The Journal of Systems and Software 99 (2015) 1–19

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

PROW: A Pairwise algorithm with constRaints, Order and Weight

Beatriz Pérez Lamancha [a,*], Macario Polo [b], Mario Piattini [b]

[a] Software Testing Centre, Republic University, Montevideo, Uruguay
[b] Alarcos Research Group, Castilla-La Mancha University, Ciudad Real, Spain

ARTICLE INFO

Article history:
Received 28 November 2012
Received in revised form 23 July 2014
Accepted 2 August 2014
Available online 16 September 2014

Keywords:
Software testing
Combinatorial testing

ABSTRACT

Testing systems with many variables and/or values is often quite expensive due to the huge number of possible combinations to be tested. There are several criteria available to combine test data and produce scalable test suites. One of them is pairwise. With the pairwise criterion, each pair of values of any two parameters is included in at least one test case. Although this is a widely-used coverage criterion, two main characteristics improve considerably pairwise: constraints handling and prioritisation.

This paper presents an algorithm and a tool. The algorithm (called PROW: Pairwise with constRaints, Order and Weight) handles constraints and prioritisation for pairwise coverage. The tool called CTWeb adds functionalities to execute PROW in different contexts, one of them is product sampling in Software Product Lines via importing feature models. Software Product Line (SPL) development is a recent paradigm, where a family of software systems is constructed by means of the reuse of a set of common functionalities and some variable functionalities. An essential artefact of a SPL is the feature model, which shows the features offered by the product line, jointly with the relationships (includes and excludes) among them. Pairwise testing could be used to obtain the product sampling to test in a SPL, using features as pairwise parameters. In this context, the constraint handling becomes essential. As a difference with respect to other tools, CTWeb does not require SAT solvers.

This paper describes the PROW algorithm, also analysing its complexity and efficiency. The CTWeb tool is presented, including two examples of the PROW application to two real environments: the first corresponds to the migration of the subsystem of transactions processing of a credit card management system from AS400 to Oracle with .NET; the second applies both the algorithm and the tool to a SPL that monitors and controls some parameters of the load in trucks.

1. Introduction

Software development suffers from the impossibility of exhaustive testing. Thus, researchers strive to find a balance between test suite size and coverage (e.g. the ability to find faults in the system under test). Different test generation techniques produce different test suites aimed at reaching a certain test data coverage criterion (i.e. using each test datum at least once, using all pairs of values, etc.).

Combination strategies (also called Combinatorial Interaction Testing, CIT) are a type of test case selection method where test cases are created by the combination of "*interesting values*", which have been previously identified by the tester.

The input is a set of parameters, each with some elements (values). The output is a set of combinations, each one composed by one element of each input set (Grindal et al., 2005). For instance, *t-wise* strategies ensure that all combinations of any t parameters (input sets) are included in at least one test case (Grindal et al., 2005). Pairwise is a particular case of t-wise, where $t=2$. In this case, the coverage criterion is that all the pairs between the values of any two parameters are included in at least one test case.

Fig. 1 shows the possible configuration elements of a web application that could be executed with different options for Operating System, Browser, Word processor and DataBase. A test suite that achieve pairwise coverage for this example is shown in Fig. 2, which contains all the possible combinations between pairs of values for Operating System, Browser, Word processor and DataBase. However, this test suite does not have into account the relationship between pairs of values, and can lead to the inclusion of many test cases that contain semantically meaningless combinations of test data. For the configurations in Fig. 2, the cases combining *Linux* with *IExplorer* or *Linux* with *Microsoft Word* (test cases 1, 12, 13 and 16) make no sense. If the unfeasible test cases are directly removed from the suite, then there will be interesting

* Corresponding author.
E-mail addresses: bperez@fing.edu.uy (B. Pérez Lamancha), macario.polo@uclm.es (M. Polo), mario.piattini@uclm.es (M. Piattini).

http://dx.doi.org/10.1016/j.jss.2014.08.005
0164-1212/

Operating System	Browser	Word Processor	DataBase
Linux	Firefox	Microsoft Word	SQL
Windows	IExplorer	Open Office	Oracle
Mac OS	Opera	Feng Office	
	Chrome	Google Docs	

Fig. 1. Configuration parameters.

Test Case	Operating System	Browser	Word Processor	DataBase
1	Linux	Firefox	Microsoft Word	SQL
2	Windows	IExplorer	Open Office	Oracle
3	Mac OS	Opera	Feng Office	SQL
4	Linux	Chrome	Google Docs	Oracle
5	Windows	Firefox	Feng Office	Oracle
6	Mac OS	IExplorer	Microsoft Word	SQL
7	Windows	Opera	Google Docs	SQL
8	Mac OS	Chrome	Open Office	Oracle
9	Linux	Opera	Open Office	SQL
10	Windows	Chrome	Microsoft Word	Oracle
11	Mac OS	Firefox	Google Docs	SQL
12	Linux	IExplorer	Feng Office	SQL
13	Linux	Opera	Microsoft Word	Oracle
14	Linux	Chrome	Feng Office	SQL
15	Linux	Firefox	Open Office	SQL
16	Linux	IExplorer	Google Docs	SQL

Fig. 2. Test cases obtained using pairwise criterion.

pairs which will remain untested: if test case 1 (*Linux, Firefox, Microsoft Word*) is removed due to the incompatibility of parameters 1 and 3, then the pair (*Firefox, Microsoft Word*), which is legal and should be tested, would be outside the testable configurations. Then, the further removal of invalid test cases is not an efficient solution, since valid pairs will be also removed from the test suite.

Detecting these combinations in advance requires resorting to the semantics of the involved parameters, i.e. what the parameters stand for. The tools must not focus strictly on providing an algorithmic solution to the mathematical problem of combinatorial testing, also account for other complementary features, which are rather important in order to make these tools really useful in practice, such as the ability to handle constraints on the input domains (Calvagna and Gargantini, 2008; Grindal et al., 2005).

A recent software development paradigm where combination testing is being applied is Software Product Line (SPL). A 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" (Clements and Northrop, 2001). Products in an SPL share a set of characteristics (commonalities) and differ in a number of variation points, which represent the variabilities of the products.

Variability is a central concept in SPL development. It allows for the generation of different products in the line by reusing core assets. Variability is captured through features. A feature is an increment in program functionality that customers use to distinguish one SPL product from another (Kang et al., 1990). A feature can be a specific requirement, a selection among optional or alternative requirements, or can be related to certain product implementation or characteristics (Griss, 2000). Several documented examples of SPL exists,[1] for example the Nokia SPL where several mobile phones share a set of common characteristics: in this case a feature can be the language, and the feature variants can be: Spanish, French, Chinese, among others. Feature models are used to relate features to each other in various ways, showing sub-features, alternative features, optional features, dependent features or conflicting features (Griss, 2000).

One of the main problems in SPL testing is the selection of the products to test. From a feature model, several products can be built, and the possible combinations of features to obtain products can make it impossible to test all of them. One possible solution is obtaining a Product sampling set to SPL testing. To this end, combinatorial testing strategies can be used, taking the features as parameters and the feature variants as parameters values. More information can be obtained from a feature model: we also would get all the possible relationships between features (includes, excludes) and be able to represent this with the combinatorial technique. These relationships between features restricts the possible combination of products, for example, if a feature "excludes" another feature, then all the products that combines the variants of these features are meaningless, the relationships in a feature model can be handled automatically using constraint handling. Due the relationships in a feature model defines the products that can be generated in the SPL, the functionality of constraints handling that could be desirable in combinatorial testing algorithms in SPL testing becomes essential.

The contribution of this article is twofold: first, it presents an algorithm called PROW (Pairwise with constRaints, Order and Weight) for generating pairwise test suites powered by domain semantics and knowledge, which are captured through:

- Constraints: The algorithm makes it possible to exclude all the pairs between parameter values that are semantically meaningless.
- Weight assignment: The algorithm is also capable of including those pairs which require more frequent testing, assigning a weight to each pair between parameter values, this value is referred in this paper as *"pair weight"*. With this, the tester can specify the most important pairs from a testing point of view.
- Ordering for test case prioritisation: The test suite generated by the algorithm is ordered using the summation of the *"pair weights"* involved in the test case. The ordered test suite can be used to test planning, since the most important test cases must be tested early. This is especially useful in regression testing, when the time for testing is not sufficient to execute the complete test suite.

The PROW algorithm has been implemented in a publicly available web tool, under a GNU license called CTWeb. This is the second contribution: CTWeb tool provides features that allow the use of PROW efficiently and also for product sampling in SPL. Some of the functionalities that provide for CTWeb to improve PROW include:

- Product sampling: Allows to load a feature model and automatically processes the features and their relationships to obtain the parameters to execute PROW. The result is the set of products to be tested in the SPL.
- Setting the base test suite: Test cases obtained with combinatorial testing may not exactly be the most widely used in reality, even if weight is assigned to prioritise them. This functionality allows the tester to define a set of test cases as the basis on which to run PROW. This base test suite is uploaded in a tab-separated file and CTWeb mark all pairs covered by the base test suite as visited prior to running the algorithm PROW. This functionality can be used in both, combinatorial testing and product sampling. The result is a set of test cases that satisfies pairwise coverage and contains the base test suite as a subset.

[1] Product Line Hall of Fame: http://splc.net/fame.html.

- Uploading parameters file: For combinatorial testing, CTWeb allows to upload a tab-separated file, where the tester specifies the parameters, its values and the constraints and associated weight between pairs. This file is the input to PROW to obtain the test cases for combinatorial testing.

After an overview of combination testing strategies in Section 2, the PROW algorithm is detailed and presented in Section 3. Section 3.4 presents the order analysis and the evaluation of the PROW algorithm, showing the smooth degradation of PROW along the number of parameters and parameter values, and the efficiency gains compared with other approaches. Section 4 includes a description of CTWeb and the functionality that allows both product sampling in SPL and powering of combinatorial testing. Two case studies are presented, first in Section 5, a case study where combinatorial test data was used to test a banking system in a testing consultancy and then in Section 6, a sensor monitoring SPL is used as a case study to obtain product sampling. Section 7 discusses some works related to constraint handling and test case prioritisation in combinatorial testing and product sampling in SPL. Finally, conclusions and future work are presented.

2. Combination testing strategies

The completeness of the coverage in Combinatorial Interaction Testing (CIT) is up to the designer, who determines how the parameter values are combined and used: *Each-use* coverage (also known as *1-wise*) indicates that each test value is included in at least one test case in the test suite.

Pairwise coverage (also known as *2-wise*) requires that every possible pair of values of any two parameters be included in some test case. *t-wise* is a generalisation of pairwise where every possible combination of interesting values of t parameters be at least included in one test case (Grindal et al., 2005).

Pairwise relies on the system's proneness to exhibit faults with the interactions of concrete pairs of values, offering a good relation between the test suite size and its capability for revealing faults. Due to the cost of executing and re-executing test suites, it is important to keep the test suite size within reasonable limits. Since the problem of generating minimum pairwise test sets is NP-complete (Lei and Tai, 1998), different researchers have developed strategies to generate near-minimum pairwise test sets. Different test generation strategies have been published for pairwise testing (see Grindal et al., 2005; Cohen et al., 2007; Nie and Leung, 2011).

Table 1 lists the most important CIT algorithms in the literature. For the *category* column, we use the taxonomy from Cohen et al. (2007), who classify CIT in: *Algebraic* (mathematical techniques are used, these constructions are fast and produce small covering arrays efficiently, although they are not general), *Greedy* (constructs a set of tests where each test covers as many uncovered combinations as possible) and *Heuristic search* (which start from a preexisting test set and then applies a series of transformations to the set until it covers all the combinations) (Nie and Leung, 2011).

For each algorithm, Table 1 shows: its *name*, the bibliographic *reference*, the *category*, *algorithm availability* (indicates whether the pseudocode was published or not), *test size table* (most of the algorithms include a test size table comparing the test size obtained against other algorithms), *time table* (indicates whether the algorithm includes a table with the execution time for different test inputs), *constraints* (indicates whether the algorithm considers the treatment of constraints between parameters), *prioritisation* (indicates whether the algorithm considers priorities between parameter values, *strength* (refers to the strength of the algorithm, i.e., if it resolves pairwise or t-wise coverage), *license* (if the algorithm is included in a tool developed by the authors, this column indicates the license type for its distribution) and *platform* (indicates the execution platform for the tool). Last row in Table 1 shows the data for the PROW algorithm presented in this paper. The table also shows that greedy algorithms are the most popular strategy for CIT.

There are two classes of greedy methods: the first is one-row-at-a-time, based on the Automatic Efficient Test Case Generator (AETG). Algorithm 1 shows its pseudocode. Basically, each iteration starts with an empty combination, which is progressively completed with the inclusion of the value which appears in the most unvisited pairs. If the selected test value proceeds from the ith parameter, it is placed at the ith position of the combination, which in this way stays partially filled. The remaining values are selected taking into account both (1) their compatibility with the previously selected values and (2) the number of unvisited pairs they visit. As an example, consider testing the web application whose configuration parameters are shown in Fig. 1. The tester wishes to test the system behaviour depending on the operating system, browser, word processor and database (see Fig. 1). AETG produces the 16 test cases appearing in Fig. 2.

The second class of greedy methods is based on the In Parameter Order (IPO) algorithm (Lei and Tai, 1998), which begins by generating all t-sets for the first t factors and then incrementally expands the solution, both horizontally and vertically using heuristics until the array is complete (Nie and Leung, 2011).

Algorithm 1. AETG algorithm (taken from Cohen et al., 1996)

Assume a system with k test parameters and that the ith parameter has l_i 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 $f_1, \ldots, f_k$.
3. Assume that values have been selected for parameters $f_1, \ldots, f_j$. For $1 \leq i \leq j$, let the selected value for f_i be called v_i. Then, choose a value v_{j+1} for f_{j+1} as follows. For each possible value v for f_j, find the number of new pairs in the set of pairs $\{f_{j+1} = v$ and $f_i = v_i, 1 \leq i \leq j\}$. Then, let v_{j+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 j values already chosen for parameters $f_1, \ldots, f_j$.

2.1. The constraint problem

In software testing, some combinations of parameter values are frequently invalid. If the pairwise algorithm does not provide a solution for the constraints between parameters, the tester must manually review the results obtained from pairwise strategies, since there are many situations where the generated test suite contains meaningless pairs.

Nie and Leung (2011) recently presented a survey about combinatorial testing. They argue that the existence of constraints increases the difficulty in applying combinatorial testing due to: (1) Several existing test generation methods cannot deal with constraints, and ignore them, in spite of the fact that ignoring constraints may lead to the generation of test configurations that are invalid, ineffective test planning and wasted testing effort. (2) It is

Table 1
Existing CIT techniques.

Technique	Reference	Category	Algorithm	Size table	Time table	Constraints	Prioritisation	Strength	License	Platform
AETG	Cohen et al. (1997) and Lott et al. (2005)	Greedy	Y	Y	N	Y	N	*t*-wise	Proprietary	Web
PICT	Czerwonka (2006)	Greedy	Y	Y	N	Y	Y	*t*-wise	Freeware	Windows Comm. line
IPO/PairTest	Lei and Tai (1998)	Greedy	Y	Y	Y	N	N	2-wise	Not available	
ATGT	Calvagna and Gargantini (2008)	Algebraic	Y	Y	N	Y	N	2-wise	Freeware	Java App
Simulated Annealing/CASA	Cohen et al. (2003) and Garvin et al. (2009)	Heuristic	Y	Y	Y	Y	N	*t*-wise	Open Source (GNU)	Comm. line
Test Cover/CATS	Sherwood (1994)	Greedy, Algebraic	N	Y	N	Y	N	2-wise	Proprietary	Web
DDA	Bryce and Colbourn (2006, 2007, 2009)	Greedy	Y	Y	Y	Y	Y	*t*-wise	Not available	
CTS	Hartman (2005)	Algebraic	Y	Y	Y	Y	N	*t*-wise	Has been retired by IBM	
TConfig	Williams (2002)	Algebraic	Y	Y	Y	N	N	*t*-wise	Freeware	Java App
TCG	Tung and Aldiwan (2000)	Greedy	Y	N	N	Y	N	2-wise	Freeware	Windows
EXACT	Yan and Zhang (2008)	Algebraic	Y	Y	Y	N	N	*t*-wise	Freeware	Linux
IPOG/ACTS/FireEye	Lei et al. (2008)	Greedy	Y	N	Y	Y	N	*t*-wise	Freeware	Java App
PROW		**Greedy**	**Y**	**Y**	**Y**	**Y**	**Y**	**2-wise**	**Open Source (GNU)**	**Web**

difficult to design a general algorithm for test generation with due consideration of constraints. (3) Even a small number of constraints may give rise to an enormous number of invalid configurations. When the generated test suite contains many invalid test cases, this will cause a loss of combination coverage. (4) Complicated constraints may exist in the SUT, and multiple constraints can interact to produce additional implicit constraints. It is both time consuming and highly error prone to manually deal with constraints in test suite generation (Nie and Leung, 2011).

Cohen et al. (2007) discuss that managing constraints in existing algorithms for CIT exhibits at least one of the following limitations: it ignores the constraints altogether, requiring the user to explicitly define all illegal configurations; it attempts to bias test generation to avoid constraints "if possible", but does not guarantee the avoidance of illegal configurations; it mentions constraints as a straightforward engineering extension to be solved later; or it uses a proprietary (unpublished) method that cannot be reproduced by the research community.

It is noteworthy that constraints handling brings about a change in the number of test cases obtained for pairwise coverage. Depending on the restrictions that are defined, more or less test cases may be obtained, comparing with those obtained without taking into account constraints. For the configuration example, if the pairs *(Firefox, Microsoft Word)* and *(Firefox, SQL)* are not possible, then the first test case in Fig. 2 must be changed. This test case visits 6 pairs, 2 of which were deleted. The remaining 4 pairs cannot be covered with a single test case, since the pairs *(Linux, Firefox)* and *(Linux, SQL)* can not be in the same test case because (Firefox, SQL) has been eliminated, then we need two test cases instead of one.

Thus, proper handling of constraints is a key issue in combinatorial testing. We solve this problem with a pairwise algorithm called PROW, a greedy algorithm that supplements Fig. 1 with information about constrained combinations which should not be included in the test suite (e.g. combining Linux with IExplorer or Linux with Microsoft Word), so that the algorithm skips these combinations. A more detailed analysis of the existent CT methods and tools is presented in Section 7.

3. The PROW algorithm: Pairwise coverage with constRaints, Order and Weight

PROW algorithm (Pairwise with constRaints, Order and Weight) is a greedy heuristic, building one-test-a-time that locally optimises the solution. PROW is a greedy algorithm partially based on AETG, empowered with domain semantics. Specifically, parameter constraints and weights for popular configurations are introduced to avoid undesired pairs and handle repetitive value testing, respectively. Another significant difference is that the PROW solution is deterministic, i.e., two executions with the same input return the same output. Obviously, this allows the regeneration of the same test cases later on, but is specially interesting to generate more or less similar test suites when constraints, weights or new parameters or features are added or changed. So, the differences between PROW and AETG are enough significant: besides requiring additional inputs (namely constraints and weights), the method for selecting the next test case and the stop condition are also different, since it takes into account values incompatibilities and the importance of each new test case through its total weight.

In general, PROW strives to overcome some of the shortcomings in pairwise approaches, namely:

- inter-parameter constraints are not considered. This can lead to unrealistic value combinations being checked (e.g. *Linux* with *IExplorer*).
- repetitive value testing. In the example shown in Fig. 2, *Linux* is tested in 8 test cases while *Windows* and *Mac* are tested 4 times each. Since this repetition is inevitable, it is preferable to select those values that account for the most popular settings: e.g. if the most common configurations will be based on *Windows* and *Chrome*, then the tester will be interested in including this pair in more test cases (if s/he can select one of two pairs with the same number of visits, probably s/he will prefer to include (*Windows, Chrome*) instead of *(Linux, Opera)*).

This section describes preconditions, pseudocode and post-conditions for PROW algorithm, using as running example the configurations in Fig. 1.

3.1. Preconditions

PROW takes into account the elimination of pairs and the weight assigned to each pair: before executing the algorithm, the undesired pairs are deleted and the pair weights assigned. These preconditions are described in Algorithm 3.

Algorithm 2. PROW: auxiliary functions

Let S be the set of parameters, $S = \{S_1 \ldots S_n\}$. Each parameter S_i is an ordered set of values, with $k \geq 1$ variable for each parameter S_i.

Let $P = \{S_i \times S_j | i = 1 \ldots n-1, j = i+1 \ldots n\}$ the Cartesian product of S, representing all the pairs between parameters values. Let $P' \subseteq P$ the result of deleting the undesired pairs in P.

The followings functions are used in the algorithm:

- The weight of each pair is defined by a function $weight : P' \rightarrow \mathbb{R}$, $weight(p) = w$, $p \in P'$ and $w \in \mathbb{R}$, w represents the weight assigned to the pair p in P'.
- The number of times that a pair is visited by the algorithm is defined by the function $visits : P' \rightarrow \mathbb{N} | visits(p) = n$. At the beginning of the algorithm $visits(p) = 0, \forall \in P'$.
- The number of pairs in that each value for each parameter appears in P' is represented by the *remains* function: $S \times \mathbb{N} \rightarrow \mathbb{N}$. At the beginning of the algorithm $remains(S_i, v_t) = \sum_{j=1}^{n} \left(\sum_{x=1}^{\#S_j} r_t \right)$, where $r_t = 1$ when $\{\exists p \in P' | p \in S_i \times S_j, p = (v_t, v_x)$ or $p = (v_x, v_t)\}$, else $r_t = 0$. *remains* represents, for each value in each parameter, the number of pairs that the algorithm has not covered yet. This function has into account the deleted pairs.
- TS represents the test suite, $TS = \{TS_1 \ldots TS_m\}$. The weight of the test case TS_i can then be calculated as the summation of the weight of the pairs covered by the test case, the function is: $weight_{testcase} : TS \rightarrow \mathbb{R} \mid weight_{testcase}(TS_i) = \sum_{t=1}^{n-1} \sum_{j=t+1}^{j=n} weight(s_{it}, s_{ij})$.
- The amount of pairs visited by a test case $t = t_1 \ldots t_n$, that were not visited by previous test cases. $Pairs_{visited}(t) = \sum_{i=1}^{n-1} \sum_{j=2}^{n} visited_{ij}$, where $visited_{ij} = 1$ if $visits(ti, tj) = 0$ and $visited_{ij} = 0$ if $visits(ti, tj) > 0$.

Fig. 3 indicates the preconditions for the sample problem in Fig. 1. Constraints between pairs are marked as removable and prioritisation between pairs is set in the "*Sel. factor*" column, which represents the weight assigned. The weights can be set by hand or be given in the descriptive file of the system under test.

3.2. Pseudocode

Algorithm 3 shows the pseudocode of PROW. Auxiliary functions used are described in Algorithm 2.

Main concepts involved in PROW are:

1. *The next test value to be included in the next test case*: the decision about whether a combination is included in or excluded from the final test suite considers: (a) that the pairs selected have not been removed, (b) the combination that visits the highest number of unvisited pairs, and (c) if more than one combination visits the highest number of pairs, choose the combination with the highest weight.
2. *The stop condition*: The original AETG algorithm stops when all pairs in the pair set have been visited once. When some pairs are deleted, pairs that are unreachable may exist. In this case, PROW stops when the combination selected does not visit any unvisited pair. If $visits(p) = 0$, then the pair p is unreachable and there is no combination with the remaining parameters that achieves a valid test case using the pair p. If all the pairs are reachable, then PROW stops when there are no more unvisited pairs in P'.

   Let $TS = \{TS_1 \ldots TS_m\}$, test suite obtained using the PROW algorithm where $TS_i = \{s_{i1} \in S_1, \ldots s_{in} \in S_n\}$ represents the ith test case.

   Let $Pairs_i = \{(s_{it}, s_{ir}) | t = 1 \ldots n-1, r = 2 \ldots n\}$, where s_{it} and $s_{ir} \in TS_i$. $Pairs_i$ are the pairs covered by the test case TS_i. At the end of the algorithm the function $visits : P' \rightarrow \mathbb{N} \mid visits(p) = n$, which represents the number of times that the pair p is visited by all the test cases, can be:
   (a) $visits(p) > 0$, then the pair p is visited in at least one test case
   (b) $visits(p) = 0$, then the pair p is unreachable and there is no combination of the remaining parameters that achieves a valid test case using the pair p. In this case, it does not exist a test case $c = \{s_{i1} \in S_1, \ldots s_{in} \in S_n\}$ where $Pairs_{visited}(c) > 0$ and c contains the pair p.
3. *Ordered test suite*: The algorithm orders the test cases according to the summation of the weight of the pairs involved in the test case. It allows testers to know the most valuable test cases to be executed. If the time scheduled for testing tasks is not sufficient to run all the test cases, they can be executed according to this order.

Algorithm 3. PROW algorithm

Preconditions:

Let S be the set of parameters. Each parameter S_i is an ordered set of values, with $k \geq 1$ variable for each parameter S_i.

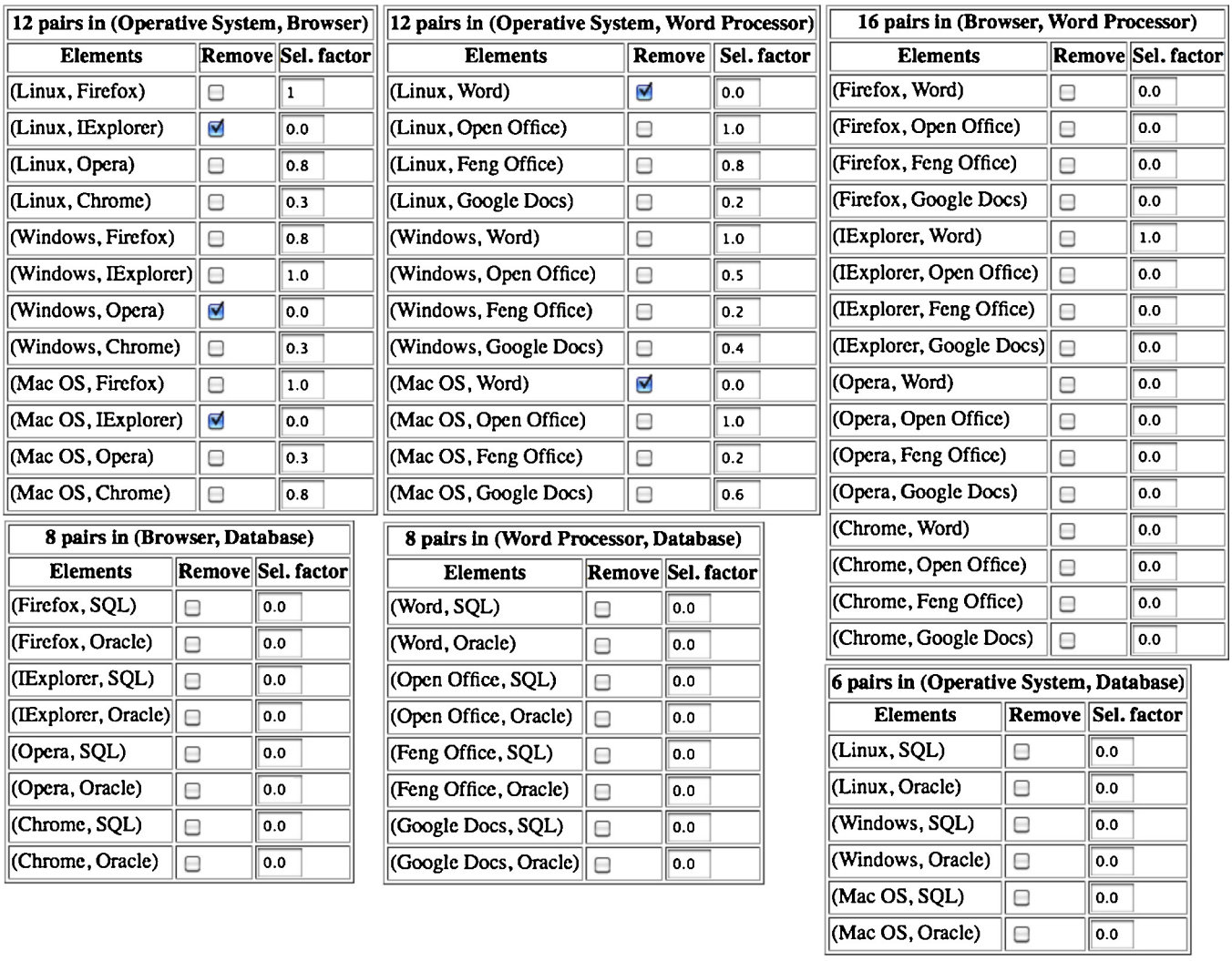
Let $P = \{S_i \times S_j | i = 1 \ldots n-1, j = i+1 \ldots n\}$ the Cartesian product of S, representing all the pairs between parameters values. Let $P' \subseteq P$ the result of deleting the undesired pairs in P.

For each pair $p \in P'$, the weight is assigned by the tester, being zero the weight by defect.

Input: S,P'

Steps:

1. $TS = \{\}$; TS represents the test suite
2. $c = \{\}$; c represents the next test case to be added to TS
3. *continue* := true
4. While $((\exists p \in P' | visits(p) = 0)$ and *continue*
   (a) Initialise test case c putting the value v_j in the position i, where v_j is the value that appears in more unvisited pairs, i.e., v_j maximises the function: $Max_{i=1}^{n}(Max_{j=1}^{\#S_i}(remains(S_i, v_j)))$.
   (b) For each remaining parameter $S_k = \{v_1 \ldots v_t\}$, $k = 1 \ldots n$, $k \neq i$
      i. $maxPairs := Pairs_{visited}(c)$, maxPairs stores the pairs covered by the test case c up to now and uncovered by previous test cases.
      ii. $maxWeight := weight_{testcase}(c)$, stores the summation of the weight of the pairs covered by the test case c up to now.
      iii. $maxRemains := 0$, stores the value that appears in more unvisited pairs
      iv. For each value $v_i \in S_k$, let c_i be the combination c with the value v_i assigned for the parameter k, c_i is candidate to be c if meets:
         A. $\exists$ pair $p = (v_j, v_i) \in P'$, $\forall$ parameter j assigned previously in c_i
         B. if $(maxPairs < Pairs(c_i))$ then $\{maxPairs = Pairs(c_i)$; $maxRemains := remains(S_k, v_i)$; $c_{temp} := c_i\}$
         C. else if $((maxPairs = Pairs(c_i))$ and $(maxRemains \leq remains(S_k, v_i)))$ then { if $(maxWeight < weight_{testcase}(c_i))$ then $c_{temp} := c_i\}$
      v. endfor
      vi. $c := c_{temp}\}$



12 pairs in (Operative System, Browser)

Elements	Remove	Sel. factor
(Linux, Firefox)	☐	1
(Linux, IExplorer)	☑	0.0
(Linux, Opera)	☐	0.8
(Linux, Chrome)	☐	0.3
(Windows, Firefox)	☐	0.8
(Windows, IExplorer)	☐	1.0
(Windows, Opera)	☑	0.0
(Windows, Chrome)	☐	0.3
(Mac OS, Firefox)	☐	1.0
(Mac OS, IExplorer)	☑	0.0
(Mac OS, Opera)	☐	0.3
(Mac OS, Chrome)	☐	0.8

12 pairs in (Operative System, Word Processor)

Elements	Remove	Sel. factor
(Linux, Word)	☑	0.0
(Linux, Open Office)	☐	1.0
(Linux, Feng Office)	☐	0.8
(Linux, Google Docs)	☐	0.2
(Windows, Word)	☐	1.0
(Windows, Open Office)	☐	0.5
(Windows, Feng Office)	☐	0.2
(Windows, Google Docs)	☐	0.4
(Mac OS, Word)	☑	0.0
(Mac OS, Open Office)	☐	1.0
(Mac OS, Feng Office)	☐	0.2
(Mac OS, Google Docs)	☐	0.6

16 pairs in (Browser, Word Processor)

Elements	Remove	Sel. factor
(Firefox, Word)	☐	0.0
(Firefox, Open Office)	☐	0.0
(Firefox, Feng Office)	☐	0.0
(Firefox, Google Docs)	☐	0.0
(IExplorer, Word)	☐	1.0
(IExplorer, Open Office)	☐	0.0
(IExplorer, Feng Office)	☐	0.0
(IExplorer, Google Docs)	☐	0.0
(Opera, Word)	☐	0.0
(Opera, Open Office)	☐	0.0
(Opera, Feng Office)	☐	0.0
(Opera, Google Docs)	☐	0.0
(Chrome, Word)	☐	0.0
(Chrome, Open Office)	☐	0.0
(Chrome, Feng Office)	☐	0.0
(Chrome, Google Docs)	☐	0.0

8 pairs in (Browser, Database)

Elements	Remove	Sel. factor
(Firefox, SQL)	☐	0.0
(Firefox, Oracle)	☐	0.0
(IExplorer, SQL)	☐	0.0
(IExplorer, Oracle)	☐	0.0
(Opera, SQL)	☐	0.0
(Opera, Oracle)	☐	0.0
(Chrome, SQL)	☐	0.0
(Chrome, Oracle)	☐	0.0

8 pairs in (Word Processor, Database)

Elements	Remove	Sel. factor
(Word, SQL)	☐	0.0
(Word, Oracle)	☐	0.0
(Open Office, SQL)	☐	0.0
(Open Office, Oracle)	☐	0.0
(Feng Office, SQL)	☐	0.0
(Feng Office, Oracle)	☐	0.0
(Google Docs, SQL)	☐	0.0
(Google Docs, Oracle)	☐	0.0

6 pairs in (Operative System, Database)

Elements	Remove	Sel. factor
(Linux, SQL)	☐	0.0
(Linux, Oracle)	☐	0.0
(Windows, SQL)	☐	0.0
(Windows, Oracle)	☐	0.0
(Mac OS, SQL)	☐	0.0
(Mac OS, Oracle)	☐	0.0

Fig. 3. Pair tables generated using PROW algorithm.

(c) endfor
(d) if $Pairs_{visited}(c) > 0$ then
  i. $TS := TS \cup \{c\}$
  ii. update the number of visits for the pairs covered by c (and returned later by the function *visits*
  iii. update the numbers of pairs that remains uncovered subtracting the pairs covered by each value assigned in the test case c (these values are returned later by the function *remains*
(e) else
  i. continue:= false
5. endWhile

Output: TS

A step-by-step tracking of the PROW algorithm for the sample problem is depicted in Fig. 4. For each parameter value (*O.S.*={*Linux, Windows, Mac*}, . . .), the corresponding cell represents the number of unvisited pairs in which that value appears in the iteration denoted by the first column. This value is returned by the function *remains* in the algorithm pseudocode. The first test case is initialised by setting the value which visits the most unvisited, non-removed pairs in the corresponding position, for example, at the beginning of the algorithm (row 1), *remains*(*OperatingSystem, Linux*) = 8 because Linux appears in eight pairs of values.

The algorithm starts with a test suite TS={} and a test case c = {}. In step 4(a) (corresponding to the first row), *SQL* and *Oracle* appear in eleven pairs, the algorithm selects *SQL* and, then, c={-,-,-,SQL}. Step 4(b) completes the other parameters. For *Operating System*, the selected value is *Windows*, because it is the one with the most unvisited pairs (nine). Now, c={Windows,-,-,SQL}.

For *Browser*, both *Firefox* and *Chrome* visit nine pairs. The algorithm selects *Firefox* because it maximises the total weight, being now c={Windows, Firefox,-,SQL}. The same occurs for *Word Processor*, *Open Office*, *Feng Office* and *Google Docs* which have nine pairs each. The algorithm selects *Open Office* because this value maximises the weight of the combination, which is: *weight(Windows, Firefox) + weight(Windows, Open Office) + weight(Windows, SQL) + weight(Firefox, OpenOffice) + weight(Firefox, SQL) + weight(OpenOffice, SQL) = 0.8 +0.5 + 0 + 0 + 0 + 0 = 1.3. Finally c={Windows, Firefox,Open Office,SQL}*.

Before starting the next iteration, PROW updates the number of unvisited pairs that remain for each value (second row in Fig. 4). The second iteration selects *Oracle* as the most unvisited *value*, includes it in the current test case and completes it with *Linux, Chrome* and *FengOffice*. This test case remains as*(Linux, Chrome, Feng Office, Oracle)*, whose weight is 1.1.

Step	Operating System			Browser				Word Processor				DataBase		Test Case	Weight
	Linux	Windows	Mac	Firefox	IExplorer	Opera	Chrome	Word	Open	Feng	Google	SQL	Oracle		
1	8	9	8	9	7	8	9	7	9	9	9	11	11	{Windows, Firefox, Open, SQL}	1.3
2	8	6	8	6	7	8	9	7	6	9	9	8	11	{Linux, Chrome, Feng, Oracle}	1.1
3	5	6	8	6	7	8	6	7	6	6	9	8	8	{Mac, Opera, Google, SQL}	0.9
4	5	6	5	6	7	5	6	7	6	6	6	5	8	{Windows, IExplorer, Word, Oracle}	2.0
5	5	3	5	6	4	5	6	4	6	6	6	5	5	{Linux, Firefox, Google, Oracle}	1.2
6	3	3	5	3	4	5	6	4	6	6	3	5	3	{Mac, Chrome, Open, SQL}	1.8
7	3	3	3	3	4	5	3	4	4	6	3	4	3	{Windows, IExplorer, Feng, SQL}	1.2
8	3	2	3	3	2	5	3	4	4	3	3	2	3	{Linux, Opera, Open, Oracle}	1.8
9	1	2	3	3	2	2	3	4	1	3	3	2	1	{Windows, Chrome, Word, SQL}	1.3
10	1	1	3	3	2	2	1	2	1	3	3	1	1	{Mac, Firefox, Feng, Oracle}	1.2
11	1	1	0	1	2	2	1	2	1	1	3	1	0	{Windows, IExplorer, Google, SQL}	1.4
12	1	0	0	1	1	2	1	2	1	1	1	1	0	{Linux, Opera, Feng, SQL}	1.6
13	0	0	0	1	1	1	1	2	1	0	1	0	0	{Windows, Firefox, Word, SQL}	1.8
14	0	0	0	0	1	1	1	1	1	0	1	0	0	{Windows, IExplorer, Open, SQL}	1.5
15	0	0	0	0	0	1	1	1	0	0	1	0	0	{Linux, Chrome, Google, SQL}	0.5
16	0	0	0	0	0	1	0	1	0	0	0	0	0		

Fig. 4. PROW algorithm executed step by step.

At this point is important to note that the algorithm must find the value in each parameter that appears in the most unvisited pairs, always taking into account that the pairs between the selected values must exist. In iteration 9 for example,*Microsoft Word* has the most unvisited pairs (4). PROW includes it in the initial combination *(-, -, Microsoft Word, -)*. For *Operating System, Mac* is the value with the greatest value, but it cannot be picked up since the pair *(Mac, Microsoft Word)* was deleted. Therefore, PROW selects Windows and completes the combination with *(Windows, Chrome, Microsoft Word, SQL)*, whose weight is 1.3.

To illustrate the stop condition, consider that in the 16th row, the pair *(Opera, Microsoft Word)* remains unvisited because it is not possible to find a combination of parameter values that contains this pair. A partially built test case is {-, *Opera, Microsoft Word*, -}. The possible values for O.S. are: *Linux, Windows* and *Mac*, but both *Linux* and *Mac* are incompatible with *Microsoft Word.* Thus, only *Windows* could be selected with respect to *Word*. But *Windows* is incompatible with *Opera*. Then, by the self definition of the system (and although it is not explicitly specified), it is impossible to find a combination containing *(Opera, Microsoft Word)*, so remaining this pair unvisited at the end of the algorithm. The algorithm stops when there are no more selectable test cases and all valid pairs have been visited, such as in this example.

3.3. Postconditions

Once the PROW algorithm has been executed, its results fulfil the postconditions described in Algorithm 4. In algorithms without constraints handling (for example in AETG), at the end of the algorithm, each pair was visited at least once. In the case of PROW, due to the constraint handling and the weight, the pairs in P′ can have been visited once, more than once (this is the case for the pairs with more weight assigned by the tester) or zero times (this is the case with unreachable pairs).

Algorithm 4. PROW postconditions

Postconditions:

Let $TS = \{TS_1 \ldots TS_m\}$, test suite obtained using the PROW algorithm where $TS_i = \{s_{i1} \in S_1, \ldots s_{in} \in S_n\}$ represents the ith test case. Each test case TS_i covers $(n^*(n-1)/2)$ pairs.

Let $Pairs_i = \{(s_{it}, s_{ir}) | t = 1 \ldots n-1, r = 2 \ldots n\}$, where s_{it} and $s_{ir} \in TS_i$. $Pairs_i$ are the pairs covered by the test case TS_i. The set of all the pairs covered by the test suite TS is $Pairs = U_{i=1}^{m} Pairs_i$.

The weight of the test case TS_i can then be calculated as the summation of the weight of the pairs covered by the test case, the function is:

$$weight_{testcase} : TS \to \mathbb{R} \mid weight_{testcase}(TS_i) = \sum_{t=1}^{n-1} \sum_{j=t+1}^{j=n} weight(s_{it}, s_{ij}).$$

The test cases resulting are ordered using the $weight_{testcase}$ function. At the end of the algorithm the function $visits : P' \to \mathbb{N} \mid visits(p) = n$, which represents the number of times that the pair p is visited by all the test cases, can be:

1. $visits(p) = 1$, then the pair p is visited once for the PROW algorithm
2. $visits(p) > 1$, then the pair p is visited in more than one test case. Let $TS_i = \{s_{i1} \in S_1, u, \ldots v, s_{in} \in S_n\}$ be one of the test cases where the pair (u, v) appears. If $(x \in S_i, w \in S_j)$ exists, where the u can be substituted by x in S_i and v can be substituted by w in S_i, remaining TS_i as a valid test case, then $weight(u, v) \geq weight(x, w)$.
3. $visits(p) = 0$, then the pair p is unreachable there is no combination of the remaining parameters that achieves a valid test case using the pair p.

Fig. 5 shows the test cases generated by PROW, ordered by their weight. The tester can leave the weight for one or more pairs unassigned; in that case the weight is considered 0.

3.4. PROW evaluation

This section evaluates PROW in the followings aspects: complexity analysis, test size and test execution time.

3.4.1. Complexity analysis

For the complexity analysis, we consider the number of parameters. Each parameter has a number of values which are usually different. To estimate the complexity we use an arbitrarily large input n ($n \to \infty$), taking n as both the number of parameters and the parameter with the most values.

Step 4 is a while loop which builds the test suite. The first step in the while loop is 4(a) where the value that visits the most unvisited pairs is selected as the first value. This search is carried out through the rows of the table in Fig. 4, which has n^2 columns. Thus this step has an order of n^2. Step 4(b) completes test case c. Observe that step 4(b) considers, first, that the pairs in the combination exist;

Test Case	Operating System	Browser	Word Processor	DataBase	Weight
1	Windows	IExplorer	Microsoft Word	Oracle	2.0
2	Mac OS	Chrome	Open Office	SQL	1.8
3	Windows	Firefox	Microsoft Word	SQL	1.8
4	Linux	Opera	Open Office	Oracle	1.8
5	Linux	Opera	Feng Office	SQL	1.6
6	Windows	IExplorer	Open Office	SQL	1.5
7	Windows	IExplorer	Google Docs	SQL	1.4
8	Windows	Chrome	Microsoft Word	SQL	1.3
9	Windows	Firefox	Open Office	SQL	1.3
10	Mac OS	Firefox	Feng Office	Oracle	1.2
11	Windows	IExplorer	Feng Office	SQL	1.2
12	Linux	Firefox	Google Docs	Oracle	1.2
13	Linux	Chrome	Feng Office	Oracle	1.1
14	Mac OS	Opera	Google Docs	SQL	0.9
15	Linux	Chrome	Google Docs	SQL	0.5

Fig. 5. Ordered test suite using PROW algorithm.

second, that the combination visits the most unvisited pairs; and third, that the weight of the combination is maximum. Step 4(b) is the most complex with a complexity order of $O(n^6)$.

The complexity for the entire algorithm depends on the times that the while loop (step 4) is executed. For that, we studied the worst case. The worst case occurs when the algorithm calculates one test case for each unvisited pair. Since the number of pairsTables is $(n^*(n-1)/2)$, and n^2 is the number of pairs in each table, the complexity of the while decision is $n^2 * (n^*(n-1)/2) = (n^4 - n^3)/2$. Thus, in the worst case the while construct must be executed $(n^4 - n^3)/2$ times.

Within the while loop, the step with the most complexity is step 3(b) with $O(n^6)$. The while decision order is $O(n^4 * n^6)$. Then the while loop has $O(n^{10})$ in the worst case. The order of the PROW algorithm is then $O(n^{10})$, a polynomial order.

3.4.2. Test size analysis

For the test size analysis, PROW is compared with existing algorithms for CIT. First we compare **PROW without constraints** (i.e., without excluding any pair or assigning weights). PROW was created to handle constraints, but the comparison helps to see that PROW works fine without constraints. Fig. 6 extends the table presented in Czerwonka (2006), presenting the resulting test size for different inputs of parameters and values for each tool. These inputs are commonly used to compare CIT algorithms. The input is described as $Numvalues^{Numparameters}$, indicating that the same number of values is repeated for the number of parameters. In the first row, for example, the input is four parameters with three values each. References to the tools can be found in Table 1. The last column in Fig. 6 shows the test size for PROW, with the results showing that the comparison with PROW depends on the input and the algorithm. As it is seen the test sizes of the other algorithms are in general sightly less than PROW, but for some inputs PROW is better. Note, however, that these data are only to have an idea of the PROW behaviour, since PROW is designed to be used in presence of constraints and/or weights.

Regarding **PROW with constraints**, the following tools handle constraints (see Table 1): AETG, PICT, ATGT, TestCover, DDA, EXACT, CASA, CTS, TCG, ACTS. Some of these tools are not available, and the existing work that deals with constraints does not compare the test size with previous algorithms. Therefore, comparing PROW with the others is very difficult because it depends not only on the test input size and the number of deleted pairs, but also on the relation between the deleted pairs. To show the performance of PROW with respect to other algorithms, we use PICT, TestCover ACTS and CASA.

Fig. 7 shows the comparison for different test inputs. The first row corresponds to the Configurations example in Fig. 1. The others rows correspond to examples published in the literature, showing the reference.

Considering **PROW with prioritisation**, due the fact that the weight is used in PROW for repetitive testing (as explained in Section 3), the test size is the same for PROW without constraints or weights or for PROW without constraints and with weights. The same occurs with PROW with constraints. Then, Figs. 6 and 7 are also valid for PROW with weights assigned.

Parameter Sizes	AETG	IPO	TConfig	CTS	TestCover	DDA	PICT	EXACT	ATGT	TCG	PROW
3^4	9	9	9	9	9	9	9	9	11	?	**9**
3^{13}	15	17	15	15	15	18	18	15	23	20	**20**
$4^{15}\,3^{17}\,2^{29}$	41	34	40	39	29	35	37	?	62	35	**38**
$4^{1}\,3^{39}\,2^{35}$	28	26	30	29	21	27	27	21	65	27	**30**
2^{100}	10	15	14	10	10	15	15	10	25	16	**14**
10^{20}	180	212	231	210	181	201	210	?	367	218	**219**

Fig. 6. Test size comparison.

Example	Parameter Sizes	Deleted Pairs	PICT	TestCover	ACTS	CASA	PROW
Configuration Problem [Figure 1]	$3^1\,4^2\,2^1$	5	16	16	15	15	15
BBS Billing System[8]	3^4	1	12	12	12	11	11
Cruise Control [8]	$4^1\,3^1\,2^4$	2	14	14	12	12	12
Mobile Phone [8]	$3^3\,2^2$	7	11	12	12	9	11
Configuration Constraints [35]	$4^1\,3^1\,4^2 3^1$	3	24	20	22		20
Spin Simulator [13]	$4^5\,2^{13}$	13	16	19	23	19	25

Fig. 7. Test size comparison with constraints.

	Number of values					
Parameters	5	10	15	20	25	29
5	0,016	0,031	0,125	0,297	0,625	1,094
10	0,016	0,188	0,703	1,812	3,796	6,297
15	0,062	0,547	1,984	4,954	10,453	16,813
20	0,157	1,094	4,296	9,843	20,641	33,406
25	0,25	1,812	7,485	16,922	35,61	56,875
30	0,406	3,172	11,75	26,922	54,969	86,797
35	0,547	4,094	16,422	38,954	77,531	125
40	0,735	6,141	22,672	51,86	105,828	171,219
45	1,078	8,344	31	67,891	141,204	314,813
49	1,406	9,796	37,36	83,875	176,437	469,75

Fig. 8. PROW execution times (in seconds).

%Values	10										14									
% Deleted	0	10	20	30	40	50	60	70	80	90	0	10	20	30	40	50	60	70	80	90
Time (seconds)	0.15	0.219	0.218	0.25	0.266	0.313	0.328	0.39	0.438	0.578	0.344	0.391	0.406	0.484	0.547	0.609	0.704	0.718	0.891	0.969
Test Cases	222	185	187	179	177	183	171	164	157	154	433	364	362	341	339	338	336	329	317	317
%Values	18										22									
% Deleted	0	10	20	30	40	50	60	70	80	90	0	10	20	30	40	50	60	70	80	90
Time (seconds)	0.73	0.828	0.938	1.06	1.235	1.375	1.437	1.64	1.89	2.328	1.532	1.593	1.75	1.985	2.219	2.593	2.828	3.235	3.562	3.907
Test Cases	665	593	589	558	558	558	555	558	546	534	990	861	850	818	829	815	799	808	807	806
%Values	26										30									
% Deleted	0	10	20	30	40	50	60	70	80	90	0	10	20	30	40	50	60	70	80	90
Time (seconds)	2.61	2.829	3.187	3.48	3.938	4.531	4.906	5.58	6	7.093	3.609	3.859	4.297	4.86	5.40	6.094	6.906	7.906	8.813	10.33
Test Cases	1342	1192	1152	1140	1133	1155	1125	1109	1123	1094	1696	1478	1434	1403	1401	1373	1388	1389	1377	1350

Fig. 9. Execution time in function of deleted pairs.

3.4.3. Execution time analysis

With the aim of evaluating the execution time, we launched the algorithm against several sets of parameters (from 2 to 49) with a variable number of values (from 2 to 29). Fig. 8 shows the execution times (in seconds) of the algorithm for different configurations of parameters and values. This study was carried out on a common laptop with a CPU Intel Core 2 Duo at 2.39 GHz and 3.49 GB of RAM memory. We do not include comparisons with other algorithms because not all of them are available and, those which are, run on different platforms, what should lead to non-extrapolable results (for example: PICT runs with command line of Windows; TestCover runs on a web application).

As we noted when the analysis order was presented, the execution time depends on both variables included in Fig. 8. Note that the time has a stronger dependence on the number of values than on the number of parameters. In conclusion, we can say that the algorithm obtains test cases in a reasonable time even in uncommon situations, such as when using 30 parameters with 50 values each. In that case, the algorithm takes 469 s. In most likely cases, with 5–20 parameters and 5–20 values each, PROW takes between 0.01 and 10 s to obtain the result.

It is also interesting to know the performance of PROW when different numbers of pairs are deleted. Fig. 9 shows the time execution and the test size obtained when the parameters have 10, 14, 18, 22, 26 and 30 values each. Taking into account the constraints, between 0 and 90% of the pairs were deleted. As more pairs are deleted, fewer test cases are generated by PROW, but more time is required to obtain the test suite. This is due to the fact that PROW must search for the suitable combination to fulfil the requirements of deleted pairs, which is costly. Even in the least favourable situation (30 values, 90% deleted pairs), the time is 10.33 s.

4. Implementation of PROW in CTWeb

CTWeb[2] is a web application for combinatorial testing which implements different algorithms for test case generation. CTWeb assists the tester in the use of PROW, for two different purposes: combinatorial testing and product sampling.

4.1. CTWeb for combinatorial testing

In this case, CTWeb assists the test designer in obtaining meaningful coverage tests with the PROW algorithm. The process follows these steps:

1. **Introduce test parameters and their values**: The data can be manually introduced or by uploading a tab-separated values file. Fig. 10 shows the main page of the application, with the data presented in the previous example. The designer selects the radio button corresponding to the PROW algorithm and presses the Execution button. CTWeb outputs a table with all the possible combinations (see Fig. 3).
2. **Set constraints**: This is an optional step, where the tester can delete any unrealistic combinations. In Fig. 3 the tester removes *(Linux, IExplorer)*, *(Linux, Microsoft Word)*, *(Mac, IExplorer)*, *(Mac, Microsoft Word)* and*(Windows, Opera)*. The constraints could be also uploaded in the tab-separated file where the parameters and values are defined.
3. **Set weights**: This is an optional step, where the tester assigns the weight. Fig. 3 illustrates this point: the Sel. Factor column indicates the weight for each pair. The designer sets the following combinations at the top with a commonness weight

[2] http://alarcosj.esi.uclm.es/CombTestWeb/combinatorial.jsp.

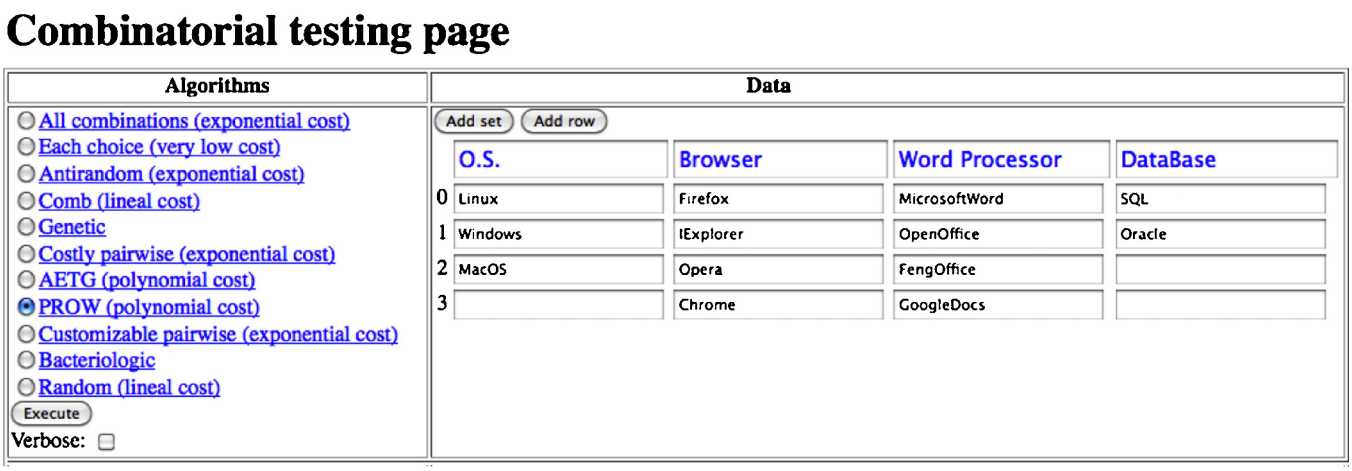


Fig. 10. CTWeb and the PROW algorithm.

of 1: *(Linux, Firefox)*, *(Windows, Explorer)*, *(Mac, Firefox)*, *(Linux, Open)*, *(Windows, Microsoft Word)*, *(Mac, Open)* and *(IExplorer, Microsoft Word)*. The weight assigned between pairs could be also uploaded in the tab-separated file where the parameters, values and constraints are defined.

4. **Set base test suite**: This is an optional step, where the tester can upload a file with the test cases that are used as a base for PROW. This functionality can be applied for combinatorial testing or for product sampling in SPL. Section 6.2 explains this functionality.
5. **Run PROW**: The tester presses again the Execute button and CTWeb runs the PROW algorithm with the information entered by the tester. The output is the test suite ordered according to the weight of the test cases. Fig. 5 shows the 15 test cases obtained for the configurations example. Section 5 shows the application of PROW to a real system.

4.2. CTWeb for product sampling in SPL

In this case, CTWeb helps the tester to obtain the product sampling set of all possible defined in a feature model; the steps in CTWeb are:

1. **Upload feature model**: A feature model representing the SPL is uploaded. The feature model is represented using a UML Model, that models the Orthogonal Variability Model (OVM) (Pohl, 2005). A UML profile for OVM was previously defined for us in Pérez Lamancha and Polo (2010). CTWeb automatically processes the feature model in two steps: first, the features are defined as parameters, its variants as its values and the relationships between the feature and its variants are processed. These relationships add new values to the parameters. The second step processes the relationships between features or between variants belonging to different features. These relationships constraints the combination between parameters.
2. **Set weights**: This is an optional step, where the tester assigns the weights to the combination between variants.
3. **Set base test suite**: This is an optional step, where the tester can upload a file with the test cases that are used as base for PROW. This functionality can be applied for combinatorial testing or for product sampling in SPL. Section 6.2 explains this functionality.
4. **Run PROW**: The tester presses again the Execute button and CTWeb runs the PROW algorithm with the information provided by the tester. The output is the product sampling test suite. Section 6 describes the application of CTWeb and PROW in an industrial SPL.

5. Case study for combinatorial testing: credit card management system

Recently, we collaborated with a testing consultancy in the Software Testing Centre (*Centro de Ensayos de Software*, CES)[3] in Uruguay.

The system under test was a credit card management system developed to process the credit card operations of different banks. The project corresponds to a migration from an AS400 platform with RPG to a .NET application with an Oracle database, so that the testing involved the migration of data between platforms. One of the critical subsystems to test was "Processing transactions": when a client makes a payment by credit card, the transaction sends a message to a server, codified under the ISO 8583 standard (*Financial transaction card originated messages — Interchange message specifications*). Depending on the card status, associated account, currency, etc., the subsystem under test processes the transaction and may authorise or reject the operation, as well as taking other kinds of decisions. In this subsystem, a set of 21 parameters was identified (Fig. 11 shows the names of the parameters and the number of values in each).

Taking into account the 21 parameters, the maximum number of possible combinations in the subsystem under test is 990,904,320. For illustration, Fig. 12 shows the values of 12 of the 21 existing parameters.

5.1. Considering neither constraints nor weights

The application of **PROW** to the set of parameters in Fig. 11 without taking into account the constraints produces **99** test cases, 37 of which contain unfeasible pairs. Since each test case is a tuple of 21 values proceeding from the 21 parameters, the search for unfeasible test cases is costly and error-prone. Moreover, valid pairs could be set outside the test suite if their container test cases are directly removed. Fig. 13 shows the first 10 test cases generated by PROW.

5.2. Considering constraints between pairs

Testing systems with many variables and/or values is often quite expensive, due to the huge number of possible combinations to be tested. In order to produce an example, we only focus on the constraints between three parameters:

[3] Software Testing Centre (Centro de Ensayos de Software, CES), http://www.ces.com.uy/.

	Parameter	Number of Values
1	Automated	2
2	Brand	2
3	Exists account	2
4	Exists card	2
5	Belongs to X bank	2
6	Credit card operates abroad?	2
7	Card status	7
8	Issuing bank	2
9	Portfolio limit overhead	2
10	Transaction Currency	4
11	Limitation by country exception	3
12	Cardholder	2
13	Additional limit	4
14	Account Currency	2
15	Postponed	2
16	Amount	4
17	Card expired	2
18	Transaction Location	2
19	Consume type	2
20	Account status	15
21	Credit limit	3
Total number of combinations: 990,904,320		

Fig. 11. Parameters and number of values.

- Transaction currency: represents the currency used in the transaction.
- Account currency: represents the currency of the account corresponding to the credit card.
- Location: indicates whether the transaction was made in the card holder's country or abroad.

The restrictions between these parameters are:

- If the *Account currency* is *peso*, then the *Transaction currency* cannot be *euro* or *other*.
- The customer in the country can only pay with the national currency (*peso*) or *dollars*, and cannot pay abroad with *pesos*. Then these pairs between *Transaction currency* and *Transaction location* are deleted: *(peso, abroad), (other, local), (euro, local).*

The shadowed cells in Fig. 13 correspond to unfeasible pairs. Test case 2 contains a transaction currency in pesos with the location abroad. This test case is unfeasible and needs to be modified. The same occurs with test cases 3, 6 and 7. In this example, taking only into account restrictions between 3 parameters, 37% of the test cases need to be modified. When we take into account all the restrictions between all the parameters, the number of modified test cases grows and it is very difficult to make the changes manually.

To take into account the **constraints with PROW**, the tester makes a previous selection of undesired pairs in order to generate the combinations. PROW resolves this problem generating 116 test cases, 17 test cases more than without any constraints. This difference is due to the restrictions between pairs (deleted pairs). Fig. 14 shows the first 10 test cases generated using PROW: all of them are valid and can be executed immediately. PROW takes 670 ms to generate the 116 test cases.

5.3. Considering constraints and weights between pairs

Moreover, the tester can assign a weight to those pairs that are more worthwhile to test. The following weights were assigned:

- For transactions inside the country, it is more important to test transactions in pesos. For transactions abroad, it is more important to test them in dollars. Then the pairs between Transaction currency and Transaction Location have: $weight(peso, local) = 2$, $weight(dollar, local) = 1 weight(peso, local) = 2$, $weight(dollar, abroad) = 1$, $weight(euro, abroad) = 0.5$, $weight(other, abroad) = 0.5$.
- The most important combinations are the local transactions. Furthermore, the tester assigns more importance to testing the dollar transactions over other currencies for payments by credit card in foreign countries. The interaction between

Account status	Credit limit	Belongs to X bank	Account Currency
Normal	Limit*excess> Available balance+ purchase amount	Yes	peso
Account retained by issuing bank	Limit*excess= Available balance+ purchase amount	No	dollar
Account cancelled by issuing bank	Limit*excess= Available balance+ purchase amount	**CC operates abroad?**	**Exists Card**
Exceeded up to 10%	**Card status**	Yes	Yes
Exceeded up to 20%	Normal	No	No
Exceeded up to 30%	Cancelled by inactivity	**Transaction Currency**	**Automated**
Exceeded up to 40%	Lose	dollar	Yes
Exceeded up to 50%	Fraud	peso	No
Exceeded up to 100%	Stolen	euro	**Brand**
Disabled	Cancelled	other	Z
Fraud	Returned	**Transaction Location**	Others
Cancelled	**Issuing bank**	abroad	**Exists account**
Debtor	Defined	local	Yes
Unchargeable	Undefined		No
Inactive account			

Fig. 12. Some parameters and the values of the "Processing Transaction" subsystem.

Test Case	0	1	2	3	4	5	6	7	8	9
Account Status	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Unchargeable	Unchargeable	Unchargeable
Card Status	Returned	CancelledBy Inactivity	Stolen	Fraud	Lose	Cancelled	Normal	Returned	CancelledByIn activity	Stolen
Autom	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Brand	Others	Z	Z	Z	Z	Z	Z	Z	Z	Z
Issuing bank	defined	defined	defined	defined	defined	undef	defined	defined	undef	defined
Exist Account	yes	yes	yes	yes	yes	no	yes	yes	no	yes
Credit Limit	lim2	lim1	lim1	lim1	lim1	lim2	lim3	lim3	lim1	lim1
Belongs X Bank	yes	yes	yes	yes	yes	no	no	yes	no	yes
CC operates abroad?	abroad_yes	abroad_yes	abroad_yes	abroad_yes	abroad_yes	abroad_no	abroad_yes	abroad_yes	abroad_no	abroad_yes
Account Curr	dolar	peso	peso	peso	peso	peso	peso	peso	peso	peso
Trn Curr	other	dolar	euro	peso	dolar	dolar	peso	euro	dolar	dolar
Trn location	abroad	abroad	abroad	abroad	abroad	local	abroad	abroad	local	abroad
Exist Card	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Portfolio	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Limitation	lim2	lim1	lim1	lim1	lim1	lim2	lim3	lim1	lim2	lim1
Card holder	no	yes	yes	yes	yes	yes	no	no	yes	yes
Add limit	ad4	ad1	ad1	ad2	ad1	ad2	ad3	ad2	ad1	ad1
Posponed	yes	yes	yes	yes	yes	no	yes	yes	no	yes
Amount	amount3	amount1	amount2	amount1	amount1	amount2	amount4	amount2	amount1	amount1
Card Expired	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Type	Type 1	Type 1	Type 1	Type 1	Type 1	Type 1	Type 1	Type 1	Type 1	Type 1

Fig. 13. First 10 test cases generated using the PROW algorithm without constraints.

the Transaction location and whether the credit card operates abroad or not is also important. The weights assigned are: $weight(abroad_{yes}, local) = 1$, $weight(abroad_{no}, local) = 2$, $weight(abroad_{yes}, abroad) = 0.5$, $weight(abroad_{no}, abroad) = 1$.

With these weights assigned, each combination (test case) will be returned to the tester in a given order, which will depend on the weight of the pairs in the combination. Thus, the algorithm provides a prioritised list of combinations, which allows the prioritised execution of those test cases with more importance. A characteristic of pairwise techniques is that, depending on the number of values in the parameters, there will be pairs which will appear more than once. Sometimes, different values can be selected for a new test case. From the tester's point of view, the best test cases are those that test the most important combinations. For the banking example, without assigning weight, 24 transactions (20%) take *local* as the Transaction Location. When the weight is assigned as described above, 87 out of 112 transactions are local.

Fig. 15 shows the test cases obtained by means of PROW with restrictions and weights, using the values for the weight and the

Test Case	0	1	2	3	4	5	6	7	8	9
Account Status	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account	Inactive Account
Card Status	Returned	Returned	CancelledBy Inactivity	Stolen	Stolen	Fraud	Lose	Lose	Lose	Cancelled
Autom	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Brand	Others	Z	Z	Z	Z	Z	Others	Z	Z	Z
Issuing bank	defined	defined	defined	undefined	defined	defined	undefined	defined	defined	defined
Exist Account	no	yes	yes	yes	yes	yes	yes	no	yes	yes
Credit Limit	lim1	lim1	lim1	lim2	lim1	lim1	lim2	lim3	lim1	lim1
Belongs X Bank	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
CC operates abroad?	abroad_yes	abroad_yes	abroad_yes	abroad_yes	abroad_yes	abroad_yes	abroad_no	abroad_no	abroad_yes	abroad_yes
Account Curr	peso	peso	peso	peso	dolar	peso	dolar	peso	peso	peso
Trn Curr	dolar	dolar	dolar	dolar	euro	dolar	other	peso	dolar	dolar
Trn location	local	abroad	abroad	abroad	abroad	abroad	abroad	local	abroad	abroad
Exist Card	no	yes	yes	yes	yes	yes	no	yes	yes	yes
Portfolio	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
Limitation	lim1	lim1	lim1	lim3	lim1	lim1	lim2	lim2	lim1	lim1
Card holder	yes	yes	yes	yes	yes	yes	no	yes	yes	yes
Add limit	ad4	ad2	ad1	ad3	ad1	ad1	ad2	ad3	ad1	ad1
Posponed	no	yes	yes	no	yes	yes	no	yes	yes	yes
Amount	amount1	amount2	amount4	amount4	amount1	amount1	amount1	amount1	amount3	amount1
Card Expired	no	yes	yes	yes	yes	yes	yes	no	yes	yes
Type	Type1	Type1	Type1	Type1	Type1	Type1	Type1	Type1	Type1	Type1

Fig. 14. First 10 test cases generated using the PROW algorithm with constraints.

Test Case	0	1	2	3	4	5	6	7	8	9
Account Status	Inactive Account	Account cancelled	Exceeded 100%	Exceeded 50%	Exceeded 50%	Exceeded 40%	Exceeded 30%	Exceeded 20%	Cancelled	Unchargeable
Card Status	Lose	Returned	Normal	Returned	Normal	Fraud	CancelledBylnactivity	Normal	Cancelled	Lose
Autom	yes	yes	no	yes	yes	yes	no	yes	no	yes
Brand	Z	Others	Z	Z	Others	Others	Others	Others	Others	Z
Issuing bank	defined	undefined	undefined	defined	undefined	defined	defined	defined	undefined	undefined
Exist Account	no	no	no	no	yes	yes	no	no	no	no
Credit Limit	lim3	lim2	lim3	lim3	lim1	lim1	lim1	lim1	lim2	lim2
Belongs X Bank	yes	yes	no	yes	yes	yes	no	no	no	yes
CC operates abroad?	abroad_no	abroad_no	abroad_no	abroad_no	abroad_no	abroad_no	abroad_no	abroad_no	abroad_no	abroad_yes
Account Curr	peso	dolar	peso	peso	peso	peso	peso	peso	peso	peso
Trn Curr	peso	peso	peso	peso	peso	peso	peso	peso	peso	peso
Trn location	local	local	local	local	local	local	local	local	local	local
Exist Card	yes	no	yes	yes	no	no	no	yes	no	yes
Portfolio	yes	yes	yes	yes	yes	no	no	yes	no	yes
Limitation	lim2	lim1	lim2	lim3	lim2	lim3	lim1	lim3	lim2	lim1
Card holder	yes	yes	yes	yes	yes	no	yes	yes	no	yes
Add limit	ad3	ad1	ad3	ad3	ad4	ad2	ad1	ad1	ad2	ad4
Posponed	yes	no	yes	yes	no	yes	no	no	no	no
Amount	amount1	amount1	amount4	amount4	amount1	amount3	amount4	amount2	amount2	amount1
Card Expired	no	yes	yes	no	yes	no	no	yes	no	yes
Type	type1	type2	type2	type1	type2	type2	type2	type2	type2	type1
Weight	**4.0**	**4.0**	**4.0**	**4.0**	**4.0**	**4.0**	**4.0**	**4.0**	**4.0**	**3.0**

Fig. 15. First 10 test cases generated using PROW with constraints and weights.

restrictions described above. Last column shows the total weight calculated for each test case; this weight is calculated by adding the weight of each pair in the test case as defined by the tester. The test cases are, moreover, ordered using the total weight. Both during the implantation of the new system as in its future maintenance, testers will be able to execute the most important test cases first, according to the ordered test suite.

In the actual case study, there were much more constraints that those presented in the example, and were much more complex since they involved almost all the parameters. One singular case is a special bank (labelled "Bank X" in Fig. 11) which only operates abroad, in dollars, with a certain value in the "Type" parameter, etc. In order to do the managing of these restrictions more easily, we decided to split the input file into two files: one with the concrete values for Bank X, and another with the general constraints applicable to the remaining banks. Also in the special Bank X file there were restrictions among pairs, which also served as a motivation to apply PROW. The testers positively valued the help provided by PROW, which made their work in this project easier. In previous testing projects with pairs exclusions, the task of reviewing the test to remove invalid combinations was difficult and time-consuming, and PROW relieved them of these tasks, accelerating the test design step and letting them anticipate the test execution step. Note that, besides removing the invalid pairs of a test suite got with a usual pairwise algorithm, the tester must add new test cases to visit the valid pairs that are visited by the invalid test cases.

6. Using PROW for product sampling in SPL

As we pointed out in the Introduction, combinatorial testing strategies are also required in other contexts, such as Software Product Lines (SPLs). At a high level, a SPL can be described with a feature model, that describes the features in the line and how they can be combined in order to get products. The notation in the feature model restricts the presence of certain features in some products, what is a excellent context for the application of PROW.

As a case study we use the Sensors System SPL, that allows monitoring and controlling the conditions of the load transported in a truck. Along the truck there are several nodes, and each node has up to 5 sensors to measure the temperature, pressure, humidity and concentrations of CH_4 and CH_6 gases. Via bluetooth, the nodes periodically send the sensor data to a smart phone in front of the driver. This SPL has 8 features and more than 20 variants. Fig. 16 shows its feature model, the rectangles represent features, optional features are depicted using a white circle above the rectangle and mandatory features are depicted using a black circle. Notice that the features are not totally independent. For instance, "*F1 requires F2*" denotes that no product can exist that exhibits feature F1 without F2 (e.g., Basic monitoring requires Bluetooth). Following the features are briefly explained:

- Sensor system: The sensors used to monitor the truck can be of three types: GPS, Inertial system or Sensor node.
- Sensors device: Indicates the possible sensors in the truck, the variants can be: Temperature, Pressure, CO, CH_4, C_6H_6, Radiation, Humidity, etc.
- Positioning: Indicates the position of the truck. It can be calculated using a GPS or the Inertial System in the truck.
- Bluetooth: Indicates the way in which the sensors data is send.
- Mobile Phone: Indicates whether the mobile monitoring uses a basic or a tom-tom system.
- Mobile O.S: Indicates the O.S. of the mobile, it can be: Windows Mobile, Android or Symbian.
- Mobile Screen: Indicates whether the mobile has touch screen or not.
- Alerts Communication: Indicates the way in that the alerts are communicated to the user. It can be: sms, mms or e-mail.

For this feature model, up to **18432** products could be generated. Building and testing such number is impossible in a reasonable time. CTWeb and PROW help to reduce the testing effort

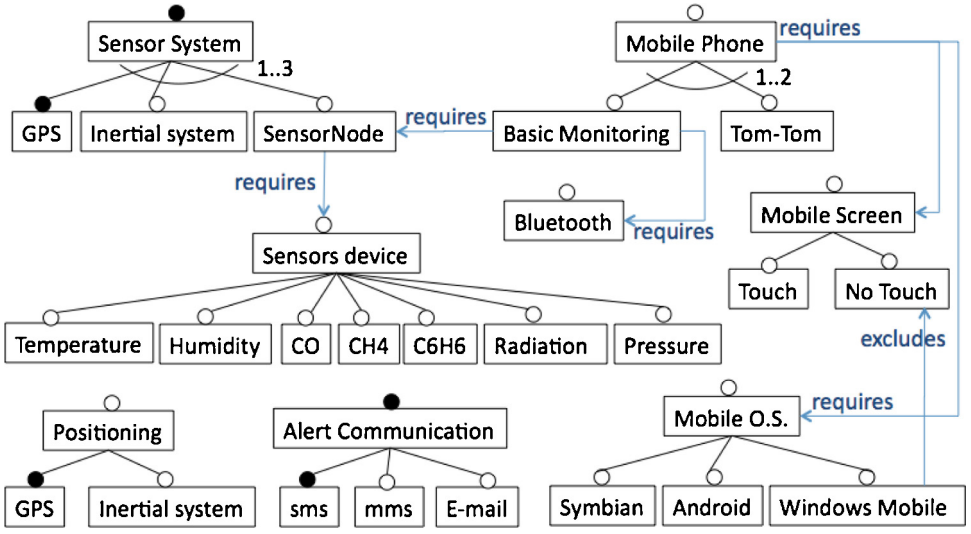


Fig. 16. Feature model for Sensors System SPL.

obtaining a product sampling set. Next subsection explains these steps in detail for the Sensor System SPL.

6.1. Uploading and processing the feature model

The first step is Uploading the Feature Model. Once the feature model is loaded in CTWeb, it is automatically processed. In previous work (Pérez Lamancha and Polo, 2010), the way in which the feature model is loaded and processed in CTWeb was described. This section explains briefly how CTWeb processes the Sensor System feature model. This processing is made in two automated steps:

1 **Creating parameters and its values**: This step takes into account the feature, its variants and the relationships between them. The basic relationships between a feature and its variants can be:
- Optional: optional variants are included in some products (Pohl, 2005). In Fig. 17(a), the *Bluetooth* feature is optional. For this feature, two values must be taken into account: when the feature is part of the product and when not. The values for this parameter are: *Bluetooth_yes* and *Bluetooth_no*. *Mobile Screen* (Fig. 17(b)) is also optional and has two variants, both of them optional too. The values for this parameter are: *touch*, *no touch* and *MobileScreen_no*. This last parameter is added to specify when the feature is not included in a product.
- Mandatory: mandatory variants must be selected for an application if, and only if, the associated variation point is part of the application (Pohl, 2005). In Fig. 17(c), the *Alert Communication* feature is mandatory and has three variants: mms and e-mail (which are optional) and sms (which is mandatory and, thus, must be present in all the values of this parameter). Then, the values for AlertCommunication are: sms, sms and mms, sms and e-mail.
- Alternative: The alternative choice groups a set of variants that are related through an optional variability dependency to the same variation point. Alternative defines the range for the amount of optional variants to be selected for this group (Pohl, 2005). Fig. 17(d) shows the optional *Mobile Phone* feature, which has two optional variants: Basic Monitoring and Tom-tom. This feature also has an alternative relationship which states that this feature has either 1 or 2 variants. The values for *Mobile Phone* are obtained combining the possible variants, i.e. Basic Monitoring, Tom-Tom, Basic Monitoring and Tom-tom. Also the value MobilePhone_no is added because the feature is optional.

Fig. 18 shows the 8 parameters obtained, showing in red the values that do not appear in the feature model and that were added due the relationships between the feature and its variants.

2 **Set constraints between values**: This step processes the relationships between features, these relationships can be: requires or excludes. For detailed information about how to treat all the relationships, consult (Pérez Lamancha and Polo, 2010). Fig. 16 shows the relationships that occurs in Sensor System feature model, these relationships are:
- Feature requires feature: Fig. 19 describes this situation, where the *Mobile Phone* feature requires the feature *Mobile Screen*. This means that the product that combines *a variant of Mobile Phone* without *a* variant of *Mobile Screen* is not possible. In this case the pairs between *Basic Monitoring* and *MobileScreen_no* and between Tom-tom and *MobileScreen_no* are deleted from the pairs table between the features *Mobile Phone* and *Mobile Screen*.
- Variant requires feature: Fig. 20(a) describes this situation, where the variant *Basic Monitoring* requires the feature *Bluetooth*. This means that the product that combines *Basic*

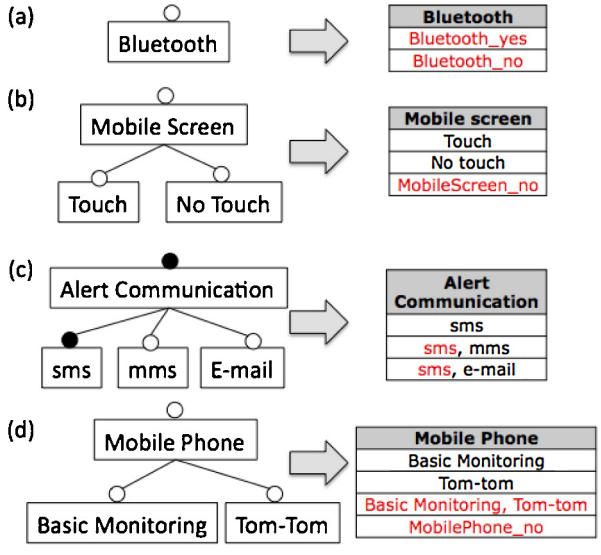


Fig. 17. Some features and how they are converted into parameters.

Sensor System	Mobile Phone	Sensors Device	Mobile O.S.	Bluetooth
GPS	Basic Monitoring	Humidity	Windows	Bluetooth_yes
GPS, Inertial system	Tom-tom	Temperature	Symbian	Bluetooth_no
GPS, Sensor Node	Basic Monitoring, Tom-tom	GasCO	Android	
GPS, Inertial system, Sensor Node	MobilePhone_no	Pressure	MobileO.S._no	
Alert Communication	**Positioning**	GasCH4		**Mobile screen**
sms	GPS	GasC6H6		Touch
sms, mms	GPS, Inertial system	Radiation		No touch
sms, e-mail	Positioning_no	SensorsDevice_no		MobileScreen_no

Fig. 18. Set of parameters for Sensor System SPL.

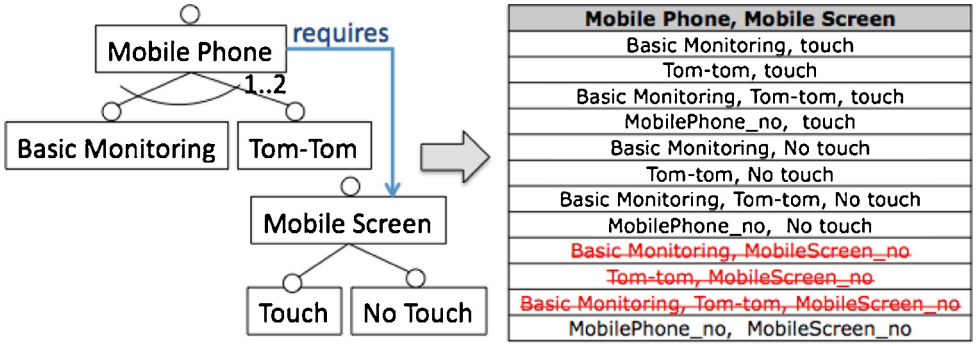


Fig. 19. Feature requires feature relationships.

Monitoring without *Bluetooth* is not viable. In this case the pairs containing *Basic Monitoring* and *Bluetooth_no* are deleted from the pairs table between the features *Mobile Phone* and *Bluetooth*. The same occurs with the variant *Sensor Node* that requires the feature *Sensors Device* (see Fig. 20(b)).

- Variant requires variant: Fig. 21 describes this situation, where the variant *Basic Monitoring* requires the feature *Sensor Node*. This means that the product that combines *Basic Monitoring* without *Sensor Node* is not viable. In this case the pairs containing *Basic Monitoring* and not containing*Sensor Node* are deleted from the pairs table between the features *Sensor System* and*Mobile Phone*.
- Variant excludes variant: Fig. 22 describes this situation, where the variant *Windows Mobile* excludes the variant *No Touch* in the feature *Mobile Screen*. This means that the product that combines *Windows Mobile* with the *variant NoTouch* of *Mobile Screen* is not possible. In this case the pair between *Windows Mobile* and *No Touch* is deleted from the pairs table between the features *Mobile O.S.* and *Mobile Screen*.

Once the feature model is uploaded and automatically processed, CTWeb shows the parameters, its values and the pairs table with the pairs that were removed.

6.2. Setting the base test suite

This functionality is particularly important in SPL testing because each test case obtained from PROW is a product that will be assembled and tested. Since this is costly and time consuming, it is preferable to assemble the most frequent products. This feature allows to enter a list of the products that will be tested anyway

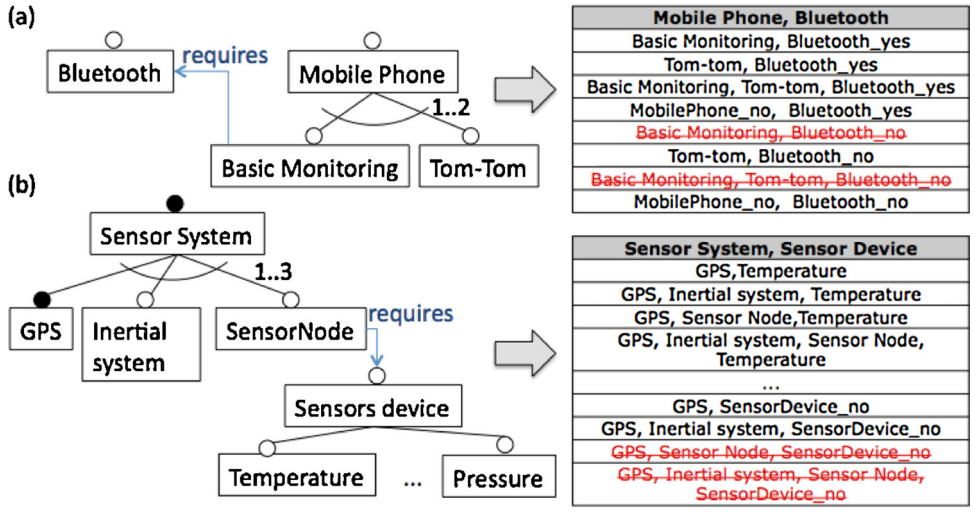
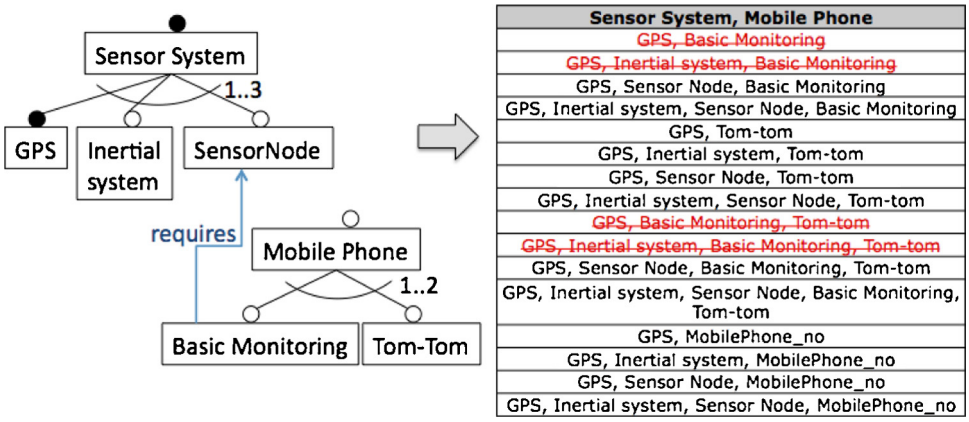
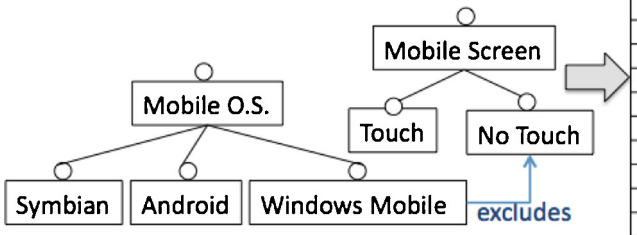


Fig. 20. Variant requires feature relationships.



Sensor System, Mobile Phone
~~GPS, Basic Monitoring~~
~~GPS, Inertial system, Basic Monitoring~~
GPS, Sensor Node, Basic Monitoring
GPS, Inertial system, Sensor Node, Basic Monitoring
GPS, Tom-tom
GPS, Inertial system, Tom-tom
GPS, Sensor Node, Tom-tom
GPS, Inertial system, Sensor Node, Tom-tom
~~GPS, Basic Monitoring, Tom-tom~~
~~GPS, Inertial system, Basic Monitoring, Tom-tom~~
GPS, Sensor Node, Basic Monitoring, Tom-tom
GPS, Inertial system, Sensor Node, Basic Monitoring, Tom-tom
GPS, MobilePhone_no
GPS, Inertial system, MobilePhone_no
GPS, Sensor Node, MobilePhone_no
GPS, Inertial system, Sensor Node, MobilePhone_no

Fig. 21. Variant requires variant relationships.



Mobile O.S., Mobile Screen
Windows Mobile, touch
Symbian, touch
Android, touch
MobileO.S._no, touch
~~Windows Mobile, No touch~~
Symbian, No touch
Android, No touch
MobileO.S._no, No touch
Windows Mobile, MobileScreen_no
Symbian, MobileScreen_no
Android, MobileScreen_no
MobileO.S._no, MobileScreen_no

Fig. 22. Variant requires variant relationships.

because they are important products. These products are uploaded into CTWeb. The tool marks as visited the pairs they include. So, PROW takes these test cases as the basis for the rest of the test suite. Fig. 23 shows 4 products that are the most commonly built for the Sensor System SPL.

6.3. Run PROW for product sampling in SPL

Once the testers set the preferences, the PROW algorithm can be executed, and as a result, 47 products are obtained. PROW takes 63 ms in obtain the test suite. These 47 products are the product sampling set that represents the products to test in Sensor System SPL. Fig. 24 shows 10 out of 47 products to be tested for Sensor System LPS. For example, the first product has a Sensor system with GPS, Inertial system and Sensor Node, the mobile phone uses the Basic Monitoring and the Tom-tom, the Sensor Device measures Radiation, the mobile screen is NoTouch, has Bluetooth, uses Android as mobile O.S., the positioning is calculated using the GPS and the alerts are communicated sending sms and an e-mail. None of the products defined in Fig. 23 appear in the final test suite obtained. The reason for that is that PROW optimises the test suite to cover all the pairs between values, minimising the test suite size.

Using the base test suite: If a base test suite was upload to CTWeb, the pairs involved in each test case that belongs to the base test suite are marked as visited. Then, PROW is executed. In this case, the test cases included in the base test suite are copied to the output test suite. The other test cases are built to cover pairwise and minimising the test suite size. Obviously, due to the fact that the base test suite may not be the minimum test set that covers these pairs, the output test suite obtained by PROW is equal or grater than the obtained without using a base test suite. For the base test suite shown in Fig. 23, 53 products are needed to obtain pairwise coverage.

7. Related works

Table 1 summarised the main algorithms in CIT. In this section we present related works describing solutions for two aspects of combinatorial testing: the constraint problem and the prioritisation problem.

Product	Sensor System	Mobile Phone	Sensors Device	Mobile screen	Bluetooth	Mobile O.S.	Positioning	Alert
1	GPS, Inertial system, Sensor Node	Basic Monitoring, Tom-tom	Temperature	Touch	Bluetooth_yes	Windows	GPS, Inertial system	sms
2	GPS, Inertial system, Sensor Node	Basic Monitoring	Radiation	No touch	Bluetooth_yes	Android	GPS, Inertial system	sms
3	GPS, Inertial system	Tom-tom	SensorsDevice_no	No touch	Bluetooth_no	Symbian	GPS, Inertial system	sms, mms
4	GPS, Sensor Node	Basic Monitoring	Humidity	Touch	Bluetooth_yes	Windows	Positioning_no	sms

Fig. 23. Base test suite for Sensor System SPL.

Product	Sensor System	Mobile Phone	Sensors Device	Mobile screen	Bluetooth	Mobile O.S.	Positioning	Alert
1	GPS, Inertial system, Sensor Node	Basic Monitoring, Tom-tom	Radiation	NoTouch	Bluetooth_yes	Android	GPS	sms, e-mail
2	GPS, Inertial system	MobilePhone_no	SensorsDevice_no	MobileScreen_no	Bluetooth_no	MobileO.S._no	Positioning_no	sms, mms
3	GPS, Sensor Node	Basic Monitoring, Tom-tom	Temperature	NoTouch	Bluetooth_yes	Symbian	GPS, Inertial system	sms, e-mail
4	GPS	Tom-tom	SensorsDevice_no	NoTouch	Bluetooth_no	Symbian	GPS	sms, e-mail
5	GPS, Inertial system, Sensor Node	Tom-tom	Temperature	NoTouch	Bluetooth_no	Android	GPS, Inertial system	sms, e-mail
6	GPS, Sensor Node	BasicMonitoring	GasC6H6	NoTouch	Bluetooth_yes	Symbian	Positioning_no	sms, e-mail
7	GPS, Sensor Node	BasicMonitoring	GasCH4	Touch	Bluetooth_yes	Windows	GPS	sms, mms
8	GPS, Inertial system, Sensor Node	BasicMonitoring	GasC6H6	Touch	Bluetooth_yes	Android	GPS	sms, e-mail
9	GPS, Inertial system	MobilePhone_no	SensorsDevice_no	Touch	Bluetooth_yes	MobileO.S._no	GPS	sms, mms
10	GPS, Sensor Node	Basic Monitoring, Tom-tom	Humidity	NoTouch	Bluetooth_yes	Android	GPS, Inertial system	sms, mms

Fig. 24. Ten of the 47 products obtained with PROW for Sensor System SPL.

Constraint problem: Several works about combinatorial testing leave the treatment of constraints on inputs as future work. However, some researchers have worked on solutions to this problem. The PICT algorithm, proposed by Czerwonka (2006), deals with constraints which are specified as conditional statements in a kind of configuration program. Thus, each unfeasible pair is described as a condition. There are not many details about the syntax of the language and the detail in the algorithm is not enough to reuse this technique. Also, PICT only works by command line in the Windows platform. Calvagna and Gargantini (2008) deal with full constraints too, integrating model checkers with combinatorial testing in the presence of constraints. The constraints are expressed as formal predicate expressions on the input variables. They present the pseudocode of the algorithm, whose solution is developed to be used with enumerations and booleans. The main drawback of this approach is the use of formal predicates to use this technique: although the more formal the specification of a program, the greater the capability to automate the testing process, formal specifications are expensive to write and maintain (Hoffman et al., 2005), difficult to apply in practice and usually unavailable in industrial software (Ball et al., 2000). Thus, in conventional testing practice, the programmer writes the tests for a new system feature after the corresponding production code for that feature (Erdogmus et al., 2005). Lott et al. made an improvement to AETG to handle restrictions between pairs using a modelling language (Lott et al., 2005). This tool is in the remodel category and these authors show how to use the modelling language to set the constraints. However, the information about how these restrictions are handled in the AETG algorithm is missing. They do not show either how their algorithm works or the resulting test suite. Also, the AETG[4] code is proprietary and its use requires the acquisition of a license.

Sherwood (1994) presents the TestCover tool, which requires that the tester remodels the input to handle the constraints. The tool uses blocks that define valid configurations and the tester separates the inputs depending on the validness of their values, what is very similar to apply a common pairwise algorithm to several combinations of valid values. As the number of parameters or unfeasible pairs increases, more quickly grows up the complexity of the method. This way of handling the constraints by constructing valid blocks is not very intuitive.

Cohen et al. (2007) present a general technique for representing constraints that can be efficiently processed by existing constraint-solving libraries and can be incorporated into existing classes of algorithms. These authors do not propose any algorithm, but just a notation to describe the constraints, which later should be processed and integrated into existing algorithms.

Bryce and Colbourn (2006) addressed "soft constraints" in their proposal: a weight is assigned to each pair, and unfeasible pairs are not removed although they receive a very low weight. Thus, these pairs finally appear in the last test cases in the suite. In our research, unfeasible pairs never appear in the final test suite and the test suite is also ordered by weight.

As noted in Section 2.1, Cohen et al. (2007) categorise constraints handling in Remodel, Soft-Only, Expand and Full. PROW is in the Expand category: the constraints between pairs must be explicitly described, which can be done by checking the pairs to remove in the table of pairs or they can be uploaded from a CVS file.

Prioritisation problem: Regarding test case prioritisation, for Rothermel et al. (2001), testers order their test cases according to some priority criterion, such as "the expected frequency of use of some system features" or "the expected capability of test cases to reveal faults". These authors focus their work on the second goal, analysing nine prioritisation techniques to increase the likelihood of revealing faults earlier in the testing process: No prioritisation, Randomised ordering, Optimal rate of fault detection (as per the authors, "this is not a practical technique, as it requires a priori knowledge of the existence of faults and of which test cases expose which faults"); All statements; Not yet covered statements; All branches; Not yet covered branches; Probability of exposing faults and Probability of exposing faults, adjusted to consider effects of previous test cases. As Rothermel et al. explain, their work considers only one objective (likelihood of detecting faults), but they recognise that other interesting objectives exist, such as the frequency of use of system features, which is one of the goals addressed and resolved in this article, by means of the pairs weight and the ordered test suite produced. Elbaum et al. (2002) present empirical results of the effectiveness of six prioritisation techniques. The experiment indicates that the relative effectiveness of prioritisation techniques can vary across the type and nature of the programs.

Bryce et al. (2011) examined a set of prioritisation criteria for two classes of event driven software: GUI and web application. The results revealed that the characteristics of the application influenced the results, but CIT techniques was one of the best criteria in terms of rate of fault detection. In other work, Bryce et al. (2011) add cost of the test case to the prioritisation criteria, defining a metric

[4] AETG is a trademark of Telcordia Technologies. http://aetgweb.argreenhouse.com/.

for cost-based combinatorial coverage. As result, they obtain that cost-based pairwise is quite effective.

The algorithms that take into account the prioritisation problem are: PICT and DDA. PICT (Czerwonka, 2006) allows the tester to include a weight, but this weight is applicable only to the values and not to the interactions between values (i.e. pairs). Furthermore, the algorithm published does not show how this prioritisation is made. DDA (Bryce and Colbourn, 2006, 2009) proposes a method for constructing a prioritised test suite. As in our case, the testers can apply the prioritisation criteria of their choice to derive weights (priorities), locally optimising each test.

Product sampling for testing in SPL: Related includes: Cabral et al. (2010) define a graph-based testing approach that selects products and features for testing based on a feature dependency graph. Oster et al. (2010) combine graph transformation, combinatorial testing, and forward checking to define the products to test. They translate the feature model into a binary constraint solving problem (CSP) and consider predefined sets of products. Schürr et al. (2010) define an approach to obtain test cases from features models, using the classification tree method and a metamodel for feature testing to obtain the test cases. Stricker et al. (2010) adapt and extend the def-use testing technique. Based on a data flow-based coverage criterion, the test coverage of previously tested SPL products is employed in order to identify data dependencies that are covered and do not have to be tested again. Perrouin et al. (2010) use *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. The authors take into account mandatory and optional features and the requires relationship, but do not consider excludes. Segura et al. (2010) present a set of relations between input feature models and their set of products. Given a feature model and its known set of products, a set of neighbouring feature models together with their corresponding set of products are automatically generated and used to test different analyses. Hervieu et al. use pairwise to generate test configurations from a feature model using constraint programming. These authors define a specific metamodel to represent a feature model. They only take into account relationships from variant to variant in the feature model and not relationships from feature to variant, variant to feature and feature to feature as in our case (Hervieu et al., 2011). In a previous work (Pérez Lamancha and Polo, 2010) we presented the *Customisable pairwise algorithm* to select products to test in SPLs. This work improves this algorithm with the possibility of assigning a weight between pairs of feature values allowing prioritisation. The final test suite is ordered using this weight.

With respect to the reviewed works, we consider that our approach makes an important contribution due to the fact that no other algorithm combines constraint handling with weight-based prioritisation for repetitive testing, as PROW does. Also, other important contributions are the publicly available web tool CTWeb to use the algorithm, as well as the publication of its source code under the GNU license.

8. Conclusions and future work

In this paper we have presented PROW, an algorithm for pairwise testing that allows the tester to handle constraints and weight. The introduction of this double first step avoids the costly and error-prone task of the manual elimination of unfeasible pairs (i.e., pairs which will not be present in the released system and whose elimination may also remove interesting pairs); moreover, it makes it possible to obtain a test suite whose test cases are ordered according to the corresponding weight assigned to each one.

The algorithm pseudocode is described and implemented in CTWeb, a publicly available web tool. The algorithm has a polynomial cost and is thus applicable to the actual practice of software testing. In fact, it has been applied to test a complex banking system with many variables. In this case study, classic pairwise strategies produced many unfeasible combinations. PROW successfully helped to generate the test cases in less time and to prioritise according to the pairs' importance, preventing the inclusion of meaningless pairs.

The PROW algorithm is also suitable for use in Software Product Line testing. Variability in the line is captured through features. Features are represented in a feature model, which is later used to generate the products from the line. From a 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. We are extending CTWeb to deal with feature models, using PROW to define the products in the line to be tested. Depending on the restrictions between features (excludes, requires, etc.), feature models can be processed to detect invalid pairs of features, which correspond to products that do not belong to the line. This is our ongoing and future work with the PROW algorithm: the definition of the rules to determine unfeasible combinations between features. One possible improvement of PROW would be its extension to *t*-wise testing.

Acknowledgements

The authors would like to thank Oscar Díaz and Maider Azanza from the University of the Basque Country for their comments and helpful reviews. Also thanks to Monica Wodzislawski and María Elisa Presto from the Software Testing Centre (CES) in Uruguay, for having accepted the application of PROW in the case study. Also thank to Danilo Caivano and Giuseppe Vissaggio, who provided us the MonIca Mobile SPL. This work is partially supported by the GEODAS project (Spanish Ministry of Economy and Competitiveness and European Fund for Regional Development, TIN2012-37493-C03-01).

References

Ball, T., Hoffman, D., Ruskey, F., Webber, R., White, L., 2000. State generation and automated class testing. Softw. Test. Verif. Reliab. 10 (3), 149–170.
Bryce, R., Sampath, S., Memon, A., 2011. Developing a single model and test prioritization strategies for event-driven software. IEEE Trans. Softw. Eng. 99, 1–18.
Bryce, R.C., Colbourn, C.J., 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Inf. Softw. Technol. 48 (10), 960–970.
Bryce, R.C., Colbourn, C.J., 2007. The density algorithm for pairwise interaction testing. Softw. Test. Verif. Reliab. 17 (3), 159–182.
Bryce, R.C., Colbourn, C.J., 2009. A density-based greedy algorithm for higher strength covering arrays. Softw. Test. Verif. Reliab. 19 (1), 37–53.
Bryce, R.C., Sampath, S., Pedersen, J.B., Manchester, S., 2011. Test suite prioritization by cost-based combinatorial interaction coverage. Int. J. Syst. Assur. Eng. Manage., 1–9.
Cabral, I., Cohen, M., Rothermel, G.,2010. Improving the testing and testability of software product lines. In: Int. Software Product Lines Conference. Springer, pp. 241–255.
Calvagna, A., Gargantini, A., 2008. A logic-based approach to combinatorial testing with constraints. Tests Proofs, 66–83.
Clements, P., Northrop, L.M., 2001. Software Product Lines – Practices and Patterns. Addison-Wesley, Boston, MA, USA.
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 Trans. Softw. Eng. 23 (7), 437–444.
Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C., 1996. The combinatorial design approach to automatic test generation. IEEE Trans. Softw. Eng. 13 (5), 83–88.
Cohen, M.B., Colbourn, C.J., Ling, A.C.H.,2003. Augmenting simulated annealing to build interaction test suites. In: International Symposium on Software Reliability Engineering. IEEE, pp. 394–405.
Cohen, M.B., Dwyer, M.B., Shi, J.,2007. Interaction testing of highly-configurable systems in the presence of constraints. In: International Symposium on Software Testing and Analysis. ACM, pp. 129–139.

Czerwonka, J., 2006. Pairwise testing in real world. In: 24th Pacific Northwest Software Quality Conference, vol. 82.
Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. 28 (2), 159–182.
Erdogmus, H., Morisio, M., Torchiano, M., 2005. On the effectiveness of the test-first approach to programming. IEEE Trans. Softw. Eng., 226–237.
Garvin, B.J., Cohen, M.B., Dwyer, M.B.,2009. An improved meta-heuristic search for constrained interaction testing. In: 2009 1st International Symposium on Search Based Software Engineering. IEEE, pp. 13–22.
Grindal, M., Offutt, J., Andler, S.F., 2005. Combination testing strategies: a survey. Softw. Test. Verif. Reliab. 15 (3), 167–200.
Griss, M.L.,2000. Implementing product-line features by composing aspects. In: Software Product Lines: Experience and Research Directions, vol. 576, First Software Product Lines Conference (SPLC). Springer, p. 271.
Hartman, A., 2005. Software and hardware testing using combinatorial covering suites. In: Graph Theory, Combinatorics and Algorithms. Springer US, New York, USA, pp. 237–266.
Hervieu, A., Baudry, B., Gotlieb, A.,2011. Pacogen: automatic generation of pairwise test configurations from feature models. In: 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 120–129.
Hoffman, D., Strooper, P., Wilkin, S., 2005. Tool support for executable documentation of java class hierarchies. Softw. Test. Verif. Reliab. 15 (4), 235–256.
Kang, K.C., et al., November 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021. SEI.
Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J., 2008. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. Softw. Test. Verif. Reliab. 18 (3), 125–148.
Lei, Y., Tai, K.C.,1998. In-parameter-order: a test generation strategy for pairwise testing. In: International High-Assurance Systems Engineering Symposium. IEEE Computer Society, p. 254.
Lott, C., Jain, A., Dalal, S.,2005. Modeling requirements for combinatorial software testing. In: 1st Workshop on Advances in Model-based Testing. ACM, pp. 1–7.
Nie, C., Leung, H., 2011. A survey of combinatorial testing. ACM Comput. Surv. (CSUR) 43 (2), 11.
Oster, S., Markert, F., Ritter, P.,2010. Automated incremental pairwise testing of software product lines. In: Int. Software Product Lines Conference. Springer, pp. 196–210.
Pérez Lamancha, B., Polo, M.,2010. Testing product generation in software product lines using pairwise for features coverage. In: In 22nd IFIP Int. Conference on Testing Software and Systems. Springer-Verlag, pp. 111–125.
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: 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, pp. 459–468.
Pohl, K., et al., 2005. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Stuttgart, Germany.
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng., 929–948.
Schürr, A., Oster, S., Markert, F., 2010. Model-driven Software Product Line Testing: An Integrated Approach. Springer, Berlin, Germany, pp. 112–131.
Segura, S., Hierons, R.M., Benavides, D., Ruiz-Cortés, A.,2010. Automated test data generation on the analyses of feature models: a metamorphic testing approach. In: International Conference on Software Testing. IEEE, pp. 35–44.
Sherwood, G., 1994. Effective testing of factor combinations. In: Proceedings of the Third International Conference on Software Testing, Analysis, and Review (STAR94).
Stricker, V., Metzger, A., Pohl, K.,2010. Avoiding redundant testing in application engineering. In: Int. Software Product Lines Conference. Springer, pp. 226–240.
Tung, Y.W., Aldiwan, W.S.,2000. Automating test case generation for the new generation mission software system. In: 2000 IEEE Aerospace Conference Proceedings, vol. 1. IEEE, pp. 431–437.
Williams, A.W., 2002. Software Component Interaction Testing: Coverage Measurement and Generation of Configurations. University of Ottawa, Ottawa, Canada.
Yan, J., Zhang, J., 2008. A backtracking search tool for constructing combinatorial test suites. J. Syst. Softw. 81 (10), 1681–1693.

Beatriz Perez received a PhD in computer science from the University of Castilla-La Mancha, Spain. His research interests include software engineering and software testing.

Macario Polo is an associate professor of computer science at the University of Castilla-La Mancha, Spain. His research interests include software testing and software engineering automation. Polo received a PhD in computer science from the University of Castilla-La Mancha.

Mario Piattini is a full professor of computer science at the University of Castilla-La Mancha, Spain. His research interests include global software development and green software. Piattini received a PhD in computer science from the Universidad Politécnica de Madrid.