IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays

Andrea Calvagna
University of Catania, Italy
Dip. Ingegneria Informatica
e delle Telecomunicazioni
andrea.calvagna@unict.it

Angelo Gargantini
University of Bergamo, Italy
Dipartimento di Ingegneria dell'informazione
e metodi matematici
angelo.gargantini@unibg.it

Abstract

Verification and validation of highly-configurable software systems, such as those supporting many optional or customizable features, is a challenging activity. In fact, due to its intrinsic complexity, formal modeling of the whole system may require a great effort. Modeling activities may become extremely expensive and time consuming, and the tester may decide to model (at least initially) only the inputs and require they are sufficiently covered by tests. The recent wide spreading usage of combinatorial interaction testing (CIT) is dramatically improving the effectiveness of this activity. Although there exist analytical ways to derive minimal sized CIT test suites, they are not applicable to all task sizes. Therefore, researchers have explored many techniques based on greedy or heuristic algorithms that may lead to sub-optimal result in the size of the built test suite but that are applicable to problems of real size. In this paper, a new parameter-based heuristic algorithm for the construction of pairwise covering test suites is presented; it is based on a symmetry property of covering arrays and it is called IPO_S. Time and space complexity of IPO_S is discussed in comparison especially with the only other parameter-based approach existing in literature. The proposed approach is supported by a prototype implementation, and experimental assessment is also presented.

1 Introduction

Systematic testing of highly-configurable software systems can be challenging and expensive. Moreover, traditional approaches to systematic testing may not be able to detect incorrect behaviors or failures caused by unintended interaction between optional features [25, 38]. Since most of the faults in a software system are triggered by unintended interaction of a relatively low number of input parameters, typically 4 to 6 [25], testing of all combinations of input configurations can be very effective in revealing software defects [24]. The recent widely spreading usage of combinatorial interaction testing (CIT) is improving the effectiveness of this activity. CIT consists in systematically testing all possible "partial" configurations (that is, involving up to a fixed number of parameters only) of a given software system in order to reveal unintended feature interactions. In fact, it has been empirically shown that most of the failures of software systems are actually triggered by unintended interaction between a very small number of parameters.

Combinatorial Interaction Testing (CIT) systematically explores *t*-way feature interactions inside a given system, by effectively combining all t-tuples of parameter assignments in the smallest possible number of test cases. This allows to budget-constraint the costs of testing while still having a testing process driven by an effective and exhaustive coverage metric [25, 8]. The most commonly applied combinatorial testing technique is *pairwise* testing, which consists in applying the smallest possible test suite covering all the *pairs* of input values (each pair in at least one test case). In fact, it has been experimentally shown that a test suite covering just all pairs of input values can already detect a significantly large part (typically 50% to 75%) of the faults in a program [26, 15]. Dunietz *et al.* [16] compared t-wise coverage to random input testing with respect to the percentage of structural (block) coverage achieved, showing that the former achieves better results if compared to random test suites of the same size. Burr and Young [5] reported 93% code coverage from applying pairwise testing of a large commercial software system, and many CIT tools (see [30] for an up to date listing) and techniques have already been developed [14, 17, 25] and are currently applied in practice [3, 33, 23].

Combinatorial testing can be applied to a wide variety of problems: highly-configurable software systems, software

Table 1. Input domain of a basic billing system (BBS) for phone calls

access	billing	calltype	status
LOOP	CALLER	LOCALCALL	SUCCESS
ISDN	COLLECT	LONGDISTANCE	BUSY
PBX	EIGHT_HUNDRED	INTERNATIONAL	BLOCKED

Table 2. A test suite for pairwise coverage of BBS

#	billing	calltype	status	access
1	EIGHT_HUNDRED	LOCALCALL	BUSY	PBX
2	CALLER	LONGDISTANCE	BLOCKED	LOOP
3	EIGHT_HUNDRED	INTERNATIONAL	SUCCESS	ISDN
4	COLLECT	LOCALCALL	SUCCESS	LOOP
5	COLLECT	LONGDISTANCE	BUSY	ISDN
6	COLLECT	INTERNATIONAL	BLOCKED	PBX
7	CALLER	LOCALCALL	SUCCESS	ISDN
8	CALLER	LOCALCALL	BUSY	PBX
9	EIGHT_HUNDRED	LONGDISTANCE	BLOCKED	ISDN
10	COLLECT	LONGDISTANCE	BUSY	LOOP
11	COLLECT	LONGDISTANCE	SUCCESS	LOOP

product lines which define a family of softwares, hardware systems, and so on. As an example, Table 1 reports the input domain model of a simple telephone switch billing system [27], which processes telephone call data with four call properties, each of which has three possible values: the `access` parameter tells how the calling party's phone is connected to the switch, the `billing` parameter says who pays for the call, the `calltype` parameter tells the type of call, and the last parameter, `status` tells whether or not the call was successful or failed either because the calling party's phone was busy or the call was blocked in the phone network. While testing of all the possible configurations for BBS would require $3^4 = 81$ tests, pairwise coverage can be obtained by the test suite reported in Table 2 which contains only 11 tests.

Significant time and cost savings can be achieved by implementing this approach, as well as in general with *t*-wise combinatorial interaction testing. In fact, although *t*-wise coverage of a system featuring n configuration options, each ranging in r values, requires testing a number of configurations which grows exponentially, $r^t \binom{n}{t}$, combinatorial test suites effectively selects a much lower number of test cases. As an example, for a system with a hundred boolean parameters (2^{100} possible test cases) pairwise coverage would require 19800 number of configurations to be tested, which can be accomplished with only 10 test cases by a combinatorial test suite. Similarly, pairwise coverage of a system with twenty ten-valued options (10^{20} possible tests) requires coverage of 19000 pairs, for which it is sufficient a combinatorial test suite of only 200 tests cases.

From a mathematical point of view, the problem of generating a minimal set of test cases covering all *t-wise* tuples of input assignments is equivalent to computing a *covering array* (CA) of *strength t* over the alphabet of all the input symbols [19]. Covering arrays are combinatorial structures which extend the notion of *orthogonal array* [2], which in turn are generalizations of *latin squares*. For a given system under test exposing n features, all ranging in r distinct values, a *t-strength, n-degree, r-order* covering array $CA_\lambda(N; t, n, r)$ is an $N \times n$ array where every $N \times t$ subarray shall contain each t-wise combination of the r symbols at least λ times. However, when applied to combinatorial system testing only the case when $\lambda = 1$ is of interest, that is, where every t-tuple is covered at least once. Moreover, for testing of real systems we are really interested in mixed covering arrays $MCA_\lambda(N; t, n, \vec{r})$, with $\vec{r} = (r_1, r_2, ...r_n)$ a vector of positive integers, in which each system feature may range on a different number r_i of symbols. Each of the N rows will be a complete test case specification, assigning values to all features, while each column of the CA will list all values chosen for each feature.

Devising a general algorithm to compute a CIT test suite is a non trivial problem, as computing a minimal set of test cases that satisfies n-wise coverage can be NP-complete [36, 31]. Although exact solutions to compute it by *algebraic* construction do exist [22], they are not generally applicable to large problems. As a consequence, researchers have addressed the issue of designing general solutions to compute a minimal sized CIT test suite based on *greedy* heuristic searches [4], although these may lead to near-optimal results (typically, only an upper bound on the size of constructed suite may be guaranteed). Less traditional *meta-heuristic* algorithms have also been proposed [9, 29], based on bio-inspired techniques (i.e. genetic algorithms, simulated annealing) in order to converge to a near-optimal solution after an acceptable number of iterations. In addition, *recursive* construction techniques do exist [12, 32], computing a near-optimal test suite by composing together instances of sub-arrays which are already minimal. However, most of the already existing algorithms and tools fall in the *greedy* category. These algorithms build up the test suite incrementally by adding either one test case, that is a row, at the time or one parameter, that is a column, at a time to the test suite, until coverage is complete. Although this latter strategy (known as *parameter-based* construction) has proven to outperform many of the existing strategies of the former type (known-as *AETG-like* after its most influential algorithm [8]), many variants of the *AETG-like* strategy have been already proposed in literature, in contrast to only one *parameter-based* algorithm IPO [26], which makes this point worth investigating.

Note that many algorithms and tools for combinatorial interaction testing already exist in the literature. Grindal et al. count more than 40 papers and 10 strategies in their survey [17]. There is also a web site [30] devoted to this subject and several automatic tools are commercially [8] or freely available [26]. For this reason, even a small improvement over existing techniques seems now hard to accomplish.

In this paper a new *parameter based* algorithm for the construction of pairwise covering test suites is presented. Based on a symmetry property of combinatorial test suites we were able to formulate a (very short) recursive construction algorithm. This algorithm's heuristic, in contrast to existing parameter-based approaches, can optimize in a smaller set of pairs, actually a subset of just two parameters, irrespective from the total number of the system parameters. The paper is structured as follows: section 2 gives insights on the theoretical aspects of the considered problem and explains the idea behind our new construction technique, section 3 discusses results from experiments on its implementation in order to assess its performance and scalability, comparing also with most existing tools, and eventually section 4 draws our conclusive statements and directions for future work.

2 Computing the Test Suite by IPO$_S$

In this section we present a novel test generation technique called IPO$_S$ (the s is short for *Symmetry*). IPO$_S$ is a *parameter-based* strategy as it computes the test suite incrementally, one parameter at time, starting from the first two parameters and then extending the test suite in order to add pairwise coverage, one column for each additional parameter. However, complete pairwise coverage of the new parameter may require also vertical extension of the test suite, that is additional rows. The parameter-based construction technique has been introduced by Tai and Yu with the IPO algorithm [26]. In contrast with that algorithm, in the proposed approach the horizontal and vertical growth are not performed in separate and consecutive stages, but are instead interleaved. As it can be easily derived, adding pairwise coverage for a new feature to an existing n-features wide test suite implies extending it to cover an additional set P of $|P| = n(n+1)r^2$ pairs between the new parameter and all the previous[1]. In fact, the heuristic in [26, 34] searches in P in order to choose the best symbol for next assignment. Conversely, in our algorithm this extent is reduced to a constant r^2 pairs, that is P does not grows with the number n of features in the suite. This is possible as all other required pairs will be already covered with a suitable initialization of the new column symbols, thanks to a key property of covering arrays that we call *symmetry*, as showed in details in the following. However, when mixed ranges are considered, in order to compute the corresponding MCA the additional assumption that parameters will be processed in *descending* ranges order will be required. As a consequence, the variables and their ranges have to be known in advance all at once, although this is not a big issue as it is usually the case.

[1]We assume here for sake of simplicity that all features have same range, r, but is easy to extend it to mixed alphabet sizes.

2.1 Incremental Construction

Let us consider a system under test which has n distinct input parameters $(v_1, \dots, v_n)$, and assume that v_i has range r_i and values in $V_i = \{0, 1, ..., r_i - 1\}$, that is $V = V_1 \cup \dots \cup V_n$ is the whole input domain. Note that actual values of any parameter have been mapped to an equivalent set of symbols (natural numbers) for convenience. Let S be the current test suite that covers every pair between the first $i-1$ parameters with $2 < i \leq n$. The current test suite is a mixed covering array of $N \times i - 1$. We want now to extend S to cover also the i-th parameter, v_i. Given the assumption that parameters have been processed in descending range order, we have $r_{k-1} \leq r_k \quad \forall k = 2, \dots, n$, that is the range of v_i will always be less or equal to those of all $v_1, \dots, v_{i-1}$ parameters already in the test suite.

As a consequence, it is always possible to define an injection $f_k : V_i \rightarrow V_k$ which associates distinct values of v_i with distinct values of one parameter v_k $(k \in \{1 \dots i-1\})$ of choice. We now consider the extended range $V_k^\star = V_k \cup \{x\}$, and similarly $V_i^\star$, in order to include the additional *don't care* symbol, x. It is now possible to define a new function $F_k : V_k^\star \rightarrow V_i^\star$ as follows:

$$F_k(\alpha) = \begin{cases} f_k^{-1}(\alpha) & \alpha \in \mathrm{V_i} \\ \\ x & \alpha \notin V_i \end{cases}$$

Simply put, F_k is a surjection, which reverses the injection f_k (as we actually aim to compute values in V_i) and extends it by mapping any unassociated value in its range to the *don't care* value. As an example, a straightforward injection f_k is the *copy* function $f(\alpha) = \alpha$. In this case the function $F_k(\alpha)$ is equal to α if $\alpha \in V_i$, otherwise is equal to x. That is, it copies the k-th column except for those values that are not in the range of the new variable v_i, which become x.

Theorem 1 (Symmetry property) *By using function F_k to initialize v_i from v_k guarantees that v_i is already paired to all the parameters except v_k.*

This can be proved by considering that the chosen parameter v_k was by definition already paired with all other $(i-1)$ parameters in S, then also v_i will be paired to the same parameters as v_k, i.e. all except v_k itself. In fact, the function F_k, by construction associates each value of v_i to a distinct value of v_k, that is, V_i will be just a renamed subrange of V_k. Thus, we only need to complete coverage for pairs between v_i and v_k that, at this stage, will always be exactly r_i pairs out of $r_i \cdot r_k$, whatever the chosen injection. Note that the additional coverage requirements of the extended test suite now have been already greatly reduced without any greedy processing, yet. At this point, for each of the uncovered pairs a greedy function will select an available row position, i.e. one corresponding to a redundant or already covered pair, that can be edited to create the considered missing pair. Changes to actual values of the suite can only happen to the values of the newly added column only, while x values could safely be edited also in the rest of the suite at any moment, as their modification can only increase the overall coverage. It is important to note that the changes operated to the last column will have to preserve the pairs already covered between all parameters, v_i included. As the number of suitable rows for inserting the missing pairs is limited to a subset of the (few) suite rows, the effectiveness of this greedy stage relies in correctly selecting the right sequence of edits to apply in order to avoid unnecessary rows addition to the suite. However, if it fails since no suitable positions are left available in the last column to form the missing pair, it will eventually add it to the suite as a new row. This new row (a test case) will have bindings for just the required pair of parameters, and don't care values in all other parameters.

From now on, we will use as function F_k the copy function F_{i-1} which copies the last generated column in the column to add for v_i, except for those values of v_{i-1} that are not in the range of v_i and that will be substituted by x.

Now let's clarify it with an example. Assume we have a system with variables v_1, v_2, and $v_3 \in \{0,1,2\}$, and a fourth variable $v_4 \in \{0,1\}$. We than first combine the two major parameters and get an intermediate test suite with nine rows. At next iteration we then initialize third column (for parameter v_3) with the same values of the second, or with x, as from F_{i-1}. As v_2 was already totally paired[2] with v_1, also v_3 whose assignment resembles that of a subset of v_2, is consequently totally paired with v_1 (see figure 1-(a)). We then only need to complete pairing of v_3 and v_2. Note also that some of the pairs between v_3 and v_2 are already covered by construction: $\{(0,0),(1,1),(2,2)\}$, and that they are (three times) redundant. We only need to cover pairs $\{(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)\}$ and have six redundant column places than can be reassigned to this aim. A simple heuristic applicable here is to edit the symbol in the row position of first redundant instance of a pair, that is, in this case, we choose to change assignment on fourth row so to form the missing pair $(v_2,v_3)=(0,1)$. Since this will delete the unique pair $(v1,v3)=(1,0)$ we need to restore it elsewhere, and precisely it can be done by another change of assignment (always in last column) in sixth row from 2 to 0. This last induced change delete in turn the existing pair $(v_1,v_3)=(1,2)$, immediately restored by changing fifth row redundant assignment to 2, and also increased (v_2,v_3) coverage also by additional pairs $(2,0)$ and $(1,2)$. Similarly, missing pair $(v_2,v_3)=(0,2)$ is added by changing value in seventh row from 0 to 2, which deleted pair $(v_1,v_3)=(2,0)$ as a side-effect, immediately restored by changing value in redundant eighth row from 1 to 0. Eventually, missing pairs $(v_2,v_3)=(1,0)$ and $(v_2,v_3)=(2,1)$ are obtained by changing v_3 assignment in sixth and ninth row respectively, without any side-effect. This ends the first iteration since pairing of v3 and v2 is now complete (see figure 1-(b)). Please note that this procedure guarantees that at least one instance of each pair is in the final suite, but does not require that each pair of values is covered the same number of times.

A further iteration is needed to add the last parameter v_4 to the suite, which has smaller range than its predecessors. Again, this column's values are copied from current values in adjacent column v_3 (see figure 1-(c)) and then edited to complete the coverage. Since v_3 has higher range than v_4 this time x (don't care) values will be used to initialize unmatched rows. As shown in figure 1-(c) we only miss the set of pairs $(v_3,v_4)=\{(1,0),(0,1),(2,0),(2,1)\}$ to complete the coverage. The first pair is created changing v_4 value in fourth row, and restoring deleted pairs $(v_2,v_4)=(0,1)$ and $(v_1,v_4)=(1,1)$ by replacing with 1 the x in seventh row and fifth row respectively. This also add missing pairs $(v_3,v_4)=(2,1)$. Moreover, creating missing pair $(v_3,v_4)=(0,1)$ by changing to 1 the value in sixth row has no side effects, and eventually changing the x in third row to 0 creates the last missing pair $(v_3,v_4)=(2,0)$. As no parameters need to be added the process is complete and the pairwise coverage is also 100% complete for all pairs (figure 1-(d)). Please note that modifications have occurred at each iteration only in the new column of S, and that a value can be changed only if it is redundant and has not been changed already. This requirement ensure that coverage is always increased, as added pairs cannot be undone, and as a consequence it guarantees that the process eventually terminates. On the other hand, this safety condition can lead to sub-optimal results. In fact, in more complex tasks it may also happen that no rows exist which allows for the required substitution. In this case a new row will be added to the test suite to host the missing pair. The following subsections shows algorithmic details of both main algorithm and its value editing, recursive routine, *recover()*.

2.2 Main loop

The main algorithm sorts the variables and builds the tests for the two first variables v_1 and v_2 simply by building every possible combination of their values. Then, it adds the test for every variable v_i at the time by calling an auxil-

[2]Please allow us to use the following the expression *totally paired* to denote that all their pair combinations of symbols are covered.

v_1	v_2	$\mathbf{v_3}$
0	0	0
0	1	1
0	2	2
1	0	0
1	1	1
1	2	2
2	0	0
2	1	1
2	2	2

(a)

v_1	v_2	v_3
0	0	0
0	1	1
0	2	2
1	0	**1**
1	1	**2**
1	2	**0**
2	0	**2**
2	1	**0**
2	2	**1**

(b)

v_1	v_2	v_3	$\mathbf{v_4}$
0	0	0	0
0	1	1	1
0	2	2	x
1	0	1	1
1	1	2	x
1	2	0	0
2	0	2	x
2	1	0	0
2	2	1	1

(c)

v_1	v_2	v_3	v_4
0	0	0	0
0	1	1	1
0	2	2	**0**
1	0	1	**0**
1	1	2	**1**
1	2	0	**1**
2	0	2	**1**
2	1	0	0
2	2	1	1

(d)

Figure 1. Example task: $3^3 2^1$**.**

```
//add the i-th column to S
add_column($S, r_i, i$){
    //let $P_i$ be the set of pairs between $v_i$ and $v_{i-1}$
    for each row $j$ in $S$ do{
        //copy the (i-1)th column
        //let $a = S[j][i-1]$
        if $a \leq r_i$ then b:=a else b:=$x$;
        S[j][i]:= b;
        if $(a, b) \in P_i$ then {
            remove $(a, b)$ from $P_i$;
            set row $j$ as required;
        }
    }
    //recovering
    for each $(a, b) \in P_i$ recover($a, i-1, b, i$);
}
```

Figure 2. suite extension algorithm

iary method *add_column()*. This method is reported in Fig. 3 and is structured in two stages: the first stage initializes the values added for the new parameter to each of the test cases of the current suite, that is its last column. To do so it implements here the copy function used in the previously shown example. This step guarantees to give complete coverage between the added i-th parameter and all the others but one: the one that has been *copied*. Thus, a second stage is necessary, where an internal recursive function, *recover* is invoked for each of the missing pairs between these two parameters. This function always ends up with the addition of the specified pair to the coverage, either because it succeeds in doing so by refining the assignments in last column or because it adds it as a new row.

```
//recover the pair $(a, b)$ at columns $(k, i)$
recover($a, k, b, i$){
    //let p be a pair $(a, b)$
    choose any row $j$ where $S[j][k] = x$
        or where $j$ is unrequired and $S[j][k] = a$;
    if none, add $p$ as a new row and return;
    if $S[j][k] = x$ then $S[j][k] := a$;
    set row $j$ as required;
    //create the missing pair
    $b_{old} := S[j][i]$;
    $S[j][i] := b$;
    // restore any deleted pairs
    if ($b_{old} = x$) return;
    for all $h : 1..i-2$ do:
        if pair $(S[j][h], b_{old})$ is not covered
            then recover(S[j][h],h,$b_{old}$,i);
}
```

Figure 3. recursive pair-recover function

2.3 Recursive recovering

The recover function edits the values in i-th column of S until the specified pair p is successfully added to S, in columns k and i. Editable rows are those whose value in column i form a redundant pair with that of column k, and thus it is replaceable with some other value to create the missing pair. Routine is recursive and either sets the x value in current "working" column k, if any, to the required value or changes the value in column i, of a given row. However, a change in one value in the i-th column could cause deleting an unique pair already existing between the current parameter and the previous, on the same row. In that case, any deleted pair will be restored recursively by means of the same routine, as the structure of the recursive recover algorithm is parametrized on the column indexes and the values of the pair that has to be created. Note that this can also recursively induce other changes and thus other deletions. However, eventually this recursive process is always guaranteed to terminate, since the recursion is subject to the condition that no row position can host more than one change, at most. In other words, what previous recursions did cannot be later undone. This ensures that the applied recursive changes will always increase the coverage, eventually, and puts an upper bound to the depth of the recursion ($N \cdot i$, see next section). In fact, if no chances exist to create the desired pair or to recover a deleted pair then the recursion terminates, after adding the missing pair as an additional row, with all x values except for the values of the pair p itself.

2.4 Time and Space complexity

In [8], the authors show that the number of tests for pairwise coverage grows at most logarithmically in n and quadratically in r with n the number of parameters and r the number of values[3]. The regression analysis we performed on experimental data in tables 3 and 4 to asses the dependency on the number of parameters n and the range of the parameters d, empirically confirmed that the size of the test suites generated by IPO$_S$ grows in $O(\log(n))$ and $O(r^2)$, respectively. Computational time complexity of *add_column* function is $O(r^2)$, as it is dominated by the loop of calls to the *recover()* function. The latter can at each execution either modify a row assignment (always in last column only) or add a new test case (a new row) to the test suite. Recursion can be induced only when a row has been modified. This in turn can happen only N times overall, as any row can be modified only once. Moreover, since the recursive call is nested inside an $O(n)$ loop, the total time complexity of the function is $O(N \cdot n)$, that is $O(r^2 n \log n)$.

[3]This result can be extended to MCAs by averaging over all the actual ranges.

Thus, as the *add_column()* is called n times, the overall time complexity of the proposed strategy is $O(r^4 n^2 \log n)$, that is the same of the reference tool AETG [8] (as proved in [26]) . Although better time complexity is achieved by IPO, which is in $O(r^3 n^2 \log n)$, this is computed assuming that a crucial step of that algorithm, that is the test of a flag indicating if a pair is already covered or not, can be implemented with time complexity in $O(1)$, that is, that it is possible to directly index a flag for each pair in the whole combinatorial space. Clearly, this assumption in turn requires a data structure of size linearly proportional to the size of the problem domain, that is, in $O(r^n)$. Indeed, this may be a huge space requirement, growing exponentially with the number of involved parameters, and so dramatically limiting the scalability of that approach, or its extensibility to higher degree of interaction strengths. In contrast, the space requirements of the presented approach is in $O(r^2)$, as it is always limited to the pair combinations of two parameters only, irrespective of the overall number of involved parameters. In addition, the fact that the our tool space complexity is dominated by r suggests that the proposed algorithm could be more effective if applied to tasks characterized by a large number of parameters but with short-to-medium ranges. This will be confirmed by results of experimental assessment in Section 3.

3 Evaluation

The proposed algorithm has been implemented in a prototype tool, which has than been applied to a series of tasks in order to benchmark its performance. In this section the results of this assessment are reported and compared with performance of other existing tools. The exponential notation used to represent the problem domain size in [19] has been also adopted here, that is d^n means a task with n parameters of range d. Note that the tool fully implements the algorithm, including its support for tasks with mixed ranges.

The first series of experiments reports the size of the computed test suite for two series of tasks, designed in order to separately assess its scalability with respect to increasing number of features n (Table 3), or increasing range r (Table 4), respectively. In these first two tables, the respective performance of the tool PairTest [26] has been also reported as a reference for comparison, since it is the only other existing tool implementing a *parameter-based* approach, besides IPO$_S$, to the best of our knowledge. In order to test the potential of this approach irrespective from any specific, local optimization strategy that could be applied instead, our tool is currently implementing a simple random search heuristic, and thus shows non deterministic behavior, in contrast to the deterministic heuristic implemented by the compared tool PairTest. In particular, the suite sizes reported in the following for our tool IPO$_S$ are the best out

n	10	20	30	40	50	60	70	80	90	100
PairTest	31	34	41	42	48	48	51	51	51	53
IPO$_{S-min}$	28	34	38	41	43	45	47	48	49	50
IPO$_{S-\mu}$	30.1	35.7	40.7	43.2	45.2	46.9	48.5	49.9	50.9	52.1
IPO$_{S-\sigma}$	1.35	1.20	1.21	1.21	1.23	0.99	1.17	1.10	1.17	1.21
IPO$_{S-time}$	0.01	0.01	0.03	0.06	0.10	0.15	0.21	0.28	0.37	0.48

Table 3. Comparative suite size for 4^n task size.

r	5	10	15	20	25	30
PairTest	47	169	361	618	956	1355
IPO$_{S-min}$	44	177	390	686	1079	1548
IPO$_{S-\mu}$	47.0	187.1	418.8	741.4	1138.9	1631.7
IPO$_{S-\sigma}$	2.13	5.36	12.72	25.71	37.06	48.45
IPO$_{S-time}$	0.01	0.12	0.80	4.67	15.31	40.74

Table 4. Comparative suite size for r^{10} task size.

of fifty tries. Although this comparison may appear unfair, please note that it is common practice in CIT literature to compare the performance of the existing CIT tools, irrespective of whether their underlying approach is deterministic or not. It is also common practice for tools based on greedy, non deterministic search heuristics (i.e. like AETG, SA-SAT, ATGT, PICT[4], our tool IPO$_S$ and so on) to be benchmarked over a relatively small series of just fifty executions, although a larger number could lead the search to better results. Experiments have all been executed on a laptop computer equipped with Intel Core Duo 2.4GHz and 2 GBytes of RAM.

Average size, standard deviation and execution time, measured in seconds for a single try of the tool, have also been reported for each computed task. The execution times shown are very small even for large sized tasks, encouraging the adoption of this tool for test data generation in a production environment, and showing that the time performance of this tool is good enough to allow for much more than fifty executions in still reasonable overall time. As an example, a five minute only execution of the tool on a task size like 4^{50}, whose computing time is 0.10 seconds, would allow for six hundred executions on the tool.

Data in Tables 3 and 4 show that our approach performance and scalability is at least comparable to that of the other parameter-based tool, both with respect to the number of parameters n and to their range r, despite current implementation is not adopting yet any optimizing heuristic.

Tables 5, 6, 7, and 8 report series of experiments performed in order to compare the performance of IPO$_S$ in terms of the size of the generated test suite, to that of main existing tools for pairwise testing, on several sets of tasks with increasing complexity. Tasks in Table 5 are covering arrays with 3 factors of 3 to 7 values each, presented in [10], while tasks in Table 6 have been taken presented in [19] and [13]. As shown, our tool performed well in all of the considered tasks, and while in most of the remaining it has the same (already optimal) performance in two other tasks, and still performs well in the two remaining tasks. Table 7 reports an additional performance comparison between our tool and other well-known existing tools on a different and larger series of tasks of increasing sizes. Although the performance achieved is not optimal, it is still acceptable and very close to that of the other compared tools, specially for smaller tasks, which in overall demonstrates the practicability of the underlying approach.

Task size	IPO$_S$	ATGT [7]	mAETG-SAT [10]	SA-SAT [10]	PICT [13]	TestCover [35]
3^3	9	9	9	9	10	9
4^3	16	15	16	16	17	16
5^3	27	25	26	25	26	25
6^3	36	36	37	36	39	36
7^3	53	51	52	49	55	49

Table 5. Comparative suite sizes for a^3

Table 8 reports computed test suite sizes for an additional set of seven example specifications which this time are actual models of real-world software systems. Comparison is made, where data was available, with the recently presented tools mAETG-SAT presented in [10] and with our model checker based tool ATGT presented in [6] and [7]. BBS is a model of a basic telephone billing system [27], already presented in the introduction of this paper. TCAS models the specification of a software module part of a Traffic Col-

[4]The PICT tool core algorithm does make pseudo-random choices but, unless a user specifies otherwise, the pseudo-random generator is always initialized with the same seed value, in order to purposely let two executions of the tool on the same input produce the same output.

Task size	IPO_S	ATGT [7]	AETG [8]	PairTest [26]	TConfig [37]	CTS [18]	Jenny [21]	DDA [11]	AllPairs [28]	PICT [13]
3^4	9	11	9	9	9	9	11		9	9
3^{13}	17	19	15	17	15	15	18	18	17	18
$4^{15}3^{17}2^{29}$	32	38	41	34	40	39	38	35	34	37
$4^1 3^{39} 2^{35}$	23	27	28	26	30	29	28	27	26	27
2^{100}	10	12	10	15	14	10	16	15	14	15
10^{20}	220	267	180	212	231	210	193	201	197	210

Table 7. Comparative suite sizes of several combinatorial tools

Task size	IPO_S	ATGT [7]	PairTest [26]	TConfig [37]	CTS [18]	Jenny [21]
4^{10}	28	31	31	28	28	30
4^{20}	34	39	34	28	28	37
4^{30}	38	45	41	40	40	41
4^{40}	41	49	42	40	40	43
4^{50}	43	51	47	40	40	46
4^{60}	45	53	47	40	40	49
4^{70}	47	56	49	40	40	50
4^{80}	48	58	49	40	40	52
4^{90}	49	59	52	43	43	53
4^{100}	50	60	52	43	43	53

Table 6. Comparative suite sizes for 4^b

lision Avoidance System (TCAS) presented in [24]. Cruise Control models a simple cruise control system originally presented in [1], while the Mobile Phone example models the optional features of a real-world mobile phone product line, and has been recently presented in [10]. SPIN is a well-known publicly available model checking tool [20], and can be used as a simulator, to interactively run state machine specifications, or as a verifier to check properties of a specification. It exposes different sets of configuration options available in its two operating modes, so they can be accounted for two different tasks of different sizes. Finally, the GCC task is derived after the version 4.1 GNU compiler toolset, supporting a wide variety of languages, e.g., C, C++, Fortran, Java, and Ada, and over 30 different target machine architectures. Due to its excessive complexity the task size has been here reduced to model just the machine-independent optimizer that lies at the heart of GCC. These models have also been presented in [10]. For all these models, as can be seen from the data shown in Table 8, in all the computed test suites the tool was able to improve the best result over the other compared tools, but for TCAS which is already at optimal size in the other too. In this paper all tasks have been applied unconstrained, that is ignoring the restrictions on which pairs may legally be present in the computed test suite (i.e. because they are actually realizable) and which may not, if any. In fact, although we are currently working on extending this approach to support constraints over the inputs, we are leaving this issue out of the scope of this paper.

Spec	Task	IPO_S	ATGT	mAETG-SAT
BBS	3^4	9	11	
TCAS	$2^7 3^2 4^1 10^2$	100	100	
Mobile Phone	$2^2 3^3$	10	11	
Spin simulator	$2^{13} 4^5$	20	23	25
Spin verifier	$2^{42} 3^2 4^{11}$	29	33	33
GCC	$2^{189} 3^{10}$	16	19	24

Table 8. Test suite sizes for models of real-world systems.

4 Conclusions and future work

In this paper a new parameter-based technique for incremental construction of pairwise covering test suites has been presented, currently exloiting a symmetry property of covering array. The core algorithm is based on the new idea of inheriting the "pairing" relations between the parameters, due to a symmetry relation between columns of a covering array. Although many algorithms have been proposed for incrementally computing a test suite one row at the time, only one other algorithm sharing with us the (orthogonal) one-column-at-the-time approach currently exist in literature. Moreover, in contrast to that other algorithm, the presented algorithm is characterized by much lower space requirements, irrespective of the number of involved parameters. As a consequence it has shown to have also better scalability with respect to increasing the number of the task parameters. The experiments performed on the implementation of the proposed approach showed its space performance is also always very close to, and in some cases even better than, the best of the other available tools too, supporting the use of this approach to implement effective testing automation in production environments. Although in current experimentation a simple random search heuristic has been applied, we believe that there is a lot of potential to further improve the performance of this approach by designing smarter search based heuristics (e.g. how to select rows, which uncovered pair to recover next, and so on). Furthermore, we use only a simple copy function F_{i-1} which could be substitued by a more complex function F_k in order to initialize every new column in a more efficent way. We are currently investigating this issue, implementing and evaluating several optional local optimization strategies, in order to achieve significant results, and will present the results in the future. As a final remark, work is undergoing to extend this algorithm to support *n-way* testing and constraints over the inputs. In particular, we are working on applying the use of constraint solvers or model checkers to the CIT in the presence of constraints, by combining the techniques proposed in [6, 7] with the technique presented in this paper.

References

[1] Joanne M. Atlee and Michael A. Buckley. A logic-model semantics for SCR software requirements. In *International Symposium on Software Testing and Analysis*. ACM, 1996.

[2] R. C. Bose and K. A. Bush. Orthogonal arrays of strength two and three. *The Annals of Mathematical Statistics*, 23(4):508–524, 1952.

[3] R. Brownlie, J.Prowse, and M.S. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.

[4] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.

[5] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513, October 1998.

[6] Andrea Calvagna and Angelo Gargantini. A logic-based approach to combinatorial testing with constraints. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2008.

[7] Andrea Calvagna and Angelo Gargantini. Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints. In John Rushby and Natarajan Shankar, editors, *AFM'08: Third Workshop on Automated Formal Methods (satellite of CAV)*, pages 43–52, 2008.

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

[9] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *ICSE*, pages 38–48, 2003.

[10] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM Press.

[11] C. J. Colbourn, M.B. Cohen, and R.Turban. A deterministic density algorithm for pairwise interaction coverage. In *IASTED International conference on: Research Techniques (TAIC PART), London, September 2007*, pages 121–130, 2007.

[12] Charles J. Colbourn, Sosina S. Martirosyan, Gary L. Mullen, Dennis Shasha, George B. Sherwood, and Joseph L. Yucas. Products of mixed covering arrays of strength two. *Journal of Combinatorial Designs*, 14:124–138, 2006.

[13] J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.

[14] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott. Model-based testing of a highly programmable system. *issre*, 00:174, 1998.

[15] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering ICSE*, pages 285–295, New York, May 1999. Association for Computing Machinery.

[16] I. S. Dunietz, W. K. Ehrlich, B.D. Szablak, C.L. Mallows, and A. Iannino. Applying design of experiments to software testing. In IEEE/Computer Society, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 205–215, 1997.

[17] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[18] Alan Hartman. Ibm intelligent test case handler: Whitch, http://www.alphaworks.ibm.com/tech/whitch.

[19] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.

[20] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[21] Jenny Combinatorial Tool. http://www.burtleburtle.net/bob/math/jenny.html.

[22] Noritaka Kobayashi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Non-specification-based approaches to logic testing for software. *Journal of Information and Software Technology*, 44(2):113–121, February 2002.

[23] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In IEEE/Computer Society, editor, *27th NASA/IEEE Software Engineering workshop*, pages 91–95, 2002.

[24] D. Richard Kuhn and Vadim Okum. Pseudo-exhaustive testing for software. In *SEW '06: IEEE/-NASA Software Engineering Workshop*, volume 0, pages 153–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[25] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.

[26] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*, pages 254–261. IEEE Computer Society, 1998.

[27] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[28] A. McDowell. All-pairs testing, http://www.mcdowella.demon.co.uk/allpairs.html.

[29] K. Nurmela. Upper bounds for covering arrays by tabu. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.

[30] Pairwise web site. http://www.pairwise.org/.

[31] Gadiel Seroussi and Nader H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.

[32] George B. Sherwood. Optimal and near-optimal mixed covering arrays by column expansion. *Discrete Mathematics*, In Press, Corrected Proof:–.

[33] B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In CO Breckenridge, editor, *Proceedings of the Fifth International conference on Artificial Intelligence Planning Systems (AIPS)*, 2000.

[34] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[35] TestCover tool. http://www.testcover.com/.

[36] Alan W. Williams and Robert L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the 14th International Conference on the Testing of Communicating Systems (TestCom) Berlin, Germany*, pages 283–298, march 2002.

[37] A.W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the 13th International Conference on the Testing of Communicating Systems (TestCom 2000)*, pages 59–74, August 2000.

[38] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.