SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2009; **00**:1–7 (DOI: 10.1002/000)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/000

T-wise combinatorial interaction test suites construction based on coverage inheritance

Andrea Calvagna[1,*], Angelo Gargantini[2]

[1] *Dipartimento di ingegneria informatica e delle telecomunicazioni, University of Catania, Italy*
[2] *Dipartimento di ingegneria dell'informazione e metodi matematici, University of Bergamo, Italy*

SUMMARY

Combinatorial interaction testing (CIT) is a testing technique which requires covering all t-sized tuples of values out of n parameters attributes or properties, modeled after the input parameters or the configuration domain of a system under test. CIT test suites have shown to be very effective in software testing already at *pairwise* ($t = 2$) level, and that effectiveness of CIT grows with the tuple width t. Unfortunately, also does the number of tuples to be tested. In order to reduce the testing effort, researchers addressed the issue of computing *minimal-sized* CIT test suites with effective and scalable algorithms. However, still very few generally applicable t-wise covering construction algorithms (and tools) do exist in literature. This paper presents an original greedy algorithm to compute t-wise covering mixed covering arrays with constant space complexity, irrespective of the number of involved parameters and strength of interaction. The proposed algorithm has been implemented in a prototype tool, featuring also support for user constraints over the inputs. Assessment of the tool performance on a set of large, real-world test systems is reported, with results encouraging its adoption in industrial production environments.

Received day month year; Revised day month year

KEY WORDS: combinatorial interaction testing; mixed covering arrays; parameter based constructions; forbidden tuples.

*Correspondence to: Andrea Calvagna, DIIT - Cittadella Universitaria, Viale A.Doria 6 - 95127, Catania (Italy)
E-mail: andrea.calvagna@unict.it

1. INTRODUCTION

Systematic testing of highly-configurable software systems, e.g. systems with many optional features, can be challenging and expensive due the exponential growth of the number of configurations to be tested with respect to the number of features. As an example, consider the burden of testing the correctness of the GNU *GCC* compiler output, for a given source file, against every possible configuration of the tool's almost two hundreds distinct command-line parameters. The usage of *Combinatorial Interaction Testing* (CIT) technique can improve the effectiveness of the testing activity for these kind of systems, at the only cost of modeling the system's configurations space. In fact, CIT consist in systematically testing all possible *partial* configurations (that is, involving up to a fixed number of parameters only) of the system under test. Moreover, it has been shown that most of the faults in a software system are already triggered by unintended interaction between a relatively low number of input parameters, typically up to six [30]. Thus, modeling the input parameters as a set of features ranging over their most representative or relevant values, and then systematically testing for t-wise coverage, where t relatively low, can be very effective in revealing software defects [28]. 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).

It has been experimentally shown that a test suite covering just all pairs of input values can already detect a large part (typically 50% to 75%) of the faults in a program [40, 18]. Moreover, incorrect behaviors or failures due to unintended feature interaction, detected by CIT, may not be detectable by other more traditional approaches to systematic testing [30, 42]. Dunietz *et al.* [19] compared t-wise CIT 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.

Tools for building *t-wise* CIT test suites aim also at effectively combining all the *t-tuples* of parameter assignments in the smallest possible number of *complete* test cases, that is, tests assigning all parameters. In fact, although t-wise coverage of a system featuring n parameters, each ranging in r values, would require testing of $r^t\binom{n}{t}$ configurations, which is still impractical, in a combinatorial test suite they can be effectively combined together in a much lower number of test cases. This allows significant testing time and cost savings to be achieved, while still having a testing process driven by an effective coverage metric [30, 10]. As an example, for a system with a hundred boolean parameters (2^{100} possible test cases) pairwise coverage would require $2^2\binom{100}{2} = 19800$ pairs of input values to be tested, which a combinatorial test suite can fit into only ten complete (100 values each) test cases. Similarly, pairwise coverage of a system with twenty ten-valued options (10^{20} exhaustive tests) requires coverage of only 19000 pairs, which in turn a combinatorial test suite can cover in only 200 tests cases. Table I reports the input domain model of a simple telephone switch billing system, the Basic Billing System (BBS) [34], 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 requires only 54 pairs, which can be covered in only nine tests cases by the minimal sized test suite, reported

DOI: 10.1002/stvr

Table I. Input domain of a basic billing system for phone calls

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

Table II. Example pairwise test suite for BBS

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

in Table II. In this paper, which extends on our previous conference paper [9], a new *parameter-based* algorithm for the construction of t-wise CIT test suites is presented, falling in the category of *parameter based* approaches, since it computes the test suite incrementally, starting from the first t parameters and then extending it by one column for each additional parameter. Although many different approaches to build CIT test suites one test case at the time already exist in the literature, only a single parameter-based technique is available, by Y. Lei and K.C. Tai [33]. However, note that the proposed algorithm only shares with it the choice to proceed by adding one column at the time. Instead, it introduces the original ideas of *coverage inheritance* and of a recursive routine encoding the sole general principle of coverage preservation. In fact, the basic steps proposed to compute a column for a new parameter are: first, initialize the column in such a way that a nicely known set of tuples gets covered by inheritance; then, for every missing tuple left, a new column assignment is computed recursively, that covers it while preserving existing coverage.

The coverage inheritance principle allows reducing the overhead for computing and storing the list of tuples to be covered, that unfortunately grows exponentially with the size of the considered task, and which is required in order to decide termination of the construction processes based on *greedy* heuristics, like this. The proposed technique has been also extended to support constraints of the type of *forbidden tuples*, that is combinations that are not allowed to appear in the test suite. The technique has also been implemented on a prototype tool, and an assessment of its performance is presented, based on a set of test cases modeled after real world, large-sized software systems, with results comparable to existing state of the art tools.

The paper is structured as follows: Section 2 gives the reader insights on the backgrounds of this work and its related context; Section 3 exposes the main ideas behind the proposed construction

DOI: 10.1002/stvr

technique; Section 4 presents the pseudocode of an algorithm implementing the proposed construction; Section 5 thoroughly discusses the points of originality and the peculiarities of the proposed approach, also presenting a detailed computational complexity analysis; Section 6 presents an original solution to extend the approach in order to support tasks with constraints; Section 7 presents an implementation of the algorithm and discusses the results of a comparative evaluation with existing similar tools. Eventually, section 8 draws our conclusive statements and directions for future work.

2. BACKGROUND

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 [24]. Covering arrays are combinatorial structures which extend the notion of *orthogonal arrays* [2], which in turn are generalizations of *latin squares*. For a given system under test exposing n input parameters (or features), all ranging in r distinct symbols, 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 testing only the case when $\lambda = 1$ is of interest, that is, where every t-tuple is covered at least once. Note that this the fact that at least one instance of each tuple is in the final suite does not require that each tuples is covered strictly the same number of times. Moreover, for testing of real systems one is 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 input parameter 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 inputs, while each column of the CA will list all values chosen for each input.

Devising a general algorithm to compute a minimal sized set of test cases that satisfies t-wise coverage is a non trivial problem, since computing a minimal CA is NP-hard [41, 37]. Many tools and techniques for building *t-wise* CIT test suites have already been developed and are currently applied in practice [10, 40, 3, 39, 29, 17, 30]. Grindal et al. counts more than 40 papers and 10 strategies in their survey [23]. There is also a web site [36] devoted to this subject and several automatic tools are commercially or freely available. For this reason, and for the inner complexity of the task, even small improvements over currently available best performances seem hard to accomplish.

Solutions exist in the literature to compute covering arrays algebraically [27], but these techniques are applicable to tasks with an homogeneous alphabet of symbols only, and/or that restrictions apply on the task geometry (t, n, r). As a consequence, researchers have addressed the issue of designing general solutions to compute minimal sized CIT test suites based on *greedy* heuristics searches [4]. Heuristic-based approaches have the great advantage of being generally applicable to the construction of MCAs without any limitation, but the efficiency of specific algorithm itself, at the cost of possibly near-optimal performance in terms of the resulting CIT test suite size. In fact, typically only an upper bound on the size of constructed suite may be guaranteed. Less traditional *meta-heuristics* algorithms have also been proposed [11, 35], based on bio-inspired techniques, such as genetic algorithms, or the simulated annealing process used by the mAETG algorithm [22], in order to converge to a near-optimal solution after an acceptable number of iterations. In addition, *recursive* construction techniques do exist [15, 38], 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.

DOI: 10.1002/stvr

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 [10]), many variants of the *AETG-like* strategy have been already proposed in literature, in contrast only one *parameter-based* approach exist, which is IPO by Y.Lei and K.C.Tai [40], which makes this point worth investigating. Moreover, very few of the actual tools have the key features which would enable their useful integration into industrial development/testing environments, that is, support for higher interaction degrees, support for constraints, general applicability to any MCAs and not only CAs, and an efficient usage of the available computational resources. A more recent paper by Y. Lei et al. [31] poses attention on improving the usability of these tools into real testing environments, addressing the issue of scalability, with respect to increasing task sizes and interaction degrees. The authors propose a new technique which reduce the computation times with respect to their original technique, but at the costly trade-off of significantly increased size of the built test suites. It is interesting to note that only three non commercial tools, out of a far larger number of existing tools, were found eligible for comparison, as featuring support for t-wise MCA construction, among which only two of them where really fairly comparable in terms of performance: FireEye (currently renamed ACTS [32], featuring variants of the IPO algorithm), the ITCH tool from IBM (whose performance was however not competitive), and one poorly documented greedy tool named Jenny [26], which is the evidence that this area of research is far from being exhausted, and more investigation is needed in the context of t-wise MCA construction tools.

This paper presents an original t-wise MCA construction technique falling in the category of *greedy*, *parameter based* approaches, and implemented it in a tool which has been then tested and compared to the best performing available tools with analogous features. It is important to highlight that while our technique of course may share with other algorithms the *parameter based* background, on the other hand, it has introduced some original ideas and improvements in that context, which will be exposed in details in the following sections.

3. COVERAGE INHERITANCE

The approach proposed in this paper is inspired by one property of CAs, which is that they are *symmetric* with respect to *switching* of any two columns. In fact, column ordering in a CA is irrelevant, so the symbol assignment of one specific parameter (a column), could be safely exchanged with that of any other parameter (another column), still having a valid CA. This holds also for MCAs, if the switch is between parameters with the same alphabet size (range). In the following it is shown how it is possible to take advantage of this property in the context of an MCA construction, to algebraically compute the initial values of a column added for a new parameter in order to quickly achieve coverage of a known subset of its required tuples.

Consider a system under test which has n input parameters $(p_1, \ldots, p_n)$, and assume that p_k has range r_k and values in $P_k = \{0, 1, ..., r_k - 1\}$, that is $P = P_1 \cup \cdots \cup P_n$ is the whole input domain. Note that actual symbols of every parameter have been mapped to an equivalent set of symbols (natural numbers) for convenience. Assume that the first $j - 1$ $(t \leq j \leq n)$ parameters have already been combined into a t-wise mixed covering array, and that one wants to extend the covering array with an

DOI: 10.1002/stvr

additional column in order to include one more parameter p_j. Assuming that the system parameters will be processed in not-ascending ranges order, that is:

$$r_{k-1} \geq r_k, \forall k \in [2..n].$$

In this case, it will always be possible to choose k among the previous and define a *mapping* which associates each symbol of parameter p_j with a distinct symbol of parameter p_k, and the *don't care* value x with any left symbol of p_k.

As an example, a straightforward mapping is to copy the values of the k-th parameter into the new column p_j, except for those out of range, which are assigned to x. However, it is important to highlight that many mappings are possible, even different for each added column. By initializing the new column by such mappings, the extended MCA will already cover all of the additionally required tuples, except for those that include both p_j and p_k. This *coverage inheritance* property can be easily proved by considering that the previous j-1 parameters, including p_k, were already combined with each other in the test suite, by hypothesis. Thus, as long as the assignment of p_j reproduces that of p_k all tuples with p_j and the same parameters, but p_k, will be covered too. Note that after this initialization the number of tuples still required to be covered for the new parameter have been greatly reduced then, yet without any *greedy* processing. In fact, to only the additional tuples including both p_j and p_k, which have not been inherited.

The tuples left to be covered will then be added to the CIT test suite on a one by one basis, until complete coverage is achieved. Many strategies can be adopted to order the processing of these tuples, affecting the overall process performance, but not its successful termination. To create a given missing tuple, one entry of the new column is changed accordingly. Note that, in this construction, changes take place in the entries of the new column only. In case a change deletes any required tuples previously existing in that row, then these will be restored by changing some other suitable column entries, too, so that existing coverage is also preserved. Note that this is a recursive task, but its termination is guaranteed as long as an upper bound has been put on its depth, e.g. by means of allowing each entry to be edited only once, per processed tuple. When no more suitable entries are left to recover a tuple, a new row will be added at the end of the test suite, in order to host it and terminate the recursion.

What's more important, several entries in the column might be suitable to host the same change, but these might not be equivalent in terms of the path of changes they would induce, and additional rows they would trigger. Thus, a smart strategy is highly desirable in order to select the best entry where to apply next change, as this will significantly affect the algorithm performance. However, any strategy can be safely applied, since it will not impact the correctness of the built covering array. In fact, the core recursive task of column change implements the sole general principle that any manipulation applied to the column must not decrease the coverage.

To summarize, adding a new column for parameter p_j to the covering array consists of the following six steps:

1. Choose an existing column, k (e.g. k=j-1) and initialize the column of values for p_j by mapping (e.g. copying in modulo) the column for p_k, to create a starting point where all tuples of p_j and the previous parameters, except those including both p_j and p_k, are covered by inheritance
2. for each the other missing tuple, do the following:
3. initialize a set of flags, one for each position in the column, indicating that it can be changed.
4. change one of the available values of p_j to increase coverage with respect to p_k and flag the position as changed.

DOI: 10.1002/stvr

5. If this manipulation destroys inherited tuples, recursively make further manipulations of unchanged values of p_j until original coverage is fully restored in some other rows (and mark each edited row as changed).
6. When there are no more unchanged values left to restore a tuple, add it as a new row and terminate recursion.

To conclude this section, note that the whole construction process does not define only a single specific heuristic. It is also a framework operating a separation of concerns, where the validity of the result is ensured by the framework, and the performance depends on the efficiency of external selection strategies adopted to complement it, among the variety of those applicable in order to choose *what* missing tuple to add next and *where* in the column it is best to make a change to create it.

3.1. Numerical example

Let's clarify the approach with an example system with four parameters p_1, p_2, and $p_3 \in \{0,1,2\}$, and $p_4 \in \{0,1\}$, for which a pairwise covering test suite is built. First of all, p_1 and p_2 are combined together to get an initial test suite of nine rows. At next iteration the third column p_3 is then initialized by mapping from p_2, assuming that in this example each new column p_j will be set by copying exactly the previous, p_{j-1}, except for out of range values that will be set to x.

Since p_2 was already fully combined with p_1, and p_3 has same number of symbols of p_2, then p_3 will be already fully combined with p_1 too (see figure 1-(a)). Then, coverage of the pairs between p_3 and p_2 only, which is incomplete, need to be explicitly processed. Note that some of the pairs between p_3 and p_2 are already covered by construction: $\{(0,0),(1,1),(2,2)\}$, and that they are (three times) redundant. The pairs to be covered are $\{(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)\}$ and six redundant column entries are available to this aim, among which to choose. A simple criteria applicable here is to edit the symbol in the row position of first redundant instance of a pair. In this case, one can choose to create the missing pair $(p_2,p_3)=(0,1)$ by changing the value in fourth row. Since this will delete the unique pair $(p_1,p_3)=(1,0)$, it needs to be restored elsewhere. This can be done by changing last column entry in sixth row, from 2 to 0. This last induced change deletes in turn the existing pair $(p_1,p_3)=(1,2)$, then immediately restored by changing fifth row redundant entry to 2, which also increases (p_2,p_3) coverage by additional pairs $(2,0)$ and $(1,2)$. Similarly, missing pair $(p_2,p_3)=(0,2)$ is added by changing value in seventh row from 0 to 2, which deletes pair $(p_1,p_3)=(2,0)$ as a side-effect. This latter is then restored by changing the entry in the eighth row from 1 to 0, which also creates the missing pair $(p_2,p_3)=(1,0)$.

Eventually, the last missing pair $(p_2,p_3)=(2,1)$ is obtained by changing p_3 entry ninth row, without any side-effect. This terminates the first iteration, since pairing of p_3 and p_2 is now complete and 100% pairwise coverage between p_3 and p_1 has been restored.(see figure 1-(b)). A further iteration is needed to add the last parameter p_4 to the suite, which has smaller range than the previous. Again, the fourth column's values are copied from current values in adjacent column p_3 (see figure 1-(c)) and then edited to complete the coverage. Since p_3 has higher range than p_4 this time x (don't care) values will appear in unmatched entries. As shown in figure 1-(c), after initialization only the set of pairs $(p_3,p_4)=\{(1,0),(0,1),(2,0),(2,1)\}$ is missing to reach complete coverage. The first pair is created changing p_4 value in fourth row, and restoring deleted pairs $(p_2,p_4)=(0,1)$ and $(p_1,p_4)=(1,1)$ by replacing with 1 the x in seventh and fifth row respectively. This also adds

DOI: 10.1002/stvr

p_1	p_2	$\mathbf{p_3}$
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)

p_1	p_2	p_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)

p_1	p_2	p_3	$\mathbf{p_4}$
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)

p_1	p_2	p_3	p_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: $3^3 2^1$.

missing pairs $(p_3, p_4) = (2, 1)$, without deleting any previous pair. Moreover, creating missing pair $(p_3, p_4) = (0, 1)$ by changing to 1 the sixth row entry, and eventually changing the x in third row to 0 in order to create the last missing pair $(p_3, p_4) = (2, 0)$, also do not produce any side effects. As no more parameters need to be added, the construction process is complete (figure 1-(d)). Please note that modifications have occurred at each iteration only in the new column of the test suite, and that a value can be changed only if it has not been changed already. This requirement ensure that the processing of each missing pair always ends up with a coverage increase, as previously added pairs cannot be undone, but only moved elsewhere. On the one hand, this property guarantees that the overall process eventually terminates. On the other hand, this safety property can lead to sub-optimal results. In fact, in more complex tasks it may also happen that the sequence of changes generated by the adopted row selection strategy leads to a point where no more entries in the column are available to host next required change. In this case a new row has to be added to the test suite to host the missing pair, even if the number of rows was still theoretically enough to host all pairs. The reader might want to refer to Colbourn [14] for a comprehensive listing of known minimal CA size requirements at varying task sizes and strengths. Unfortunately, to the best of our knowledge, there is not a comparable reference systematically listing known sizes for MCAs also, which is the context of this research, but some results can be collected from researching the literature and a few websites dedicated to combinatorial testing research and/or commercial tools.

4. ALGORITHM PSEUDOCODE

Assume that the n input parameters have been sorted in non-ascending range order, and the MCA has been initialized at start-up to an array covering all the t-tuples of the first t parameters, $P_1 \times P_2 \cdots \times P_t$, by simply putting one value combination per row. Pseudocode for the main *extend()* algorithm is shown

DOI: 10.1002/stvr

in Listing. 1. Given $t \leq j \leq n$, an index $1 \leq k \leq j$ of choice, and a map $f_j : P_k \rightarrow P_j \cup \{x\}$, it adds one column to the MCA, in order to cover the additional j-th parameter in the list, assuming it already covers the previous $j-1$. This algorithm is structured in two basic steps: the first consists in initializing the entries in column j, mapping by means of f the corresponding symbols in column k, so to inherit partial coverage. The second step is a loop of calls to function *recover()*, whose pseudocode is shown in Listing 2. Given a missing tuple, and the input MCA, the *recover()* function will return a manipulated MCA additionally covering at least this new tuple. The set M is the set of t-wise tuples of parameters whose coverage won't be ensured by inheritance, and is given by combining all value pairs of p_k and p_j with the set C of $(t-2)$-tuples of values from the other $j-2$ parameters: $M = C \times P_k \times P_j$. Note that C is in turn the union of exactly $\binom{t-2}{j-2}$ set of exhaustive enumerations of tuples of values, which are all already covered by the input MCA. Hence, their enumeration does not really have to be computed nor requires additional storage: they are listed in the MCA. At line six of Listing 1 a reset of the $mark$ flag array is performed. This is required in order to correctly determine a termination condition for the core algorithm *recover()*, as explained more in detail in the following.

Listing 1. Incremental extension algorithm.

```
1  extend(MCA, j, k) {
2      MCA[ ][j] := f_j(MCA[ ][k]);
3      for each m ∈ M {
4          if m ∉ MCA {
5              mark[ ] := false;
6              recover(m);
7          }
8      }
9  }
```

It is important to highlight that the set element m actually consists in a listing of t input symbols *plus* a listing of t column/parameter indexes to whom they are bound: $m = (v_1, v_2, \ldots v_t)@(c_1, c_2, \ldots c_t)$, which means $p_{c_1} = v_1$, $p_{c_2} = v_2$, and so on. Also note that by construction, all m in M always include a binding for column j, the last column in current MCA, and the only one whose entries shall be modified. Without loss of generality, one can assume that this binding is specified in the last position of m, that is, c_t is always set to j, and so p_j shall be set to v_t.

4.1. Recursive tuple recovering

The *recover()* function creates the inputted t-tuple m by recursive manipulations of the j-th column of the MCA. To this aim, it looks for a suitable row in the MCA, that is, one whose value assignment matches all the values-to-columns bindings specified by m, but the last one (that is, that for p_j), which will be ignored. Any x values in the MCA contributes also to a possible match, as it can always be safely changed to a desired value without decreasing the coverage. The missing tuple can then be created in such row by replacing the entry in column j with the last value of m, v_t. If many of such rows are available, a heuristic of choice can be applied to select one of them. If no suitable row is available instead, then a new row will be added to the MCA, containing only the tuple specified by m, and x values in the other entries. If the change applied to the select entry of column j deletes the last

DOI: 10.1002/stvr

occurrence of any tuple in that row, the same procedure will be recursively called to rebuild each of them.

Of course, to avoid triggering an infinite loop of changes, this shall happen in some different rows of the MCA. To this aim, an array $mark$ of boolean flags keeps track of the entries of column j already modified once in this recursion. Flagged rows will then be skipped, that is, they will not be suitable for selection, for the rest of the recursion. As the number N of rows in the MCA is finite, the recursion will always terminate. Moreover, each outer call to *recover()* (in the *extend()* procedure) in overall will strictly increases the MCA coverage by one tuple at least: the one specified by m plus any additional tuple possibly created as a side-effect of the operated column manipulation.

Pseudocode of algorithm *recover()* is shown in Listing. 2 and consists in few basic steps: select a row suitable to the creation of m (line 2), or add a new row for it, if none is left (line 3); save the value in last column (recall $c_t = j$, always) of the selected row (line 4), before changing it to create m (line 5), and flag the row as changed (line 6); eventually, list any tuple possibly deleted by the applied change (lines 7 and 8), and restore it by means of recursive call to self (line 9).

Listing 2. Conservative column change algorithm.

```
1  recover(m = (v_1, v_2, ..., v_t)@(c_1, c_2, ..., c_t)) {
2    choose a row i s.t. mark[i]=false ∧ MCA[i][c_x] = v_x ∀x ∈ [1..t−1];
3    if none {add m as new row and return;}
4    v'_t := MCA[i][c_t];
5    MCA[i][c_t] := v_t;
6    mark[i] := true;
7    for each parameter tuple (c'_1, c'_2, ..., c'_{t−1}) out of {p_1, p_2, ..., p_{j−1}} do {
8      m' := (MCA[i][c'_1],MCA[i][c'_2], ...,MCA[i][c'_{t−1}], v'_t)@(c'_1, c'_2, ..., c'_{t−1}, c_t);
9      if m' ∉ MCA then recover(m');
10   }
11 }
```

5. ALGORITHM DISCUSSION

In previous sections, an original t-wise MCA construction technique has been presented that falls in the category of *greedy*, *parameter based* approaches. It is important to highlight that while our technique of course shares with IPO the *parameter based* background, on the other hand, it has been developed independently and introduces some original ideas, which clearly differentiate it from IPO and the existing t-wise, greedy MCA construction techniques, in general. In the following, these novel aspects will be exposed in deep details.

5.1. Construction framework

It is easy to see that the construction algorithm encoded by *extend()* and *recover()* is not just one single parameter-based heuristic, but rather a framework of parameter-based constructions, operating a clear separation of concerns between the problem of building a valid t-wise MCA one column at a time,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

solved by the algorithm, and the problem of minimizing its final size, which relies on the external strategies adopted to complement it. Note that whatever the strategies applied in order to choose *what* missing tuple to add next, and *where* in the column it is best to make a change to create it, the algorithm produces a valid MCA.

Interestingly, many symbol mapping alternatives have been tested during the implementation of a prototype, but, as expected, a limited impact on the overall resulting performance has been observed. In fact, MCAs are also symmetric with respect to switching any two parameter symbols. In the prototype used for the evaluation presented in section 7, copying by modulo has been our final choice. This in fact produces a column assignment which has the same distribution of symbol occurrences of the copied column, which in a CA tends to be even, in spite of a possibly smaller range of the new parameter. At the same time, by always mapping to actual symbols instead of don't care values, a higher number of redundant tuples is produced. This reduces the probability of deleting last occurrence of a covered tuple, and to have then to restore it with additional calls to *recover()*, which is where rows eventually might be added.

Conversely, the choice of the column from which to initialize shown to be of significant impact on the overall performance. Specifically, the observed performance was improved if copying from the very previous column j-1, thus ensuring the smallest possible difference between the ranges of the new and mapped parameters and, as a consequence, the smallest number of redundant symbols in the resulting column initialization. Moreover, since the rows of the MCA suitable to apply a given required change might be more than one, the final size of the whole MCA will mostly depend on the heuristic adopted to select the best row to apply that change, in order to avoid unnecessary rows addition to the suite. Unfortunately, to make an optimal choice, one should be able to predict in advance the overall number of rows addition that each choice would trigger, in overall, up to the process completion, which is an entire tree of possible editing sequences. In the evaluation presented in section 7 of this paper a simple heuristic is adopted based on randomly selecting among the suitable rows, which shown to be already effective in driving good performances, and is presented in section 7.1.

5.2. Time and space complexity

As several heuristics can be devised to be used in the proposed construction, with varying effects on the performance but not on the validity of the construction process, their corresponding steps in the algorithm pseudocode are shown as abstract computations, and a complexity analysis of the construction algorithm is derived assuming that $O(1)$ (i.e. random) heuristics have been adopted in both *extend()* and *recover()*. It is well known that the number of tests N required for pairwise coverage in a CA grows at most logarithmically in n and quadratically in r, with n the number of parameters and r the number of symbols. Assume here for the sake of simplicity that all features have same range r, but it is easy to extend this result to MCAs by averaging over the different ranges, and also to a generic strength of interaction t, by writing it as $O(r^t \log(n))$ (see [10]). The regression analysis on data shown in tables of section 7 confirmed the consistency of our tool performances with this theoretical result.

Computational time complexity of *extend()* algorithm is dominated by the loop of calls to the *recover()* function, which is bound to the size of the *missing* tuples set M. Recalling that $M = C \times P_k \times P_j$, and that C is the union of $\binom{n-2}{t-2}$ set of tuples of size r^{t-2} each, then $|M| = r^{t-2}\binom{n-2}{t-2}r^2 = r^t\binom{n-2}{t-2}$. Since, as it can be easily derived, $\binom{n-2}{t-2} \leq n^{t-2}$, the time complexity is

DOI: 10.1002/stvr

then in $O(|M|) = O(r^t n^{t-2})$. It must be noted also that the set M really requires to be enumerated only, in contrast with IPO which requires storing it also, as part of the state of the algorithm. As a consequence, with our approach it is also possible to generate only one tuple at the time, and process it on the fly, with space requirements in $O(1)$.

Alternatively, the set M has the storage cost of the pairs of two parameters only, $P_k \times Pj$, that is in $O(r^2)$. In fact, C is a set of tuples already covered (that is, stored) in the MCA, and thus with no additional storage requirements. However, in this case the tuples shall be read by running down the N rows of the MCA, which includes redundancy, so the time complexity grows to $O(Nn^{t-2}r^2) = O(r^{t+2}n^{t-2}\log n)$. Either ways, the space requirements of the *extend()* algorithm are limited to the size of the flag array $mark$, which is in $O(N)$ and supersedes $O(r^2)$. Similarly, the complexity of algorithm *recover()* is dominated by the loop that checks and recovers any deleted tuples. This is run for a set of tuples whose size is $\binom{n-1}{t-1}$, and which can be easily computed only once in advance. Each iteration of this loop could trigger a recursive call, but this can occur N times in overall. In fact, recursion can be induced only when a row has been modified, which can be happen only once per row. Moreover, determining if a tuple is covered requires checking it against the MCA, whose size is in $O(N)$. As a consequence, the computational complexity of the *recover()* algorithm is in $O(N^2 n^{t-1}) = O(r^{2t} n^{t-1} \log^2 n)$. On the other hand, the space complexity is in $O(N)$, as long as each recursion level requires storing only the one tuple it is currently checking, and this requirement supersedes the storage requirements for the $\binom{n-1}{t-1}$ combinations to check. In conclusion, the total time complexity of the whole MCA extension process will be in $O(r^{3t} n^{2t-3} \log^3 n)$, if missing tuples are generated on the fly, or in $O(r^{3t+2} n^{2t-3} \log^3 n)$ if tuples are generated by reading the MCA. In both cases, the overall space complexity is in $O(r^t \log n)$, which is the order of size of the output itself. Note that the time complexity of the proposed construction algorithm is greater than many others (see also in the following), but this does not limit its applicability to real world examples and it is balanced by a lower space complexity and by a more compactness in the covering arrays as shown in section 7.

5.3. Contrasting with IPO

While IPO heuristic consists in sequentially filling in values in the new column by applying a global optimization criterion, which is the coverage increase achieved by each choice, as a key difference, this paper presents a general construction algorithm which has shown not to rely on any specific selection criterion to guarantee successful building of a valid CIT test suite. Of course, smarter heuristics will produce better overall performance, and will be the subject of further investigation in our future work. However, in this paper, in order to avoid biasing the performance evaluation of the core algorithm, the performance evaluation shown in section 7 has been based on *random* heuristics.

Moreover, the original IPO algorithm requires enumerating and storing the set of tuples to be covered for an additional parameter entirely. Even if later evolutions of the IPO algorithm included optimizations that improved this huge storage requirement, it still is proportional to the (exponentially growing) size of required tuples. As a key difference, in this construction technique the new idea of *coverage inheritance* is introduced in order to reduce the tuples enumeration and storage requirements to only a fraction of that whole set, whose size is constant irrespective from the number of system parameters. In fact, the number of tuples that have to be checked to extend the coverage to one additional parameter is reduced to the t-tuples including two given parameters, the new one p_j and

DOI: 10.1002/stvr

the mapped one p_k. Note that this set of tuples can be seen as the Cartesian product of the set of pairs $P_k \times P_j$ with the set of (t-2)-tuples of the other j-2 parameters. By construction, these (t-2)-tuples are all already listed in the test suite rows (of course, with some redundant occurrences), so their enumeration don't need to be computed at all, but only that of the product $P_k \times P_j$. As a consequence, the additional time and space required to enumerate and store t-tuples is dramatically limited to a constant set of pairs, irrespective of the number of involved parameters and the interaction degree. Of course, the number of (t-2)-tuples to be read out from the test suite is still proportional to t and n, but avoiding their separate computation and storage will save significant time and memory, as explained in detail in section 5.2.

In addition, section 4 explained that the proposed approach requires sorting the parameters by range, in order to enable the application of coverage inheritance. Note that parameter sorting is not a requirement for IPO, instead. In fact, it was not in the original pairwise IPO algorithm [33], but has been added as an improvement in later work presenting its t-wise generalization, IPOG [31]. In fact, experiments show that the performances of parameter-based constructions benefit from giving precedence to parameters with highest ranges. The reason for that being, intuitively, that they require a larger set of rows to host their tuples. This larger MCA lets the following parameters, with lower ranges and thus less tuples to cover, find more room already available to allow an easier hosting of their tuples, delaying (at least) the need for additional rows. As a last conceptual difference, note that IPO proceeds sequentially by computing each entry in the new column, while the proposed algorithm proceeds sequentially by adding one tuple (at least) at the time, each of which might require changes to many column entries.

Finally, the space and time complexity of IPOG are in $O(r^t n^{t-1})$ and $O(r^{t+1} n^{t-1} \log n)$ respectively. In contrast, our algorithm has much better $O(r^t \log n)$ space complexity, at the cost of higher time complexity of $O(r^{3t} n^{2t-3} \log^3 n))$, as is derived in section 5.2. This is due to the fact that determining if a tuple is covered is performed in IPOG with time complexity in $O(1)$, while our algorithm has to check it against the MCA, whose size is in $O(r^t \log n)$. On the other hand, although better time complexity is achieved by IPOG, this is obtained by means of a larger state space, actually a flag for each tuple indicating if it is already covered or not, whose size is proportional to the whole set of tuples required for coverage. In contrast, our algorithm requires only an additional flag for each row of the MCA. In order to deal with the problem of the exponentially growing space requirements in IPOG, in Lei et al. presented a also a variant, IPOG-D,in which the number of tuples that have to be explicit enumerated is dramatically reduced, but at the cost of significantly degraded performance in terms of MCA sizes [31]. Moreover, the time and space complexities of IPOG-D remains the same of the parent algorithm IPOG. On the contrary, as shown in section 7, our tool succeeds in having much lower space requirements while achieving at the same time significantly better MCA size performance than IPOG-D, actually comparable to that of IPOG.

6. CONSTRAINT SUPPORT

The CIT construction algorithm presented in section 4 has been also extended to support constraints over the inputs, in the form of *forbidden tuples*. Although constraints can also be conveniently expressed with more abstract and compact notations, such as logical expressions [21, 6], in this paper the simpler approach has been preferred, consisting in inputing an explicit list of forbidden

DOI: 10.1002/stvr

combinations, leaving more sophisticated solutions as future enhancements, and focusing only on showing that it is possible to support constraints with the proposed construction technique. A forbidden tuple is an assignment to a set of input parameters, which is not allowed by the user to appear in the rows of the built MCA. Thus, a generic forbidden tuple

$$p_1 = v_1, p_2 = v_2, \ldots p_x = v_x$$

implies satisfying a logical constraint of the type:

$$p_1 \neq v_1 \vee p_2 \neq v_2 \vee \ldots p_x \neq v_x.$$

Assume that the forbidden tuples are internally sorted, that is, the involved parameters are always listed in the same order as they appear in the MCA, and let AV be the set of all forbidden tuples. It is important to remark that a forbidden tuple is an assignment of any number of parameters, which can then be less, equal or higher than the size t of the required tuples.

It can be easily observed that in order to comply with the whole constraint, it is enough to satisfy only one of the logical terms in order, e.g. the last one. Of course, an MCA under construction of its j-th parameter can only comply with constraints which are already *completely* expressed, that is, that involve parameters up to P_j, at most. At the same time, all constraints which involve parameters up to P_{j-1} shall have been already satisfied by the MCA in previous extension loops. As a consequence, at this loop it is only required to comply with constraints which do have parameter P_j (in particular, as their last parameter). The subset AV_j of forbidden tuples involving parameter j, that can be easily extracted from AV at each MCA extension loop. In conclusion, it will be possible to comply with this subset of constraints by simply taking them into account opportunely when editing values in the entries of column j only. Details of the constraint supporting, modified MCA extension process, are shown in Listing 3, and commented in the following.

Listing 3. Extension algorithm modified in order to support constraints.

```
1  extend(MCA, j, k) {
2      MCA[ ][j] := f_j(MCA[ ][k]);
3      move to M all tuples in R;
4      for each row i in the MCA
5          if this row matches any av ∈ AV_j {
6              MCA[i][j] = x; // remove the violation
7              add to M all the t−tuples with P_j in that row;
8          }
9      for each m ∈ M {
10         if m ∉ MCA {
11             if m matches any av ∈ AV_j { // inherently illegal tuple
12                 set last parameter of m from P_j to P_{j+1};
13                 set last value of m from v_j to f_{j+1}(v_j);
14                 save m in R;
15             }else{
16                 mark[ ] := false;
17                 recover(m);
```


DOI: 10.1002/stvr

```
18                }// else
19            }// if
20        }// for
21    }
```

The modified algorithm starts with the usual initialization of the new column, to inherit a partial coverage, at line 2. The main differences with the unconstrained version of the algorithm are the additional loop at line 4, and the additional check at line 11. The new loop is required to ensure that the values of column j assigned by initialization have not created any forbidden tuple, in any row. If that happens, each interested row is turned to a legal row by temporarily setting its column j entry to x. Later on, in the construction process, if this x needs to be set again to an actual symbol, it shall only be a *legal* value, that is, such that the resulting row complies with the constraints. The tuples in the row that have been deleted by putting the x are added to the set M of the (potentially) missing tuples, so that they will be eventually restored, if still missing, in the loop at line 9. In this loop, the only modification with respect to the unconstrained version is an additional check to remove the t-tuples which are inherently illegal, that is, they just contain some forbidden tuple inside, thus cannot be present in the suite. However, missing any tuples from the MCA, due to the constraints, means that the extension process for next column $j+1$ will start from an incomplete MCA. In fact, some symbol occurrence will actually be omitted from column j. As a consequence, when initializing column $j+1$, its *mapped* symbol will be missing also. In other words, the inherited tuples set will not include the tuples that would be derived by *mapping* those that are missing, of course.

In a sense, it can be said that, the constraints applied to one column would be *inherited* by next column, too. To easily overcame this problem, any tuples found to be illegal will be mapped in advance by updating them with the next column's index P_{j+1} and mapped value $f_{j+1}(v_j)$. These tuples are then saved into a set R which will be reported to the next column extension loop, and then processed normally. Consequently, at line 3 of the modified extension algorithm, the set R with the tuples generated by previous extension loop is first emptied, by moving its tuples into M, and then filled in again for next loop.

Since the only modifications to the starting assignment of column j are possible by means of the *recover()* algorithm, this algorithm need to be slightly adjusted too, in order to take into account the constraints. Specifically, it has to ensure that any change to a row won't ever create an illegal combination of values in a that row. Nevertheless, this does not require significative changes in the algorithm presented in Listing 1, but only to enhance internally its row selection heuristic, that is the step at line 2 of the algorithm. In fact, besides choosing a $row\ i$ not yet modified ($mark[i]{=}false$), and compatible with the required change (MCA$[i][c_x] = v_x\ \forall x \in [1..t\text{-}1]$), in addition it shall be such that its change does not forms any illegal tuple $av \in AV_x, \forall x \in [1..j]$.

Note that before running the *extend()* algorithm, in the very first step of the whole constrained construction process, when the startup MCA is initialized with the t-tuples of the first t parameters, an additional checkup is needed too, in order to ensure that each of these combinations is not matching any constraint, so to start with a valid MCA setup. Then, the modified *extend()* algorithm can be safely invoked for the additional columns. The solution presented in this section to support constraints in the form of forbidden tuples has been also implemented into our tool, and some experiments are reported in table IV of next section.

DOI: 10.1002/stvr

7. IMPLEMENTATION AND ASSESSMENT

In order to assess the performance capabilities of the proposed construction algorithm, a prototype tool have been implemented, called Ttuples, and applied to a set of tasks modeled after real-world software systems. This allowed having a figure of its actual performance, and also to contrast it to that of other tools with similar features. [†]

In order to do this, a concrete heuristic has been adopted to complement the general construction algorithm. This section presents in details the setup of this comparison, including the selected heuristic, compared tools and applied case studies, and draw some observations on the collected results.

7.1. Multipass Heuristic

Many different ordering strategies can be applied in order to sequence the missing tuples in the *extend()* algorithm. This section presents the *multipass* processing strategy, that is adopted in the current algorithm implementation as a smarter policy over the simpler random selection strategy. The *multipass* selection heuristic is based on the observation that each invocation of the *recover()* function by definition will always increase the current test suite coverage, but at the possible cost of adding one or more rows to the current suite. As long as the objective is to minimize the size of the built MCA then one should give priority to the creation of tuples triggering no MCA rows additions. Another round of calls can then be run for the remaining tuples, on the new MCA resulting after the first round is complete. Specifically, in the first round, a rollback of the state is performed if an MCA size increase is observed after processing a tuple. In that case, the tuple will be saved to be processed again in next round. This *filtered* processing can be reiterated for many rounds, unless al least one tuple is successfully added at each round, but in the present work it has been limited to two rounds only. In fact, this heuristic increases the algorithm running time with respect to the *random* policy, as a tuple could be processed as many times as the number of processing rounds, that is, in this case, twice, at most. On the other hand, experiments show that this heuristic outperforms the simpler random heuristic. This can be explained considering that for each tuple *successfully* added in the first round, at no cost in terms of size increase, a modification in the MCA is operated, producing a new MCA where some of the *second round* tuples could have been indirectly created, or their successful insertion now could be possible, maybe. Recall that the *recover()* algorithm will always increase the coverage. Moreover, the time performance of our construction will benefit exponentially by any improvement in the MCA size, which helps balancing the cost of any additional processing. As a final remark, note that in both rounds tuples are still selected for processing randomly, thus the tool shows non deterministic behavior.

7.2. Compared Tools

In the experiments shown next in this section, a set of tasks has been applied to our tool and also to other tools which have been selected as having comparable features, among the state of the art tools for CIT. It is interesting to note that while a large number of tools for pairwise testing do

[†]The Ttuples tool is available for download at http://www.diit.unict.it/ acalva.

DOI: 10.1002/stvr

exist, very few tools have been developed which support t-wise combinatorial testing, and constraints over the input, instead. Moreover, among these, even fewer are those actually open source or freely available for academic purposes. Thus, our selection of tools includes actually the only other tool implementing parameter-based constructions, that is ACTS (formerly called FireEye) [32], the *CASA* tool [13, 22], which implements a meta-heuristic algorithm and is also a tool with non deterministic behavior, just like Ttuples, and an open source, greedy construction tool freely available on the web, Jenny [26]. Note that actually ACTS is a big Java jar bundling implementations of many construction algorithms. Specifically, it includes all the three variants of the IPO algorithm: IPOG, IPO-F and IPOG-D. IPOG implements the generalization to t-wise testing of the original IPO algorithm. This algorithm explicitly enumerates all possible combinations, and does not scale well to big tasks, where the number of combinations is large, or when resources are constrained. IPO-F is a variant of IPOG whose implementation has been optimized for speed [20] which succeeds also in achieving better performance than IPOG. IPOG-D is a variant of IPOG, incorporating a recursive technique, namely the *doubling-construct*, and developed just to address the problem of reducing the number of combinations that have to be enumerated, so to improve the algorithm scalability. Note how this latter algorithm has been designed with similar intent of ours.

All these cited algorithms are deterministic in their behavior, that is, the same input will produce the same output. On the other hand, Ttuples due to its usage of randomness has non deterministic behavior. It is common practice for non deterministic tools in the literature (see [10], [12], [1], [16]) to be benchmarked over a reasonably sized series of executions. Consequently, data reported for Ttuples and CASA in the following tables are the average out of a set of one hundred runs, or one hour of running time, at most. The best (lowest) MCA size recorded is also reported next to the average, in brackets, for convenience. To make the comparison fair, the other tools were also subject to comply to the same limitation on the overall running time. Tests have all been run on a 2.4GHz Macbook Pro equipped with 2GB of RAM.

7.3. Real-world Models

Test suite sizes have been computed for a set of five system specifications which are actual models of well known, real-world software systems. It is important to remark that the performance comparison presented in the following aims only at giving to the readers a figure of the actual capabilities of our tool when applied on realistic tasks, while having as a reference for evaluation the state of the art tools with similar features. Thus, none of the test models have been designed ad-hoc or passed any selection in order to better fit our tool's characteristics. In fact, a set of realistically large, real-world test models, already introduced in the CIT research literature [12, 22, 28] are used here.

The first model, TCAS, is a well known model of the specification of a software module part of a Traffic Collision Avoidance System (TCAS), presented by Kuhn at al. [28] and broadly used in the CIT literature. SPIN is a well-known publicly available model checking tool [25], 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: SPIN simulator and SPIN verifier. 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

DOI: 10.1002/stvr

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.

Apache and Bugzilla are similarly modeled after the well known open source browser and bug-tracking system, respectively. Note that these are all tasks with mixed ranges varying from two to ten symbols, and number of involved parameters ranging from a dozen to almost two hundreds. Moreover, all tasks have also constraints over the inputs, that is, restrictions on which tuples may legally be present in the computed MCA (i.e. because they are actually realizable). In fact, in the following, two tasks are reported, first ignoring the constraints and then taking them into account, with results shown respectively in the data tables III and IV. Note that the exponential notation used by Hartman and Raskin [24] to represent the problem domain size has been also adopted here, that is d^n means a task with n parameters of range d, while the notation used to represent the constraints has been taken from Calvagna and Gargantini[8]. In this latter, similarly, the expression d^n corresponds to a set of n forbidden tuples of width d, that is, involving d parameters.

7.4. Test Results

Table III reports the size of the MCAs produced by the tested tools, at varying strength t, in each of the considered example systems. Reported results for non deterministic tools are rounded averages, with the lowest size found shown in brackets. Please note also that in table III all tasks have been tested ignoring their constraints.

The first tested task, TCAS, has twelve parameters, whose ranges vary from two to ten symbols, thus is the task involving the highest range among all the considered tasks. However, it is also the task with the lowest number of involved parameters. In the TCAS experiment our tool outperformed all the compared tools, at all tested degrees of interaction, producing also minimal sized MCAs for all t but the sixth strength. In the SPIN Simulator task, our tool's performance is not the best, but is fairly comparable to the others', at the lower strengths. On the other hand, it is the best at strengths five and six, with minimal sized MCA produced at strength five. Note that the first two example systems, TCAS and SPIN simulator, were tested up to the sixth degree, Bugzilla up to the fifth, and the last three up to the fourth only. Although some tools (Ttuples and Jenny) can be run at any strengths, their running times for TCAS and SPIN Simulator would have in that case fairly exceeded the one hour computation limit, to which they came already close at strength six. On the other hand, in these two tasks IPO-F happens to be faster, but a hard limit at strength six is encoded in ACTS. Clearly, the one hour time bound prevented higher degrees of interaction from being completed on time in successive tasks, which are also increasingly more complex tasks. In the Bugzilla task fifty two parameters are involved, with ranges from two to four, which makes this task a medium size with low ranges involved. Results show that our tool in this case achieved better performance than the others for strength two to four, and only slightly worse than that of IPO-F at strength five. Our tool was the only achieving optimal (minimal) performance for this task, at strength two. Moreover, IPOG and IPOG-D where not able to complete on time the task for strength five.

In the SPIN Verifier task our tool performance at strength two (31) is comparable to that of the best performing tool (29), while it is significantly worse for strengths three and four, by a percentage amount of 15% and 30% respectively, which is an evidence that the heuristic adopted in Ttuples is not as efficient as the others' compared, when applied to tasks involving a significant number of non boolean parameters. In fact, by looking at the table, it is possible to observe that the tasks where

DOI: 10.1002/stvr

Table III. Test suite sizes for unconstrained models of real-world systems. A dash sign (-) means that a tool did not complete a task by 1 hour. A star ($\star$) means that the tool did not complete but an intermediate result was available. A dagger (†) mark appears on tests which required the Java heap size to be increased from the default size to 500MB, to be able to complete. Note that this was never necessary for Ttuples. IPOG-D did not complete some tests at strength t=4, due to a software failure of the ACTS tool.

task name and size	t	Ttuples	IPOG	IPO-F	IPOG-D	CASA	Jenny
TCAS	2	**100 (100)**	**100**	**100**	130	**100 (100)**	**100**
$10^2 4^1 3^2 2^7$	3	**400 (400)**	**400**	**400**	480	404 (**400**)	418
	4	**1200 (1200)**	1377	1357	NA	1230*	1548
	5	**3605 (3600)**	4292	4260	13458	-	4621
	6	**10249 (10177)**	11939	11241†	41280	-	-
SPIN simulator	2	23 (17)	20	24	28	**17 (16)**	27
$4^5 2^{13}$	3	112 (102)	**78**	96	112	**86 (85)**	108
	4	394 (373)	**341**	339	NA	-	409
	5	**1024 (1024)**	1236	1159	5054	-	1290
	6	**3328**	3516	3527	30214	-	3756
Bugzilla	2	**16 (16)**	18	19	26	17 (16)	20
$4^2 3^1 2^{49}$	3	**52 (51)**	66	69	90	63 (60)	70
	4	**202 (197)**	233	212	669	305*	230
	5	676 (670)	-	**644**	-	-	740
SPIN verifier	2	32 (29)	32	29	40	**27 (26)**	37
$4^{11} 3^2 2^{42}$	3	198 (191)	208	167	248	**162***	172
	4	1027 (992)	761	**785**	NA	-	806
GCC	2	**17 (16)**	21	21	24	18 (17)	26
$3^{10} 2^{189}$	3	**74 (70)**	78	75	98	87*	83
	4	275 (**263**)	-	271	-	-	-
Apache	2	**30 (30)**	36	36	51	34 (31)	42
$6^1 5^1 4^4 3^8 2^{158}$	3	183 (**173**)	175	**173**	255	246*	203
	4	882 (868)	-	**808**†	-	-	-
Total			24927	25812	92257	2796	14706
Total for Ttuples on the same subset			22266	23897	19645	2639	12693
Improvements of Ttuples			11%	7%	79%	6%	14%

Ttuples performance degraded faster at higher strengths are actually those with the highest number of non boolean parameters, with a constant or gradually varying ranges profile, specifically SPIN Verifier and Apache. It is intuitively clear that if the ranges have a steep profile, like it is the case for TCAS, a lot more tuples can be hosted in the MCA without requiring additional rows, making easier the task of building a compact MCA. In the GCC task our tool has shown to be always capable of performance slightly better than the other tools, despite the high number of low-ranged parameters involved, that is 199, which makes this task not a complex task but a long, time consuming task. In fact, besides Ttuples only IPO-F was able to complete it on time up to the fourth degree.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

In the last task, which is also the most complex of the set, our tool performed remarkably better then the other tools at strength two, producing also a minimal sized test suite. At strength three the performance is comparable to the best performance among the others, while at strength three it is only slightly worse (about 10%) than that of IPO-F, which is though the only other tool able to complete the task at that level. However, IPO-F required increasing the memory allocation with respect to the default Java heap that was sufficient for Ttuples.

Overall, Ttuples showed comparable performance to the IPOG algorithm, and its optimized version IPO-F, although with opposite trends in their behaviors. In fact, Ttuples is generally better or comparable at the lower strengths, while it tends to be worse at the highest strengths, which is also more evident as the task size increases. This is evidently the limitation of adopting a blind, statistical approach, that is, the random-based heuristic currently employed, to drive the search for the solution. When the search space becomes extremely larger, as it is for more complex tasks and/or higher strengths, the constant number of tries, that is 100 at most, becomes too small to explore it adequately. Thus, designing smarter heuristics to be used within Ttuples in order to improve its performance is an objective of future research work. However, the tool as it is has already performed also comparably with the other non deterministic tool, CASA, and noteworthy better than IPOG-D, in all the tasks and strengths. This is particularly significant as IPOG-D is just the version of the IPO algorithm designed, like ours, with the objective of limiting the enumeration requirements of the construction process. Moreover Ttuples gave a total improvement in terms of total cost of testing, for the six considered tasks, of about 7% with respect to IPO-F, which is the only tool comparable to it on all the tasks and strengths (see the total in table III). Considering only the tasks completed by other tools, then the cost savings obtained with Ttuples ranges from the 11% with respect to IPOG and the 79% with respect to IPOG-D. In addition, Ttuples was the only tool capable of running all reported tests by the given time lap, and without never running out of memory. It is then possible to conclude that Ttuples is a better tradeoff between performance and resource consumption, when testing at higher strength than pairwise is desirable, in environments with scarse or limited memory resources.

7.5. Tests with Constraints

In this paragraph a few additional experiments are reported, this time performed taking into account the tasks constraints, in order to show that the construction algorithm proposed in this paper successfully implements this kind of feature, too. In this case, a comparison is reported with *Jenny* and with the *CASA* tool [13, 22], which implements the most up to date, optimized version of the mAETG algorithm and, to the best of our knowledge, is the best performing tool freely available for *constrained* MCAs construction. The CASA tool's *meta-heuristic* algorithm has been specifically designed to support constraints as an internal feature, instead of a separate processing stage. This design choice is also shared with Ttuples and with the third compared tool, Jenny. Moreover, these tools all support constraints in the form of forbidden tuples of varying width, only. On the other hand, it is even more interesting to compare these three tools, as they are built upon completely different MCA construction approaches: greedy and by columns (Ttuples), meta-heuristic and by rows (CASA), greedy and by rows (Jenny). In addition, they also have differing approaches in order to support the constraints: mAETG/CASA incorporates a SAT solver algorithm, while Ttuples adopts the approach presented in section 6 (the solution adopted by Jenny is undocumented).

DOI: 10.1002/stvr

Table IV. Test suite sizes when taking into account their constraints.- : timeout of 1 hour reached. ⋆: incomplete results. †: incomplete coverage

name	constraint size	t	Ttuples	CASA	Jenny
TCAS	2^3	2	**100 (100)**	**100 (100)**	105†
		3	**401 (401)**	407 (403)	410†
SPIN simulator	2^{13}	2	26 (23)	**20** (19)	26
		3	125(112)	**101 (95)**	117
Bugzilla	$3^1 2^4$	2	**16 (16)**	**16 (16)**	21
		3	**62 (59)**	63 (60)	73
SPIN verifier	$3^2 2^{47}$	2	45 (41)	**37 (34)**	46†
		3	305 (291)	**242**⋆	251†
GCC	$3^3 2^{37}$	2	24 (21)	**21 (19)**	29†
		3	121 (115)	-	**103**†
Apache	$5^1 4^2 3^1 2^3$	2	**31 (30)**	33 **(30)**	42
		3	**185 (182)**	-	205†

Please note that among the available implementations of the original IPO algorithm featured by the ACTS tool, only IPOG does support constraints over the inputs, although support of parameter relations and constraints is listed as future work [31]. Unfortunately, ACTS was not able to build valid MCAs satisfying the constraints and as a consequence, it has not been included in this comparison. Results of the comparison are reported in Table IV for a the previous set of tasks, for which it also lists the size of their constraints, in exponential notation. The performance of our tool is always comparable to that of the others, and better in a few cases. It is interesting to note that the performance of the CASA tool tends to be always better than the others, but its running times (not reported) were always significantly longer then the other's. In fact, already at strength three, for the GCC task, after one hour it was not jet finished but already produced an intermediate MCA with the reported size (marked with a ⋆) and, for the next (larger) tasks it has missed the one hour boundary without even producing any usable intermediate result. Moreover, in order to comply with the constraints the Jenny tool produced MCAs with incomplete coverage, in tasks marked with a †. In conclusion, if constrained CIT has to be applied, Ttuples shows to be also a better trade-off between performance and viability, among the compared tools.

8. CONCLUSION

In this paper a new parameter-based technique for incremental construction of mixed alphabet covering arrays (MCAs) of arbitrary strength t, with constraints support, has been presented. Although a lot of algorithms for building pairwise test suites have been already proposed, very few currently exist that support constrained t-wise MCAs construction, among which only one parameter-based. Differently from pairwise testing, at higher strengths of interaction the time and space requirements for the

DOI: 10.1002/stvr

construction of the MCA became very relevant, as long as they grow exponentially with the size of the task. Thus, it is of major importance to take them into account in order to design really usable tools in practice, especially when the available resources are constrained. The presented algorithm exploited a symmetry property of covering arrays in order to achieve partial coverage by initializing a new column, so to *inherit* the combinatorial relations between existing parameters. As a consequence, the resulting algorithm is characterized by the lowest space requirements among the state-of-the-art greedy MCA construction approaches, to the best of our knowledge.

Thanks to the coverage inheritance concept, the enumeration requirements are reduced to a set of pairs, constant with respect to the number of parameters in the system. Coverage inheritance and the separation of concerns in the algorithm structure, between the reusable core construction algorithm and the delegate objects, implementing interchangeable heuristic strategies, are also original contributions.

A tool, Ttuples, implementing the proposed algorithm has been developed and the results of its application on unconstrained tasks has been reported and commented, with the intent to give also a picture of its performance with respect to state of the art tools with similar features. Our solution has outperformed IPOG-D, which is the only other greedy solution existing in the literature, besides ours, just designed and proposed as a solution to cope with the problem of reducing the enumeration requirements of constructing MCAs. Moreover, the comparison shows that our tool is less demanding in terms of memory space, while being at the same time slightly better, on average, in terms of MCA size, with respect to the best performing algorithm implemented by the ACTS tool, IPOF, even if this has been designed to optimize just the MCA size, regardless of the space requirements.

An algorithm extension to support constraints over the inputs, in terms of forbidden tuples, has been also presented and implemented, and results demonstrated that it performs comparably with state of the art tools designed for constrained MCA generation, and in particular with the CASA tool, which incorporates a SAT solver in the MCA construction process. It is also important to remark that, to the best of our knowledge, this is the only solution currently presented in the CIT literature showing a solution to integrate support for constraints in a parameter-based construction algorithm, which is then an original contribution of this paper. Although in current experimentation only one heuristic have been applied, there is probably a lot of potential to further improve the performance of this approach by designing smarter heuristics (e.g. how to select rows, which uncovered pair to recover next, and so on), particularly to further improve the performance with respect to very large parameter ranges, and on applying the use of constraint solvers or model checkers to the CIT in the presence of constraints, by combining the techniques proposed by Calvagna and Gargantini [6, 7] with the technique presented in this paper.

9. THANKS

The authors thank Charles J. Colbourn for reading an early version of this paper. Thanks also to Jeff Lei and D.Richard Khun for their kind support on the ACTS tool. The SPIN, GCC, Bugzilla, Apache models, and the CASA tool have been introduced in [13] and [22] by Myra B. Cohen et al., to whom goes also the authors' sincere thanks for having kindly made them available.

REFERENCES

DOI: 10.1002/stvr

1. ATGT Abstract State Machines test generation tool project. http://cs.unibg.it/gargantini/software/atgt/.
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. R. C. Bryce, C.J. Colbourn, and M.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. A. Calvagna and A. 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. A. Calvagna and A. 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. A. Calvagna and A. Gargantini. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In Catherine Dubois, editor, *Tests and Proofs, Third International Conference, Prato, Italy, April 9-11, 2008. Proceedings*, volume 5668 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2009.
9. A. Calvagna and A. Gargantini. IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays. In *5th Workshop on Advances in Model Based Testing (A-MOST 2009), Proceedings*, 2009. IEEE.
10. 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.
11. 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.
12. M. B. Cohen, C.J. Colbourn, and A.C.H. Alan. Augmenting simulated annealing to build interaction test suites. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 394, Washington, DC, USA, 2003. IEEE Computer Society.
13. M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *IEEE Transactions on Software Engineering*, to appear, 2008.
14. Charles J. Colbourn. Best known lower bounds for covering arrays. http://www.public.asu.edu/~ccolbou/src/tabby/catable.html.
15. Charles J. Colbourn, Sosina S. Martirosyan, G.L. Mullen, D.Shasha, G.B. Sherwood, and J. L. Yucas. Products of mixed covering arrays of strength two. *Journal of Combinatorial Designs*, 14(2):124–138, 2006.
16. J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.
17. S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott. Model-based testing of a highly programmable system. In *Software Reliability Engineering, The Ninth International Symposium on*, pages 174–179, Los Alamitos, CA, USA, 1998. IEEE Computer Society.
18. 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.
19. 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.
20. M. Forbes, J. Lawrence, Y. Lei, and D. R. Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. Internal Report 5, NIST Journal of Research, pp. 287-297, 2008.
21. A. Gargantini. Using model checking to generate fault detecting tests. In *International Conference on Tests And Proofs (TAP)*, number 4454 in Lecture Notes in Computer Science (LNCS), pages 189–206. Springer, 2007.
22. B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *International Symposium on Search-Based Software Engineering (SSBSE)*, pages 13–22, May 2009.
23. M. Grindal, J. Offutt, and S.F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
24. A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.
25. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
26. Jenny combinatorial tool. http://www.burtleburtle.net/bob/math/jenny.html.
27. N. Kobayashi, T. Tsuchiya, and T. Kikuno. Non-specification-based approaches to logic testing for software. *Journal of Information and Software Technology*, 44(2):113–121, February 2002.

DOI: 10.1002/stvr

28. D. R. Kuhn and V. 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.
29. 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.
30. D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.
31. Y. Lei, R. Kacker, D. R. Kuhn, V. Okum, and J. Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing Verification and Reliability*, 18(3):125–148, 2008.
32. Y. Lei and D. R. Kuhn. Advanced combinatorial testing suite (ACTS). http://csrc.nist.gov/acts/.
33. Y. Lei and K.-C. 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.
34. 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.
35. K. Nurmela. Upper bounds for covering arrays by tabu. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.
36. Pairwise web site. http://www.pairwise.org/.
37. G. Seroussi and N. H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.
38. G.B. Sherwood. Optimal and near-optimal mixed covering arrays by column expansion. *Discrete Mathematics*, 308(24):6022–6035, 2008.
39. B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In CO Breckenridge, editor, *Fifth International conference on Artificial Intelligence Planning Systems (AIPS)*, 2000.
40. K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.
41. A. W. Williams and R. 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.
42. C. Yilmaz, M.B. Cohen, and A.A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.

DOI: 10.1002/stvr