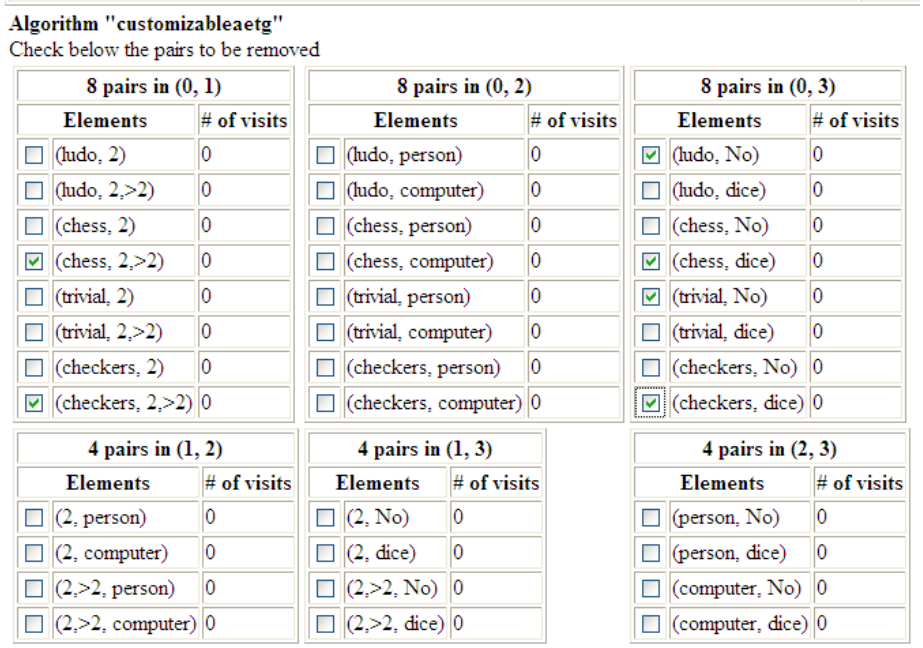


**Figure 8. The user selects the pairs to be removed**

| | Visited pairs by value | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Step | ludo | chess | trivial | checkers | dice | no dice | person | computer | 2 | 2- >2 | Test Case |
| 1 | 5 | 4 | 5 | 4 | 6 | 6 | 8 | 8 | 8 | 6 | {ludo, dice, person, 2} |
| 2 | 2 | 4 | 5 | 4 | 3 | 6 | 5 | 8 | 5 | 6 | {trivial,dice,computer,2->2} |
| 3 | 2 | 4 | 2 | 4 | 0 | 6 | 5 | 5 | 5 | 3 | {chess,no dice,person,2} |
| 4 | 2 | 1 | 2 | 4 | 0 | 3 | 3 | 5 | 3 | 3 | {checkers, no dice,computer,2} |
| 5 | 2 | 1 | 2 | 1 | 0 | 1 | 3 | 2 | 1 | 3 | {trivial,dice,person,2->2} |
| 6 | 2 | 1 | 1 | 1 | 0 | 1 | 1 | 2 | 1 | 2 | {ludo,dice,computer,2->2} |
| 7 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | {chess,no dice,computer,2} |
| 8 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | {trivial, dice, person, 2} |
| 9 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | {checkers,no dice, person,2} |
| | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | {chess,no dice, person, 2} |

**Table 5 – Visited pairs and test cases in Customizable AETG**

The algorithm selects the value for each parameter that visits the most pairs. In step 2, it first selects computer because this value appears in 8 pairs; the selected test case up to now is {-,-,computer,-}. Then for the rest of the parameters, the algorithm selects the value that visits the most pairs. The first parameter selected is *Type* and the value trivial is selected because it appears in 5 pairs. The test case is now {trivial,-,computer,-}. For the parameter *Dice*, the value no dice appears 6 times, but the pair (trivial, no dice) does not exist in pairsTable, so the value dice is selected. The test case is {trivial, dice,computer,-}. For the parameter player, value 2, moreThan2 appears 6 times and is selected. The test case is {trivial, dice, computer, 2-

MoreThan2}. Once the test case is selected, Table 6 is updated with the visited pairs for the test case.

The algorithm continues 9 more steps and the test cases selected are shown in Table 5. In the last step, only one pair is unvisited, this pair is (no dice, 2-MoreThan2). This pair is unreacheable by a combination of pairs, so in step 10 the algorithm selects {chess,no dice, person, 2}. Due to the fact that this pair does not visit any unvisited pair, the algorithm stops.

| type-dice | test case | type-opponent | test case | type-players | test case | dice-opponent | test case | dice-players | test case | opponent-players | test case |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ludo-dice | 1,6 | ludo-person | 1 | ludo-2 | 1 | dice-person | 1,5,8 | dice-2 | 1,8 | person-2 | 1,3,8,9 |
| trivial-dice | 2,5,8 | ludo-computer | 6 | ludo-2,more2 | 6 | dice-computer | 2,6 | dice-2,more2 | 2,5,6 | person-2,more2 | 5 |
| chess-no | 3,7 | chess-person | 3 | chess-2 | 3,7 | no-person | 3,9 | no-2 | 3,4,7, 9 | computer-2 | 4,7 |
| checkers-no | 4,9 | chess-computer | 7 | trivial-2 | 8 | no-computer | 4,7 | no-2,more2 | | computer-2,more2 | 2,6 |
| | | trivial-person | 5,8 | trivial-2,more2 | 2,5 | | | | | | |
| | | trivial-computer | 2 | checkers-2 | 4,9 | | | | | | |
| | | checkers-person | 9 | | | | | | | | |
| | | checkers-computer | 4 | | | | | | | | |

**Table 6 – Test cases that visit each pair in Customizable AETG**

Using the CustomizedAETG algorithm the test cases obtained are shown in Table 5, this mean that the test engineer must test the followings products in the line (where CF refers the set of common features to all the products in the line):

Product 1 = CF U {ludo, dice, person, 2}
Product 2 = CF U {trivial, dice, computer, 2, MoreThan2}
Product 3 = CF U {chess, person,2}
Product 4 = CF U {checkers, computer,2}
Product 5 = CF U {trivial, dice, person, 2, MoreThan2}
Product 6 = CF U {ludo, dice, computer, 2, MoreThan2}
Product 7 = CF U {chess, computer, 2}
Product 8 = CF U {trivial, dice, person, 2}
Product 9 = CF U {checkers, person, 2 }

## 4 Conclusions

This paper describes the application of combinatorial testing to the context of Software Product Lines. Products proceeding from a SPL consist of different types of combinations of the variants and variation points composing the line. Since exhaustive testing is not viable and, furthermore, many of the possible combinations will not belong to any of the final products, several authors have also approached combinatorial testing strategies for SPL testing, especially applying pairwise coverage. However, even some combinations proceeding from this kind of coverage criterion will not be present in any product (in the Board Games example, neither chess nor checkers will match with more than two players). Thus, the AETG algorithm for pairwise coverage has been modified to remove the unfeasible products from the final suite.

The modified version of the algorithm has been included on a web page, which furthermore makes it possible to upload a feature model described in xmi. The tool loads the variants and variation points and is capable of applying a variety of algorithms. In current SPL practice, there are pairs of combinations which the tester is

more interested in testing. Therefore, we are also improving the algorithm to give weight to each pair, in order to more exhaustively test the most important pairs.

**Acknowledgments.** This research was financed by the projects: PRALIN (PAC08-0121-1374) and MECCA (PII2I09-00758394) from the "Consejería de Ciencia y Tecnología, JCCM" and the project PEGASO/MAGO (TIN2009-13718-C02-01) from MICINN and FEDER. Beatriz Pérez has a grant from JCCM Orden de 13-11-2008.

## 5 References

1. Clements, P., Northrop L.: Software Product Lines - Practices and Patterns. Addison Wesley, Boston (2001).
2. Perez Lamancha, B., Polo, M., Piattini, M.: An automated model-driven testing framework for Model-Driven Development and Software Product Lines. In 5th Inter. Conference on Evaluation of Novel Approaches to Software Engineering. To be published (2010).
3. Pohl, K., Böckle, G., Van Der Linden F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005).
4. Cohen, D.M., et al., The combinatorial design approach to automatic test generation. IEEE Transactions on Software Engineering, 13(5): p. 83-89 (1996).
5. Bryce, R., Lei,Y., Kuhn, D., Kacker, R.: Combinatorial testing, in Software Engineering and Productivity Technologies, p. 196-208. (2010).
6. Griss, M. Implementing product-line features by composing component aspects. in Software Product Line Conference. p. 222-228 (2000).
7. Kang, K., Cohen S., Hess, J., Novak, W., Spencer, A.: Feature-oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21, (1990).
8. Kang, K., Kim, S., Lee, J., Kim, K., Kim G., Shin E.: FORM: A feature oriented reuse method with domain specific reference architectures. Annals of Software Engineering, 5(1): p. 143-168 (1998).
9. Griss, M., J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. Fifth International Conference on Software Reuse, p. 76(1998).
10. Pérez Lamancha, B., Polo Usaola, M., Piattini, M.: Towards an Automated Testing Framework to Manage Variability Using the UML Testing Profile. in 4th International Workshop on Automation of Software Test.p. 10-17 (2009).
11. Benavides, F., Ruiz-Cortés A.: Feature Model to Orthogonal Variability Model Transformations. A First Step. Actas de los Talleres de las Jornadas de Ing. del Software y BBDD, 3(2) p. 81-90 (2009).
12. Grindal, M., J. Offutt, and S. Andler, Combination testing strategies: A survey. Software Testing Verification and Reliability, 15(3): p. 167-200 (2005).
13. Mandl, R., Orthogonal Latin squares: an application of experiment design to compiler testing. Communications of the ACM, 28(10): p. 1058 (1985).
14. Williams, A. Determination of test configurations for pair-wise interaction coverage. 13th International Conference on Testing Communicating Systems. pp. 59-74( 2000).
15. Perrouin, G., et al. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. Third International Conference on Software Testing, Verification and Validation. p.10-17 (2010).
16. McGregor, J.D., Testing a Software Product Line. Carnegie Mellon University, Software Engineering Institute. Technical report, (2001).
17. Cohen, M., Dwyer, M., Shi, J.:Coverage and adequacy in software product line testing. ISSTA workshop on Role of software architecture for testing and analysis, p. 53-63(2006).
18. Thum, T., Batory, D., Kastner, C.: Reasoning about edits to feature models. 31st International Conference on Software Engineering (2009).