Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formal logic approach to constrained combinatorial testing

Andrea Calvagna · Angelo Gargantini

Received: date / Accepted: date

Abstract Combinatorial testing is as an effective testing technique to reveal failures in a given system, based on input combinations coverage and combinatorial optimization. Combinatorial testing of *strength* t ($t \geq 2$) requires that each t-*wise* tuple of values of the different system input parameters is covered by at least one test case. Combinatorial test suite generation algorithms aim at producing a test suite covering all the required tuples in a small (possibly minimal) number of test cases, in order to reduce the cost of testing. The most used combinatorial technique is the *pairwise* testing ($t = 2$) which requires coverage of all pairs of input values. Constrained combinatorial testing takes also into account constraints over the system parameters, for instance *forbidden* tuples of inputs, modeling invalid or not realizable input values combinations. In this paper a new approach to combinatorial testing, tightly integrated with formal logic, is presented. In this approach, test predicates are used to formalize combinatorial testing as a logical problem, and an external formal logic tool is applied to solve it. Constraints over the input domain are expressed as logical predicates too, and effectively handled by the same tool. Moreover, inclusion or exclusion of select tuples is supported, allowing the user to customize the test suite layout. The proposed approach is supported by a prototype tool implementation and results of experimental assessment are also presented.

Keywords model-based testing; combinatorial testing; test generation

1 Introduction

Verification and validation of highly-configurable software systems, such as those supporting many input parameters or customizable options, is a challenging activity. In

A. Calvagna
Dip. Ingegneria Informatica e delle Telecomunicazioni
University of Catania - Italy
E-mail: andrea.calvagna@unict.it

A. Gargantini
Dip. Metodi Matematici e Ingegneria dell'Informazione
University of Bergamo - Italy
E-mail: angelo.gargantini@unibg.it

fact, due to its intrinsic complexity, formal specification 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. On the other hand, unintended interaction between input parameters can lead to incorrect behaviors which may not be detected by traditional testing [39,50].

To this aim, combinatorial interaction testing (CIT) techniques [17,28,39] can be effectively applied in practice [5,46,38]. CIT consists of employing combination strategies to select values for inputs and combine them to form test cases. The tests can then be used to check how the interaction among the inputs influences the behavior of the original system under test. The most used combinatorial testing approach is to systematically sample the set of inputs in such a way that all t-way combinations of inputs are included. This approach exhaustively explores t-strength interaction between input parameters, generally in the smallest possible test executions.

For instance, pairwise interaction testing aims at generating a reduced size test suite which covers all *pairs* of input values. Significant time savings can be achieved by implementing this kind of approach, as well as in general with t-wise interaction testing. As an example, exhaustive testing of a system with a hundred boolean configuration options would require 2^{100} test cases, while pairwise coverage for it can be accomplished with only 10 test cases. Similarly, pairwise coverage of a system with twenty ten-valued inputs (10^{20} distinct input assignments possible) requires a test suite sized less than 200 tests cases only. Also, it has been experimentally shown that CIT is really effective in revealing software defects [37]. A test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [47,18]. Other experimental works have shown that usually 100% of faults are already triggered by a relatively low degree of interaction, typically 4-way to 6-way combinations [39], and that the testing of all pairwise interactions in a software system finds a significant percentage of the existing faults [18]. Dunietz *et al.* [20] compare t-wise coverage to random input testing with respect to structural (block) coverage achieved, with results showing higher reliability of the former in achieving block coverage if compared to random test suites of the same size. Burr and Young [7] report 93% code coverage as a result from applying pairwise testing of a commercial software system. For this reason combinatorial testing is used in practice and supported by many tools [44].

Combinatorial testing is applied to a wide variety of problems: highly-configurable software systems, software product lines which define a family of software, hardware systems, and so on. As an example, Table 1 reports the input domain of a simple telephone switch billing system [40], 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 covering all the possible combinations for the BBS inputs shown in Table 1 would require $3^4 = 81$ tests, the pairwise coverage of the BBS can be obtained by the test suite reported in Table 2 which contains only 11 tests.

From a mathematical point of view, the problem of generating a minimal set of test cases covering all pairs of input values is equivalent to finding a *covering array* (CA) of *strength* 2 over a heterogeneous alphabet [3,32]. Covering arrays are combinatorial structures which extend the notion of *orthogonal arrays* [4]. A covering array

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	BLOCKED	PBX
2	CALLER	LONGDISTANCE	BUSY	PBX
3	EIGHT_HUNDRED	INTERNATIONAL	BUSY	LOOP
4	COLLECT	LOCALCALL	BUSY	ISDN
5	COLLECT	LONGDISTANCE	SUCCESS	PBX
6	COLLECT	INTERNATIONAL	BLOCKED	PBX
7	CALLER	INTERNATIONAL	SUCCESS	ISDN
8	CALLER	LOCALCALL	BLOCKED	LOOP
9	EIGHT_HUNDRED	LONGDISTANCE	BLOCKED	ISDN
10	COLLECT	LOCALCALL	SUCCESS	LOOP
11	EIGHT_HUNDRED	LONGDISTANCE	SUCCESS	LOOP

$CA_\lambda(N; t, k, \mathbf{g})$ is an $N \times k$ array with the property that in every $N \times t$ sub-array, each t-tuple occurs at least λ times, where t is the strength of the coverage of interactions, k is the number of components (degree), and $\mathbf{g} = (g_1, g_2, ...g_k)$ is a vector of positive integers defining the number of symbols for each component. 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.

In most cases, constraints or dependencies exist between the system inputs. They normally model assumptions about the environment or about the system components or about the system interface and they are normally described in natural language. If constraints are considered, then the combinatorial testing becomes constrained combinatorial interaction testing (CCIT). However, as explained in sections 2 and 7, most combinatorial testing techniques either ignore the constraints which the environment may impose on the inputs or require the user to modify the original specifications and add extra information to take into account the constraints.

In this paper, we describe a new approach to CCIT in which constraint support is not implemented in a pre/post-processing stage, but it is embedded in the test suite generation process. This approach is based on formal logic, since it uses logical predicates to formalize combinatorial testing as a formal logical problem. Coverage requirements, that is the tuples to be covered, are modeled as predicate expressions, called *test predicates*. This way, generating a test case covering a given tuple reduces to the problem of finding a model of the corresponding predicate expression, which can be solved by a great variety of formal-analysis tools and techniques. Also constraints can be modeled as further testing requirements by logical expressions, so that they can be seamlessly processed together with the test predicates. Moreover, using a formal logic notation to express constraints allows the user to easily specify complex and/or large set of constraints with a compact syntax.

Moreover, since the set of combinatorial requirements grows exponentially with the size of the system, combinatorial test suites generated with naive techniques can be too large to be useful in practice. As a consequence, the effectiveness of the test suite generation process is an issue of major importance. In the proposed approach, test suites are built incrementally, one test case at a time, and several optimization techniques are inserted in the process in order to reduce the number of iterations and build smaller test suites. A *monitoring* technique is applied, consisting in keeping track of the coverage provided by each built test case, and skipping the test generation for test predicates already covered. Different strategies can be selected to determine the *order* in which test predicates are processed. A *collecting* technique has been devised to maximize the number of uncovered test predicates which will be covered by the next test case. Finally, a post-processing *reduction* algorithm tries to further reduce the final test suite size by removing test cases which are redundant.

This paper is the extended version of our paper [8] to which it adds several original new contributions, including the ability to tackle t-wise coverage, a novel way to collect test predicates in the presence of constraints, and more detailed experiments and comparison with other techniques.

The paper is organized as follows: section 2 gives some insights into the topic of combinatorial testing and states the main goals of our work. Section 3 presents our approach and an overview of the tool we implemented, while section 4 explains how we deal with constraints over the inputs. Section 5 explains how user requests can be easily integrated. Section 6 presents results of experiments carried out in order to assess the validity of the proposed approach. Section 7 presents recently published related works. Finally, section 8 draws our conclusions and points out some ideas for future extension of this work.

2 Combinatorial coverage strategies

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 recent survey [28]. There is also a web site [44] devoted to this subject and several automatic tools are commercially [11] or freely available [47]. We can classify them according to Cohen et al. [14], as follows:

a) *algebraic* when the Covering Array (CA) is given by a mathematical construction, as in [36]. These approaches usually lead to optimal results, that is minimally sized CAs. Unfortunately, no mathematical solution to the covering array generation problem exists which is generally applicable. Note that the general problem of finding a minimal set of test cases that satisfies t-wise coverage is NP-complete [49, 45]. Thus, heuristic approaches, producing sub-optimal results are widely used in practice.
b) *greedy* when some search heuristic is used to incrementally build up the CA, as done by AETG [11] or by the In Parameter Order (IPO) [47]. This approach is always applicable but leads to sub-optimal results. Typically, only an upper bound on the size of constructed CA can be guaranteed. The majority of existing solutions fall into this category, including the one we are proposing here.
c) *meta-heuristic* when genetic-algorithms or other less traditional, bio-inspired search techniques are used to converge to a near-optimal solution after an acceptable

number of iterations. Only few examples of this applications are available, to the best of our knowledge [13,43].

Beyond this classifications, it must be observed that most of the currently available methods and tools are strictly focused on providing an algorithmic solution to the mathematical problem of covering array generation only, while very few of them account also for other complementary features, which are rather important in order to make these tools really useful in practice (like i.e. the ability to handle constraints on the input domains). We have identified the following requirements for an effective combinatorial testing tool, extending previous work on this topic by Lott et al. [40]:

A. Ability to deal with user specific requirements on the test suite The user may require the explicit exclusion or inclusion of specific test cases, e.g. those generated by previous executions of the used tool or by any other means, in order to customize the resulting test suite. The tool could also let the user interactively guide the on-going test case selection process, step by step. Moreover the user may require the inclusion or exclusion of *sets of* test cases which refer to a particular critical scenario or combination of inputs. In this case the set is better described symbolically, for example by a predicate expression over the inputs. Note that *instant* [28] strategies, like algebraic constructions of orthogonal arrays and/or covering arrays, and *parameter-based*, iterative strategies, like IPO, do not allow this kind of interaction.

B. Integration with other testing techniques Combinatorial testing is just *one* testing technique. The user may be interested to integrate results from many testing techniques, including those requiring the complete model of the system to derive test cases (as in [26,25,24,23]). This shall not be limited to having a common user-interface for many tools. Instead, it should go in the direction of generating a unique test-suite which simultaneously accounts for multiple kinds of coverages (e.g., combinatorial, state, branch, faults, and so on). Our method, supported by a prototype tool, aims at bridging the gap between the need to formally prove any specific properties of a system, relying on a formal model for its description, and the need to also perform functional testing of its usage configurations, with a more accessible *black-box* approach based on efficient combinatorial test design. Integrating the use of a convenient model checker within a framework for combinatorial interaction testing, our approach gives to the user the easy of having just one convenient and powerful formal approach for both uses.

C. Integration with the entire system development process Combinatorial testing should be only one part of the entire system development process, which should include other verification and validation techniques. Both testing and formal analysis should be used in conjunction, in order to balance the required efforts over time. Specifically, in our approach, formal modeling of a system's input/output domains and of its state space (behavior), is not required all at once but can be done in successive stages, respectively. Initially, formal modeling of the input domain only is sufficient to enable the use of combinatorial testing to explore input interactions, with relatively little effort. If constraints over the inputs have to be taken into account, then thay are added now to the input domain model. Meanwhile, or later in time, the same model can be extended to include the actual system's behavioral description too. Rules to compute the expected outputs can be added to the formal model, enabling its conformance testing, i.e. to automatically check that the implemented system produces the expected outputs, and the ability to apply other testing techniques based on structural and fault-based coverage criteria. This lets the user achieve a high degree of confidence on the system correctness. Only in the end, safety properties of the system can be added too, in order to check them during testing, thus still improving the significance of the test process, or to apply formal verification techniques, as model checking or theorem proving.

D. Constraints support A fourth desired requirement of a combinatorial testing strategy is the ability to deal with complex constraints. This issue has been recently investigated by Cohen et al. [14] and recognized as a highly desirable feature of a testing method. Although the presence of constraints reduces the size of combinatorial test suites, it also makes the test generation more challenging. Note that the general problem of finding a minimal set of test cases that satisfies t-wise coverage is NP-complete [49, 45]. If constraints on the input domain are to be taken into account, even finding a single test or configuration that satisfies the constraints is NP-complete [6], since it can be reduced in the most general case to a satisfiability problem.

There are already a few approaches dealing with the constraints over the inputs. In order to deal with constraints, some methods require to remodel the original specification, very few directly support CCIT. Others simply ignore constraints or propose to post process the test suites, in order to delete combinations of inputs which do not satisfy the constraints. An overview of existing methods is presented in section 7.

In this paper we address CCIT in the presence of constraints as suggested in [14], but we keep also in mind the goals of allowing user specific requirements and a better integration with other existing techniques for verification and validation and for system development.

3 A logic approach to combinatorial testing

In this section, we describe our approach to combinatorial testing, which we can classify as *logic-based*, since it formalizes the combinatorial coverage by means of logical predicates and applies techniques normally used for solving logical problems. The preliminary definitions of *test* and *test suite* in the context of combinatorial testing follow.

Definition 1 Given m input variables, each ranging in its own finite domain, a test is an assignment of values to each of the m variables: $p_1 = v_1, p_2 = v_2, \ldots, p_m = v_m$ or $\langle p_i = v_i \rangle$.

Definition 2 A test suite is a finite set of tests. The size of a test suite is simply the number of tests in it.

To formalize pairwise testing, since it aims at validating each possible pair of input values for a given system under test, we can formally express each pair as a corresponding logical expression, a *test predicate* (or test goal), e.g.:

$$p_1 = v_1 \wedge p_2 = v_2$$

where p_1 and p_2 are two inputs or monitored variables of enumeration or boolean domain and v_1 and v_2 are two possible values of p_1 and p_2 respectively. Similarly, t-wise coverage can be modeled by a set of test predicates, each of the type:

$$p_1 = v_1 \wedge p_2 = v_2 \wedge \ldots \wedge p_t = v_t \equiv \wedge_{i=1}^{t} p_i = v_i$$

where $p_1, p_2 \ldots p_t$ are t inputs and $v_1, v_2 \ldots v_t$ are their possible values. The t-wise coverage is represented by the set of test predicates that contains every possible combination of the t input variables with their values. Please note that to reach complete t-wise coverage this has to be true for each t-tuple of input parameters of the considered system.

To build the complete set of test predicates required for t-wise coverage of a model, we employ a combinatorial enumeration algorithm, which simply takes every possible combination of t input variables and it assigns every possible value to them. In order to generate the test predicates, we assume the availability of a formal description of the system under test. This description should include at least the input parameters together with their domains[1].

In order to achieve the goals B and C stated in section 2, we choose as description language the Abstract State Machine notation in its variant of the AsmetaL language [27][2]. As an example, Listing 1 reports the AsmetaL specification of a Cruise Control System (CC) as proposed by [1]. Note that although only *monitored* variables are considered in combinatorial testing, AsmetaL specifications could contain also rules, modeling system behavior, and controlled variables, modeling the outputs. Rules and controlled variables are ignored for combinatorial test generation but can be useful if the specification is used as oracle, to compute the expected outputs in correspondence of a test. The same specification may be reused for other validation & verification activities.

The CC example has 4 boolean and one 3-valued monitored variables, thus the collection of test predicate for its pairwise coverage counts 48 predicates. These are the combinatorial explosion of all assignments for each of the five possible pairs of distinct parameters of CC. Table 3 shows an example of combinatorial test suite achieving such pairwise coverage in just six test cases. The four-wise coverage set for CC counts 112 test predicates. They can be obtained by enumerating all the possible assignments for all the combinations of four out of five parameters.

Table 3 Pairwise test suite for Cruise Control.

#	engRun	brake	fast	igOn	lever
1	true	false	true	false	RESUME
2	false	true	false	true	RESUME
3	false	false	true	true	ACTIVATE
4	true	true	false	false	ACTIVATE
5	true	true	true	true	DEACTIVATE
6	false	false	false	false	DEACTIVATE

The activity of generating the test predicates (step 1) is carried out by the *test predicate generator* of Figure 1, which shows the process proposed by our method

[1] Currently, only finite, discrete enumerable domains are supported.

[2] AsmetaL is available at http://asmeta.sourceforge.net.

```
asm cruiseControl
import StandardLibrary
//UNIVERSES and FUNCTIONS
signature:
  enum domain CCMode = {OFF| INACTIVE|CRUISE|OVERRIDE}
  enum domain CCLever = {DEACTIVATE| ACTIVATE|RESUME}
  dynamic monitored lever : CCLever
  dynamic monitored igOn : Boolean
  dynamic monitored engRun : Boolean
  dynamic monitored brake : Boolean
  dynamic monitored fast : Boolean
  dynamic controlled mode : CCMode
definitions:
// AXIOMS: ADDED LATER
// RULES:
main rule r_CruiseControl = ...
// SKIPPED
```

Listing 1 AsmetaL specification of Cruise Control

and implemented by the *ASM Test Generation Tool* (ATGT)[3]. ATGT was originally developed to support structural [26] and fault based testing [23] of *Abstract State Machines* (ASMs), and it has been extended to support also combinatorial testing.

We say that a test *ts* covers a test predicate *tp* if and only if it is a model of *tp*. Note that while a test binds *every* variable to one of its possible values, a test predicate binds only t (with $t \leq m$) variables. We say that a test suite achieves the t-wise combinatorial coverage if all the test predicates for the t-wise coverage are covered by at least one test in the suite. The main goal of combinatorial testing is to find a small test suite able to achieve the t-wise coverage.

To generate the complete test suite, one could choose one test predicate at a time and try to generate a test that covers it. By formalizing the t-wise testing by means of logical predicates, finding a test that satisfies a given predicate reduces to a logical problem of finding a complete[4] model for a logical formula. To this aim, many techniques like constraint solvers and model checkers can be applied. The complete process of generating the final test suite depicted in Figure 1 contains several optimizations and it is explained in the following sections.

3.1 Tests generation

The actual test generation (stage 4 in Figure 1) consists of finding a test that covers a given test predicate, i.e. that is a model for it. As long as constraints are not taken into account, since a test predicate is just a conjunction of atoms of the form $v = x$, finding a model is trivial and even a simple ad-hoc algorithm could be used. However, in order to support contraints many logical solvers tools that are already available can better suit this task, like e.g. constraint solvers, SAT algorithms, SMT (Satisfiability Modulo Theories) solvers, and even model checkers. Our approach exploits the well known SAL (Symbolic Analysis Laboratory) [19] model checker tool. The SAL framework aims at

[3] ATGT is available at http://cs.unibg.it/gargantini/software/atgt/.

[4] We say that a model is *complete* if it assigns a value to every input variable.

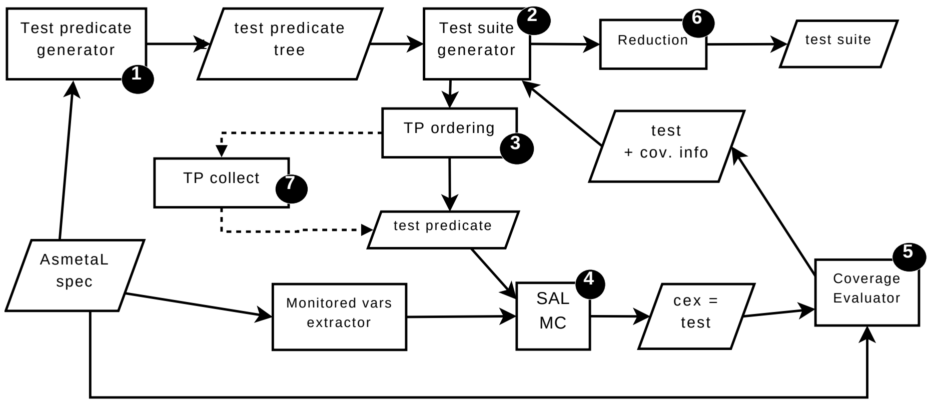

Figure 1 Generation process of a combinatorial test suite

combining different tools for abstraction, program analysis, theorem proving, SMT solving, and model checking. Even though it is mainly used for formal verification, it has been successfully used also for test generation already [29].

SAL offers a bounded (BMC) and a symbolic model checker (SMC). *Bounded model checkers* [2] are specialized to the generation of counterexamples. A BMC transforms the model checking problem into a constraint satisfaction problem: for finite state systems, this can be represented as a propositional satisfiability problem which contains the unfolded transition relation and the negation of the property up to a certain bound (depth) and this problem is given to a SMT or SAT solver. Modern SMT solvers can handle problems with many thousands of variables and constraints. A *symbolic model checker* [42] uses binary decision diagrams (BDDs) to efficiently represent states, transition relations and constraints among them and then it can apply a variety of search strategies to explore the state space and to prove a system property or to find a counter example of it.

In order to generate a test which covers a given combinatorial test predicate tp, SAL is asked to verify a *trap property* [24] in a model which contains only the monitored variables. The trap property states that tp is never true, or $never(tp)$, which in LTL, the language of SAL, becomes $G(\neg tp)$. The trap property is not a real system property, but enforces the generation of a counter example, that is an assignment of values falsifying the trap property and satisfying our test predicate. The counter example will contain bindings for all the inputs, including those missing (free) in the predicate, thus defining the test we were looking for. The SAL translation of CC for a generic test predicate $<$ `tp` $>$ is shown in Listing 2, in which the `monitored` module represents the monitored variables.

Since we do not consider any state transition information in our models, a simple constraint or satisfiability solver could be used instead of a model checker. However, not all the constraint and SMT solvers are able to find a *complete* model: some simply say that a model exists without showing it, others print the model but it is not complete (i.e. it does not bind all the variables). Even a simple SAT solver could be used. However, there are several ways to encode a constrained problem directly into a SAT solver, but the choice of which encoding to use is critical and it requires expertise in SAT algorithms, since some encodings may provide better performance than others. Moreover, we designed our approach in order to be able to deal also with temporal

```
cruiseControl: CONTEXT = BEGIN

 CCLever : TYPE = {DEACTIVATE, ACTIVATE, RESUME};

  monitored : MODULE = BEGIN

        OUTPUT igOn, fast, engRun, brake: BOOLEAN, lever: CCLever

  END;

 % trap property
 tc_92668c : THEOREM monitored |- G(NOT <tp>);

END
```

Listing 2 SAL specification of Cruise Control

constraints and to be able to include state transition information, which cannot be directly represented in satisfiability solvers. Work on this topic is currently ongoing, and some preliminary results are presented in [9]. Note that the use of BMC with depth $= 1$ is equivalent in most aspects (as number of variables, for example) to apply the underlying SMT or SAT solver directly.

A first basic way to generate a suitable test suite for the t-wise coverage, consists in collecting all the test predicates in a list of *candidates*, extracting from the set one test predicate at a time, generating the test case for it by executing a SAL model checker, removing it from the candidates set, and repeating until the candidates set is empty. This activity would be performed by the *test suite generator* (stage 2 in Figure 1) for *each* test predicate. This approach can be classified as iterative according to [28] and allows the user to pause, postpone, and resume the test generation. However, it is not very efficient since it would require a model checker run and would produce a test for every test predicate; it can be improved as follows.

3.2 Monitoring

Every time a new test *ts* is added to the test suite, *ts* always covers as many as $\binom{m}{t}$ *t-wise* test predicates, where m is the number of a system's input parameters and t is the *strength* of the covering array ($t = 2$ for pairwise interaction testing). Checking which test predicates are covered by *ts* and remove them from the candidates leads to fewer calls to the model checker and possibly to smaller test suites [21]. To enable monitoring, the tool detects if any additional test predicate *tp* in the candidates is covered by *ts* by checking whether *ts* is a model of *tp* (i.e. it satisfies *tp*) or not, and in the positive case it removes *tp* from the candidates. Checking if a test is a model for a test predicate requires very limited computational effort. This activity is performed by the *Coverage Evaluator* (stage 5 in Figure 1), which also computes the expected outputs as values for controlled variables, if any.

3.3 Reduction

Monitoring can significantly reduce the size of a test suite, but a resulting test suite could still contain redundant tests. For example, the last generated test might also cover several other test predicates previously covered by tests which may become useless. A smallest test suite is that in which each test predicate is covered by *exactly* one test case, but this very seldom happens: in most cases a test predicate will be covered by many tests creating possible redundancies. For this reason the analysis of the final test suite is useful to further reduce it.

Test suite reduction (also known as test suite minimization) is often applied in the context of regression testing, when one wants to find a subset of the tests that still satisfies given test goals. The problem of finding the minimal test suite that satisfies a set of test goals can be reduced to the minimum set covering problem which is NP-hard. A simple greedy heuristic for the minimum set covering problem defined in [10] can be adapted to the test suite minimization.

We say that a test case is *required* if it covers at least a test predicate not already covered by other test cases in the test suite. In order to obtain a final test suite with fewer test cases, we try to build the reduced test suite by gathering only all the test cases which are required. Note, however, that a required test case may become no longer required after adding a test case to the test suite, hence we cannot simply add all the required tests at once. We have implemented a greedy algorithm, reported in Alg. 1, which finds a test suite with the minimum number of required test cases.

Algorithm 1 Test suite reduction

```
T = test suite to be optimized
Op = optimized test suite (required test predicates)
Tp = set of test predicates which are not covered by tests in Op

0. set Op to the empty set and add to Tp all the test predicates
1. take the test t in T which covers most test predicates in Tp
2. add t to Op
3. remove all the test predicates covered by t from Tp
4. if Tp is empty then return else goto 1
```

3.4 Test Predicates Ordering

If monitoring is applied, the order in which the candidate test predicates are chosen and processed has a major impact on the size of the final test suite. In fact, each time a *tp* is selected, a corresponding test case is generated, covering also other test predicates, which will be then removed from the candidate pool too. In fact, the more the candidate pool is reduced, the less the variety of test cases will be. Considering test predicates in the same order in which they are generated may lead to not optimal test suites [22]. For this reason, in our process we inserted an additional processing stage (stage 3 of Figure 1) in which the test predicates are ordered according to a user specified policy, chosen among the following:

Randomly The first policy is to randomly choose the next predicate for which the tool generates a test case. This makes our method *non deterministic*, as the generated test suite may differ in size and composition at each execution of the algorithm. Random ordering is considered a good ordering policy [22].

Ordering by novelty A different policy is to order the *tps* in the candidate pool according to a well defined ordering criterion, and then process them sequentially. At each iteration, the pool is again sorted against this criterion and the first test predicate is selected for processing. In order to do this we define a *novelty* comparison criteria as follows.

Definition 3 Let t_1 and t_2 bet two test predicates, and T a test suite. We say that t_1 is more novel than t_2 if the variable assignment of t_1 has been tested less times in T than that of t_2.

Ordering by novelty and taking the most novel one helps ensuring that during the test suite construction process, for each parameter, all of its values will be evenly used, which is also a general requirement of CAs. To this purpose, usage counting of all values of all parameters in current test suite is performed and continuously updated by the algorithm, when this optional strategy is enabled and the ordering is performed every time a test predicate must be chosen.

Ordering requires additional (limited) computational effort to continuously order the test predicates.

Anti-diagonal Pairwise test predicates can be ordered once and for all at the beginning by the *anti-diagonal* criterion, which orders the test predicates such that no two consecutive $tp \equiv p_1 = v_1 \wedge p_2 = v_2$ and $tp' \equiv p'_1 = v'_1 \wedge p'_2 = v'_2$ where $p_1 = p'_1$ and $p_2 = p'_2$ will have $v_1 = v'_1$ or $v_2 = v'_2$. Simply put, for each pair of input variables, the algorithm indexes through the matrix of their possible values in *anti-diagonal* order, see Figure 2. Thus, generating their sequence of pair assignments such that both values always differ from previous ones[5]. We expect that enforcing diversity between two consecutive tps will result in test cases covering a greater number of test predicates.

3.5 Composing test predicates

Since a test predicate binds only the values of a pair of variables, all the other variables in the input set are still free to be bound by the model checker. Beside guiding the choice of the selected test predicate in some effective way, we can only hope that the model checker will choose the values of unconstrained variables in order to avoid unnecessary repetitions, such that the total number of test cases will be low. It is apparent that a guide in the choice of the values for all the variables not specified by the chosen test predicate is necessary to improve the effectiveness of test case construction, even if this may require a greater computational effort. To this aim, our proposed strategy consists in *composing* more test predicates into a *collected* test predicate, which specifies the values for as many variables as possible.

Definition 4 (Collection) A *collected* test predicate is the conjunction of one or more combinatorial test predicates.

[5] With the only exception of the first and last pairs of the sequence.

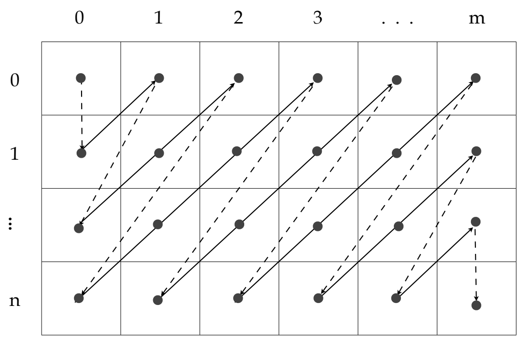

Figure 2 A schema showing *anti-diagonal* indexing order of the pair values. The row and column indexes correspond to the values of two input parameters, each ranging in [0..n] and [0..m] respectively. The arrows show the order in which values are selected to form the test predicates.

When creating a composed test predicate, we must ensure that we will still be able to find a test case that covers it. In case we try to compose too many test predicates which contradict each other, there is no test case for it. We borrow some definitions from propositional logic: since a sentence is *consistent* if it has a model, we can define consistency among test predicates as follows.

Definition 5 (Consistency) A test predicate tp_1 is consistent with a test predicate tp_2 if there exists a test case which satisfies both tp_1 and tp_2.

Let us assume now, for simplicity, that there are no constraints over the input values. In this case, the consistency propery depends only on the values of the variables in the test predicates we compose. The case where constraints over the model are defined will be considered in section 4.

Claim 1 *Let* $tp_1 \equiv v_1 = a_1 \wedge v_2 = a_2$ *and* $tp_2 \equiv v_3 = a_3 \wedge v_4 = a_4$ *be two pairwise test predicates. They are consistent if and only if* $\forall i \in \{1,2\}, \forall j \in \{3,4\}\; v_i = v_j \rightarrow a_i = a_j$

Claim 1 can be easily extended for t-wise test predicates:

Claim 2 *Let* $tp_1 \equiv \wedge_{i=1}^{t} v_i = a_i$ *and* $tp_2 \equiv \wedge_{j=1}^{t} w_j = b_j$ *be two t-wise test predicates. They are consistent if and only if* $\forall i \in [1,t] \forall j \in [1,t]\; v_i = w_j \rightarrow a_i = b_j$

We can add a test predicate *tp* to a composed test predicate TP, only if *tp* is consistent with TP. This keeps the composed test predicate consistent.

Claim 3 *A collection TP of test predicates is consistent with a test predicate tp if and only if every t in TP is consistent with tp*

Now the test suite is built up by iteratively adding new test cases until no more tps are left uncovered, but each test predicate is *composed* from scratch as a logical conjunction of as many as possible uncovered tps. The collection activity uses claims 1, 2, and 3 to keep the composed test predicate *TP* still consistent. Than, *TP* is used to derive a new test case by means of a SAL counterexample. This test will cover all the test predicates composing *TP*. Figure 3 shows the heuristic stage *TP collect* (step 7) of extracting from the pool of *candidate* tps the best sub-set of consistent tps to be joined together into *TP*.

3.6 Composing and ordering

The ordering of the predicates in the candidate pool may influence the later process of merging many test predicates into a composed one. Assume that the candidates tps for merging are searched sequentially in the candidates pool: the more diversity there will be among subsequent elements of the pool, the higher will be the probability that a neighboring predicate will be found compatible for merging. This will in turn impact on the ability to produce a smaller test suite, faster, given that the more predicates have been successfully compacted into the same test case, the fewer test cases will be needed to have a complete combinatorial coverage.

There are more than one strategy we tested in order to produce an effective ordering of the predicates. In the implemented tool one can choose to compose the test predicates (step 7) by any of the *pluggable* ordering policies presented in section 3.4. By adopting the *random* policy, the *TP collect* component randomly chooses the next test predicate and check if it is consistent. By *novelty* policy the tool chooses the most novel test predicate and tries to combine it with the others already chosen.

4 Adding Constraints

Support for constraints over the inputs is given by expressing them as axioms in the specification. In the CC example, the assumptions that the engine is running only if the ignition is on and that the car is driving too fast only if the engine is running, are modeled in AsmetaL by the following axioms:

```
axiom inv_ignition over engRun : (engRun implies igOn)
axiom inv_toofast over fast : (fast implies engRun)
```

To express constraints we adopt the language of propositional logic with equality[6]. Note that most methods and tools admit only few templates for constraints: the translation of those templates into equality logic is straightforward. For example the **require** constraint is translated by an *implication*; the **not supported** to a *not*, and so on. Even the method proposed in [14], which adopts a similar approach to ours, prefers to allow constraints only in a form of forbidden configurations [31], since it relies for the actual tests generation on existing algorithms like IPO. A forbidden combination would be translated in our model as *not* statement. For instance, a forbidden pair $x = a, y = b$ would be represented by the following axiom:

```
axiom inv_fb: not (x = a and y = b)
```

[6] To be more precise, we use propositional calculus, boolean and enumerative types for variables, equality and inequality. This language is expressive enough to represent all the constraints we found in examples and case studies presented in other papers. Nevertheless, our approach can easily support a more general language for constraints as in [9].

Our approach allows the designer to state the constraints in the form he/she prefers. For example, the model of mobile phones presented in [14] has 7 constraints. The constraint number 5 states that *"Video camera requires a camera and a color display"*. In [14], this constraint must be translated into two forbidden tuples, while we allow the user simply to write the following axiom, which is very similar to the informal requirement.

```
axiom inv_5 over videoCamera, camera, display :
videoCamera implies (camera!= NO_CAMERA and display != BLACK_WHITE)
```

Moreover, we support constraints that not only relate two variable values (to exclude a pair), but that can contain generic bindings among variables. We believe that the translation from natural language to a fixed form of constraints (as in [12]) can be error prone, while allowing a more expressive constraint language reduces the likelihood of errors. Note that any constraint models an explicit binding, but their combination may give rise to complex implicit constraints [14]: implicit constraints do not need to be formalized in the specification.

In our approach, the axioms must be satisfied by any test case we obtain from the specification, i.e. a test case is *valid* only if it does not contradict any axiom in the specification. While others [6] distinguish between forbidden combinations (*hard constraints*) and combinations to be avoided if possible (*soft constraints*), we consider only hard constraints. Since we allow the specification to contain also controlled variables and rules that assign value to them, error conditions can be modeled by an *error* controlled variable, and rules that detect erroneous conditions and assign suitable values to *error* in order to signal the occurrence of such conditions. The specification can be used then as oracle to know whether a combination causes an error in the system.

In the presence of constraints, finding a valid test case becomes a challenge similar to finding a counter example for a theorem or proving it. Verification techniques like SAT algorithms, or model checkers algorithms are particularly effective in this case, so we investigated the use of the bounded and symbolic model checkers in SAL to this aim. To include constraints in SAL they must be translated in order to embed the axioms directly in the trap property, since SAL does not support assumptions directly. Simply put, the trap property must be modified to take into account the axioms $a_1, a_2, ...a_n$. The general schema for it becomes:

$$G(a_1 \wedge a_2 \wedge .. \wedge a_n) \Rightarrow G(\neg tp) \tag{1}$$

A counter example of the trap property (1) is still a valid test case. In fact, if the model checker finds an assignment to the variables that makes the trap property false, it finds a case in which both the axioms are true and the implied part of the trap property is false. This test case covers the test predicate and satisfies the constraints.

Without constraints, we were sure that a trap property derived from a test predicate had always a counter example. Now, due to the constraints, the trap property (1) may not have a counter example, i.e. it could be true and hence provable by the model checker. We can distinguish two cases. The simplest case is when the axioms are inconsistent, i.e. there is no assignment that can satisfy all the constraints. In this case each trap property is trivially true since the first part of the implication (1) is always false. The inconsistency may be not easily discovered by hand, since the axioms give rise to some implicit constraints, whose consequences are not immediately detected by human inspection. For example a constraint may require $a \neq x$, another $b \neq y$ while another requires $a \neq x \rightarrow b = y$; these constraints are inconsistent since there is no test case that can satisfy them. Note that also input domains must be taken into account when checking axioms consistency. Inconsistent axioms must be considered as a fault in the specification and this case must be (possibly automatically) detected and eliminated. For this reason when we start the generation of tests, if the specification has axioms, we check that the axioms are consistent by trying to prove with the model checker:

$$G(\neg(a_1 \wedge a_2 \wedge .. \wedge a_n)) \tag{2}$$

If this is proved by the model checker, then we warn the user, who can ignore this warning and proceed to generate tests, but no test will be generated, since no valid test case can be found. We assume now that the axioms are consistent. Even with consistent axioms, some (but not all) trap properties can be true: there is no test case that can satisfy the test predicate and the constraints. In this case we define the test predicate as *infeasible*.

Definition 6 Let tp a test predicate, M the specification, and C the conjunction of all the axioms. If the axioms are consistent and the trap property for tp is true, i.e. $M \wedge C \models \neg tp$, then we say that tp is infeasible. If tp is the t-wise test predicate $p_1 = v_1 \wedge p_2 = v_2 \ldots p_t = v_t$, we say that this combination of assignments is infeasible.

An infeasible combination of assignments represents a set of invalid test cases: all the test cases which contain such a combination are invalid. Our method is able to detect an infeasible assignment, since it can actually prove the trap property derived from it. The tool finds and marks infeasible combinations, and the user may derive from them invalid tests to test the fault tolerance of the system. For example, the following test predicate is proved infeasible for the CC example, since the car cannot run fast when the ignition is off:

$$M \wedge C \models \neg(\mathsf{fast} = \mathsf{true} \wedge \mathsf{igOn} = \mathsf{false}) \tag{3}$$

Note that this infeasible combination is not explicitly listed in the contraints. Infeasible combinations represent implicit constraints. We believe that exposing them to the user can help the detection of possibile errors in the model: consider the example of a desired combination which results infeasible instead.

Since the BMC (Bounded Model Checker) is in general not able to prove a theorem, but only to find counter examples, it would not be suitable to prove infeasibility of test predicates. However, since we know that if the counter example exists then it has length (i.e. the number of system states) equal to 1, it follows that if the BMC does not find it then we can infer that the test predicate is infeasible.

4.1 Composition and constraints

In the presence of constraints, claims 1, 2, and 3 are no longer valid and the composition method presented in section 3.5 must be modified. Every time we want to add a test predicate to a conjoint of test predicates we have to check its consistency by considering the constraints too. We can exploit again the model checker SAL. Given a test predicate tp, the axioms a_1, a_2, ... a_n and the conjoint TP, we can try to prove by using SAL:

$$G(TP \wedge a_1 \wedge a_2 \wedge .. \wedge a_n) \Rightarrow G(\neg tp) \tag{4}$$

If this is proved, we skip *tp* since it is inconsistent with *TP*, otherwise we can add *tp* to *TP* and proceed. This approach would require to call the SAL model checker at least as many times as the number of test predicates, reducing the advantages of composing test predicates. Moreover, the presence of infeasible test predicates worsens the approach: every time we can try to add an infeasible test predicate *tp* to a composed one, the formula 4 is true and *tp* is postponed. To reduce these problems which would limit the scalability of our approach, we have defined the following either sufficient or necessary conditions for consistency.

Claim 4 ***Inconsistent by values*** *Let* $tp_1 \equiv \wedge_{i=1}^{t} v_i = a_i$ *and* $tp_2 \equiv \wedge_{j=1}^{t} w_j = b_j$ *be two t-wise test predicates. They are consistent (regardless of possible axioms) only if* $\forall i \in [1,t] \forall j \in [1,t] v_i = w_j \rightarrow a_i = b_j$

Claim 4 denotes a necessary condition for consistency: if a test predicate in the composed TP and the test predicate to be composed *tp* have $v_i = w_j$ and $a_i \neq b_j$, then TP and *tp* are inconsistent and there is no need to call the model checker.

Claim 5 ***Implied by values*** *Let TP collect the test predicates* $tp_i \equiv \wedge_{j=1}^{t} v_{i,j} = a_{i,j}$ *with* $i = 1, \ldots, m$ *and* $tp \equiv \wedge_{k=1}^{t} w_k = b_k$ *be a t-wise test predicate. If* $\forall k \in [1,t] \exists i \in [1,m] \exists j \in [1,t] v_{i,j} = w_k \wedge a_{i,j} = b_k$*, then TP and* tp *are consistent (regardless possible axioms).*

Claim 5 denotes a sufficient condition for consistency: if a test predicate in the composed *TP* contains already test predicates with the same bindings of variables to values of *tp*, then *TP* and *tp* are consistent and there is no need to call the model checker.

Figure 3 shows the entire process of adding a test predicate *tp* to a composed test predicate *TP*. First we apply claims 4 and 5, then only if they do not give a definitive answer, we call the model checker with the following trap property.

$$G(a_1 \wedge a_2 \wedge .. \wedge a_n) \Rightarrow G(\neg(TP \wedge tp)) \qquad (5)$$

If a model for it is found, then *tp* is added to *TP*, otherwise *tp* is checked for feasibility: if the trap property (1) is proved, then *tp* is marked infeasible.

5 User defined test goals and tests

Our framework is suitable to deal with user defined test goals. In fact, the user may be interested to test some particular critical situations or input combinations and these combinations are not simple *t*-wise assignments. We assume that the user defined test goals are given as generic logical predicates, allowing the same syntax as for the constraints. For example, if the user wants to test the BBS system with a specific combination of inputs, such that *access* is LOOP, *billing* is not CALLER and *calltype* is not LOCALCALL, he is allowed to write the corresponding test goal as follows:

```
testgoal loop:
        access = LOOP and billing != CALLER and calltype != LOCALCALL;
```

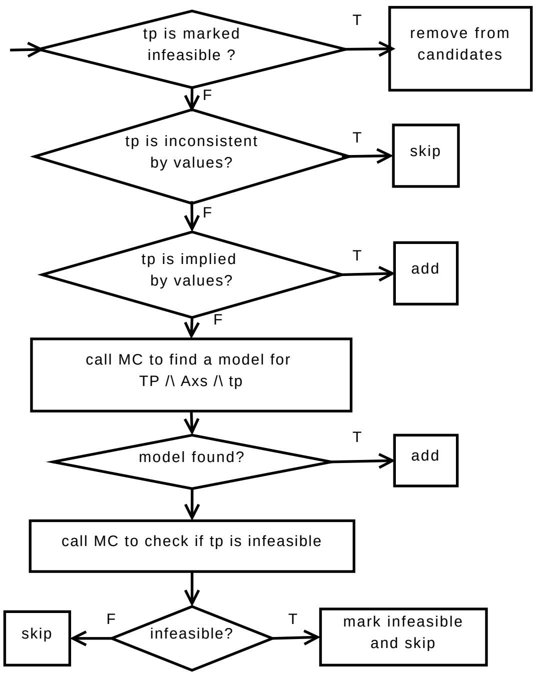
tp is marked infeasible ?
T
remove from candidates
F
tp is inconsistent by values?
T
skip
F
tp is implied by values?
T
add
F
call MC to find a model for TP /\ Axs /\ tp
model found?
T
add
call MC to check if tp is infeasible
skip
F
infeasible?
T
mark infeasible and skip

Figure 3 The collect algorithm in the presence of constraints

Then, a new instance of SAL is run on the trap property derived from the loop test goal, and a test case covering this test goal is finally obtained from the resulting counter example.

Besides user defined test goals, we also allow user defined test cases (sometimes called *seeds*) to be specified. The user may have already some tests cases that mandatorily have to be included in the test suite, and which have already been generated (by any other means). For example, the user may add the following test:

```
test basic_call:
              access = LOOP, billing = CALLER,
              calltype = LOCALCALL, status = SUCCESS;
```

Note that a test case specifies the exact value of all the input variables, while a test predicate specifies a generic scenario. ATGT allows the tester to load an external file containing user defined tests and test goals. When an external file is loaded, ATGT adds the user defined test goals in the set of test predicates to be covered. Then it adds the user defined tests and it checks which test predicates are satisfied by these tests. In this way the tester can decide to skip the test predicates covered by tests he/she has written ad hoc.

6 Evaluation

The proposed approach has been implemented in the tool ATGT, and applied to a series of example tasks taken from the literature [16,32]. These tasks have been used to

Table 4 Test suite sizes comparison using several ordering strategies for unconstrained models.

		no collect			collect				Time
Task	Size	no mon.	random	novelty	random	novelty	antidiag.	as gen.	(secs.)
CCA1u	3^3	9(27)	9(12)	9(11)	9	10(11)	9(15)	9(15)	1.8
CCA2u	4^3	18(48)	18(23)	16(21)	15	17(21)	20(28)	19(28)	4.7
CCA3u	5^3	29(75)	29(36)	29(34)	25	29(31)	32(45)	32(45)	7.6
CCA4u	6^3	43(108)	43(52)	43(53)	36 (38)	42(46)	50(66)	48(66)	11.2
CCA5u	7^3	60(147)	62(73)	59(71)	51	59(63)	66(91)	69(91)	14.2
CCA6u	5^4	500	205	211	152	210	-	225	29.6
CCA7u	6^4	864	364	373	268	372	-	396	53.4
CCA8u	7^4	1372	586	614	428	602	-	637	90.9

benchmark the performance of ATGT in terms of the size of the generated test suite, and assess the different test generation strategies discussed in section 3 of the paper. Experiments have been executed on machine equipped with two 3GHz Quad Core Intel Xeon processors and 16GB of RAM. In tables 4, 7, and 9 we report also the average times taken by our tool to complete the test suite generation by using the strategy which produced the best suite (generally such strategy is also the most demanding in terms of run time). We do not report the memory required which was never greater than 16 Mb.

Note that the exponential symbolic notation used in [32] to represent the problem domain size is also adopted in the following. All the reported suite size values are the best results over fifty tries. Size of the test suite prior to the application of the reduction algorithm (see section 3.3) is also shown bracketed, where greater than the reduced size.

The first series of experiments has been performed on a set of tasks defined in [14], CCA1u to CCA8u, the first five of which required pairwise coverage while the last three three-wise coverage. These are randomly generated constrained covering arrays with 3 to 6 factors of 3 to 7 values each. Based on these specifications, Table 4 reports a comparison of the sizes of generated combinatorial test suites for different strategies, in order to reveal the best performing options. The considered strategies differ for the settings of many configuration options, that are:

– the order of pairs processing: in the same order as generated, random, anti-diagonal, or by novelty;
– the choice between collecting the pairs to compose larger test predicates or processing them one by one;
– the option of monitoring, that is to check (or not) for additional pairs covered by a counterexample other than those for which it was generated. This is actually always enabled with only the exception of the tests in the *no mon.* column.

Since the bounded model checker and the symbolic model checker performances were equivalent, being none of them able to outperform the other in all tasks, the data shown in the tables are the best results obtained from both the smc and bmc versions of the sal tool.

As the experiments outcome shows, while collecting the pairs and enabled monitoring had a dramatic impact on the tool performance, the novelty or anti-diagonal processing order is not more effective than the random order of pair processing, with

Table 5 Comparison of best test suite sizes for unconstrained models

Task	Size	ATGT	mAETG-SAT	SA-SAT	PICT	TestCover
CCA1u	3^3	9	9	9	10	9
CCA2u	4^3	15	16	16	17	16
CCA3u	5^3	25	26	25	26	25
CCA4u	6^3	36	37	36	39	36
CCA5u	7^3	51	52	49	55	49
CCA6u	5^4	152	143	127	151	-
CCA7u	6^4	268	247	222	260	-
CCA8u	7^4	428	395	351	413	-

Table 6 Pairwise performance comparison with existing tools for unconstrained models.

Task size	ATGT	AETG [11]	PairTest [47]	TConfig [48]	CTS [30]	Jenny [35]	AllPairs [41]	PICT [16]
3^4	11	9	9	9	9	11	9	9
3^{13}	19	15	17	15	15	18	17	18
$4^{15}3^{17}2^{29}$	38	41	34	40	39	38	34	37
$4^1 3^{39} 2^{35}$	27	28	26	30	29	28	26	27
2^{100}	12	10	15	14	10	16	14	15
4^{10}	31		31	28	28	30		
4^{20}	39		34	28	28	37		
4^{30}	45		41	40	40	41		
4^{40}	49		42	40	40	43		
4^{50}	51		47	40	40	46		
4^{60}	53		47	40	40	49		
4^{70}	56		49	40	40	50		
4^{80}	58		49	40	40	52		
4^{90}	59		52	43	43	53		
4^{100}	60		52	43	43	53		
10^{20}	267	180	212	231	210	193	197	210

the exclusion of rare exceptions. Without monitoring, the number of runs and the number of the tests in the test suite before optimization are much greater than the number of runs with monitoring, and the number of the tests in the test suites is not smaller. Using the random order processing and enabling collection and monitoring options, revealed to be the best performing strategy for ATGT. For the same set of unconstrained tasks, Table 5 reports a comparison of the best sizes of our tool with those of other well-known existing tools, showing ATGT performance is substantially aligned with that of other tools for the pairwise tasks and very close for the three-wise tasks.

Table 6 report an additional performance comparison between our tool and other well-known existing tools on a different and larger series of tasks of increasing sizes. These tasks are all unconstrained tasks, and have been executed for pairwise coverage configuring ATGT with collection and monitoring enabled and random pairs processing order. Despite the performance achieved is not optimal it still is 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, despite our method is introduced to specifically target constrained models.

Table 7 Test suite sizes for different ordering strategies for constrained models.

		no collect			collect				Time
Task	δ	no mon.	random	novelty	random	novelty	antidiag.	as gen.	(secs.)
CCA1	$2^5 3^1$	9(21)	9	9	9	9	9(10)	9(10)	9.7
CCA2	$2^3 3^1$	17(44)	17(19)	18(19)	17(18)	17(18)	18(21)	19(21)	18.6
CCA3	$2^5 3^1$	28(70)	27(34)	29(33)	26(28)	30(31)	29(37)	30(37)	28.1
CCA4	$2^6 3^1$	41(102)	41(47)	41(52)	38(39)	41(46)	44(58)	46(58)	42.3
CCA5	$2^5 3^1$	60(142)	59(69)	58(69)	54(56)	60	67(80)	66(80)	34.0
CCA6	$2^3 3^1$	469	192	196	156	180	-	210	137.6
CCA7	$2^3 3^1$	817	345	355	284	304	-	381	231.9
CCA8	$2^5 3^1$	1304	570	577	456	531	-	616	469.3

Table 8 Comparison of test suite sizes for constrained models.

Task	Size	δ	ATGT	mAETG-SAT	SA-SAT	PICT	TestCover
CCA1	3^3	$2^5 3^1$	9	10	10	10	10
CCA2	4^3	$2^3 3^1$	17	17	17	19	17
CCA3	5^3	$2^5 3^1$	26	26	26	27	30
CCA4	6^3	$2^6 3^1$	38	37	36	39	38
CCA5	7^3	$2^5 3^1$	54	52	52	56	54
CCA6	5^4	$2^3 3^1$	156	138	140	143	-
CCA7	6^4	$2^3 3^1$	284	241	251	250	-
CCA8	7^4	$2^5 3^1$	456	383	438	401	-

Table 7 reports the outcome of the same experiments run on CCA1u to CCA8u, but run on their constrained counterparts, CCA1 to CCA8. In fact, these are the same spec of the unconstrained tasks but with the addition of randomly generated forbidden tuples of arity 2 or 3, where 2-way tuples were produced with a probability of .66. Thus, a column to state the number of involved constraints has been added to the tables. To be able to quantitatively measure the constraints we introduced a function δ which accounts for every axiom after it has been converted into DNF, defined as follows:

$$\begin{array}{ll} (1) & \delta(a \wedge b) = \delta(a) \cdot \delta(b) \\ (2) & \delta(a \vee b) = \delta(a) + \delta(b) \\ (3) & \delta(x = b) = \mathit{range}(x) - 1 \\ (4) & \delta(x \neq b) = 1 \end{array}$$

For forbidden combinations, δ is equal to the constraints measure proposed in [15], which simply counts a forbidden tuple as t and multiply all the forbidden combinations. For instance, the forbidden pair $x = a, y = b$, would be represented in our approach by the constraint $\neg(x = a \wedge y = b)$, which in DNF becomes $x \neq a \vee y \neq b$ which is evaluated by δ to 2. In this case, quantities expressed with this criteria can take advantage of exponential layout, e.g., $2^5 \cdot 3^1$ will read also as five pairwise constraints plus one three-wise.

Table 8 compares the best constrained suite sizes of ATGT for pairwise and three-wise coverages with those of many other existing tools, showing the tools perform very good for these small pairwise tasks and reasonably good for three-wise tasks.

Moreover, Table 9 reports computed test suite sizes of two-fold experiments (with and without constraints) performed on a new set of six example specifications, com-

Table 9 Test suite sizes and times for asm models of real-world systems.

			ATGT				mAETG-SAT		
			cnstr.		ucnstr.		cnstr.		ucnstr.
Task	Size	δ	size	time	size	time	size	time	size
BBS	3^4	2^1	11	13.7	11	2.2			
Cruise Control	$4^1 3^1 2^4$	2^2	8	16.7	6	2.2			
Mobile Phone	$3^3 2^2$	$2^5 \cdot 3 \cdot 5$	9	27.6	11	3.4			
Spin simulator	$2^{13} 4^5$	$2^{47} \cdot 3^2$	26	186.3	5.7	23	24	0.4	25
Spin verifier	$2^{42} 3^2 4^{11}$	2^{13}	46	2745.0	33	20.2	41	11.3	33
GCC	$2^{189} 3^{10}$	$2^{37} \cdot 3^3$	26	26729.8	19	22.2	23	286.9	24

pared with mAETG-SAT [14], for which these additional data were available. This latter tool is also based on an incremental construction process, derived from the AETG algorithm, which builds the test suite one test case at a time, like our tool. Also, both our tool and mAETG-SAT rely on external solvers to ensure compliance of the built test cases with the constraints, even though only our tool allows them to be expressed as logical expressions. Moreover, our tool uses the external solver also to build test cases from test predicates, while in mAETG-SAT a different, greedy heuristic is employed. Its basic algorithm alternates phases in which AETG and SAT each search the space of possible assignments of values to factors in a configuration, where AETG proposes a new test case and SAT checks its consistency and in case it suggests some changes. The examples used in Table 9 are actually models of real-world systems subject to a number of contraints (modeled as axioms), quantitatively measured in the third column again by δ. BBS is a model of a basic telephone billing system [40], already presented in the introduction of this paper. 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 [14]. Figure 4 reports all the AsmetaL axioms translating the constraints for this model. SPIN is a well-known publicly available model checking tool [34], 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. Specifications for Spin and GCC are the same also used in [14]. Please note that in this latter series, all the examples have been run for pairwise coverage only. In all the computed test suites the tool was able to correctly handle the axioms restrictions in order to ensure complete coverage of all non-forbidden pairs, without the need to enumerate those pairs explicitly. This has been particularly helpful in those examples involving many explicit and also a few implicit (derived) constraints.

Note that the time taken by our approach (with the best strategy and collecting) is much greater than the time required by mAETG-SAT. Our tool is not optimized with respect to the time requirements, and every time a test predicated is considered for collection it calls the external model checker by exchanging files. We believe that embedding the model checker or a SAT solver in our tool would significantly improve the time performances without increasing the test suite sizes.

```
asm mobile_phone

signature:
enum domain DisplayType = {MC16|MC8|BW}
enum domain EmailType= {GV|TV|NOV}
enum domain CameraType={MP2|MP1|NOC}

dynamic monitored display :DisplayType
dynamic monitored email : EmailType
dynamic monitored camera :CameraType
dynamic monitored videoCamera : Boolean
dynamic monitored videoRingtones : Boolean

definitions:
axiom inv_1 over display, email : display=BW implies email!=GV
axiom inv_2 over display, camera : display=BW implies camera!=MP2
axiom inv_3 over camera, email : camera=MP2 implies email!=GV
axiom inv_4 over display, camera : display=MC8 implies camera!=MP2
axiom inv_5 over videoCamera, camera, display :
                    videoCamera implies (camera!=NOC and display!=BW)
axiom inv_6 over camera, videoRingtones :
                camera=NOC implies !videoRingtones
axiom inv_7 over display, email, camera :
                   !(display=MC16 and email=TV and camera=MP2)
```

Figure 4 ASM specification for the *mobile phone* example.

7 Related work

There are already a few approaches of combinatorial testing that deal with the constraints over inputs. In order to deal with constraints, some methods require to remodel the original specification, very few directly support constraints in CCIT. For instance, AETG [11, 40] requires to separate the inputs in a way that they become unconstrained, and only simple constraints of type **if then else** (or **requires** in [14]) can be directly modeled in the specification. Other methods [30] require to explicitly list all the forbidden combinations. As the number of input grows, the explicit list may explode and it may become practically infeasible to find for a user. In [6] the authors introduce the concept of (explicit) *soft constraints*: they use a method to avoid tuples if possible. In this paper we consider only hard constraints: a test is valid only if it satisfies all the constraints (explicit and implicit as well). Cohen et al. [14] found that just one tool, PICT [16], was able to handle *full* constraints specification, that is, without requiring remodeling of inputs or explicit expansion of each forbidden test cases. However, there is no detail on how the constraints are actually implemented in PICT, limiting the reuse of its technique. Others propose to deal with the constraints only after the test suite has been generated, by deleting tests which violate the constraints and then generate additional test cases for the missing combinations. By this approach, any classical algorithm for CIT may be extended to support constraints. However, this is usable only if the number of missing combinations is small and all the constraints are explicitly listed. In fact, experiments show that when the test suite is generated ignoring the constraints then the number of its tests violating the constraints can be very high, and that the number of implicit constraints can grow exponentially with the number of variables [15]. Recently, several papers investigated the use of verification methods for combinatorial testing. Hnich et al. [33] translates the problem of building covering arrays to a Boolean satisfiability problem and then they use a SAT solver to generate their solution. In their paper, they leave the treatment of auxiliary constraints over the inputs as future work. Note that in their approach, the problem of finding an *entire* covering array is solved by SAT, while in our approach only the generation of a single test case is solved by the model checker. To this respect, our approach is similar to [14, 15], where a mix of logical solvers and heuristic algorithms is used to find the final test suite. Kuhn and Okun [37] try to integrate combinatorial testing with model checking (SMV) to provide automated specification based testing, with no support for constraints. Conversely, Cohen et al. [14, 15] propose a framework to incorporating constraints into established greedy and simulating annealing combinatorial testing algorithm. They exclusively focus on handling constraints and present a SAT-based constraint solving technique that has to be integrated with external algorithms for combinatorial testing like IPO or AETG. Their framework is general and fully supports the presence of constraints, even if they can be modeled only in a canonical form of boolean formulae as forbidden tuples. In [12] the authors discuss the translation of different forms of natural language constraints into this canonical form. Their work do not deal with user defined tests and test goals, nor with the integration of combinatorial testing with other testing criteria. In our work we address the use of full constraints as suggested in [14]. Furthermore, while Cohen's general constraints representation strategy has to be integrated with an external tool for combinatorial testing, our approach tackles every aspect of the test suite generation process and strive to achieve the goals A to D stated in section 2.

8 Conclusion and future work

In this paper we presented an original approach to constrained combinatorial testing, based on formal logic. This approach aimed at giving a contribution to the testing research community by providing original solutions to reach some important goals for CCIT testing. The most important contribution being maybe its unique characteristic of providing many useful features, usually found in different and alternative tools, in an tightly integrated way. In particular, these features are: the support of Asm as the language for modeling the system under test, which enables a wide variety of systems to be analized; the ability to express complex constraints on the input domain in a compact, effective syntax, as formal predicate expressions; the ability to generate t-wise combinatorial test suites, based on many optional optimization strategies; and the ability to deal with user specific requirements on the test suite by including specific test goals and/or test cases. The practicability of this approach has also been demonstrated by implementing it in the software tool ATGT, which has been also extensively experimented, with interesting results. Plans for future works are in the direction to improve our technique along these directions. We already support enumerative and boolean domain types, but we plan to extend also to domain products (e.g. records), functions (arrays), derived functions, and discrete, finite sub-domains of integer. Converting integers to enumerations by considering each number one enumeration constant, is unfeasible unless for very small domains. We plan to investigate the partition of integer domains in sub-partitions of interest. Also, we are already working to extend the constraints support by allowing generic temporal logic expressions, which may specify requirements on the inputs evolution in time. For this reason, we chose to rely a model checker instead of simply using an SMT solver in the first place, as this latter would not be able to deal with temporal constraints. Moreover, further improvements include taking into account the output and state variables, assuming that a complete behavioral model for the given system is available, and the binding of monitored input variables to some initial value at the system start state. We plan to apply combinatorial testing to complete specifications and compare it with other types of testing like structural testing [25] and fault based testing [23] which, however, require a complete system specification, including outputs, controlled variables, and transition rules.

Acknowledgements The authors would like to thank Myra B. Cohen for kindly supporting us in the performance evaluation work by giving us access to the specifications of the SPIN and GCC case studies.

References

1. J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *International Symposium on Software Testing and Analysis*. ACM, 1996.
2. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *In TACAS 99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, London, UK, 1999. Springer-Verlag. ISBN 3-540-65703-7.*, pages 193–207, 1999.
3. A. Blass and Y. Gurevich. Pairwise testing. *Current Trends in Theoretical Computer Science*, World Scientific Press:237–266, 2004.
4. R. C. Bose and K. A. Bush. Orthogonal arrays of strength two and three. *The Annals of Mathematical Statistics*, 23(4):508–524, 1952.
5. R. Brownlie, J.Prowse, and M. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.
6. R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.
7. 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.
8. A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In B. Beckert and R. 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.
9. A. Calvagna and A. Gargantini. Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints. In J. Rushby and N. Shankar, editors, *AFM'08: Third Workshop on Automated Formal Methods (satellite of CAV)*, pages 43–52, 2008.
10. V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.
11. 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.
12. M. Cohen, M. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference-Practice and Research Techniques (TAIC PART), London, September 2007*, pages 121–130, 2007.
13. 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.
14. M. B. Cohen, M. B. Dwyer, and J. 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.
15. 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.

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. *issre*, 00:174, 1998.
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. L. de Moura, S. Owre, H. Rueß, J. R. N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
20. I. S. Dunietz, W. K. Ehrlich, B. Szablak, C. Mallows, and A. Iannino. Applying design of experiments to software testing. In I. Society, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 205–215, 1997.
21. G. Fraser and F. Wotawa. Using LTL rewriting to improve the performance of model-checker based test-case generation. In *Proceedings of the 3rd International Workshop on Advances in Model Based Testing (A-MOST 2007)*, pages 64–74, 2007.
22. G. Fraser and F. Wotawa. Ordering coverage goals in model checker based testing. In *Proceedings of the Fourth International Workshop on Advances in Model Based Testing (A-MOST 2008)*, 2008.
23. 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.
24. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99*, number 1687 in LNCS, 1999.
25. A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *JUCS*, 10(8), Nov 2001.
26. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from ASM specifications. In *ASM 2003 - Taormina, Italy, March 2003. Proceedings, LNCS 2589*, 2003.
27. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for abstract state machines. *Journal of Universal Computer Science (JUCS)*, accepted, 2008.
28. M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
29. G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, September 2004. IEEE Computer Society.
30. A. Hartman. Ibm intelligent test case handler: Whitch. http://www.alphaworks.ibm.com/tech/whitch.
31. A. Hartman. *Graph Theory, Combinatorics and Algorithms Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005.
32. A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.
33. B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.
34. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
35. Jenny Combinatorial Tool. http://www.burtleburtle.net/bob/math/jenny.html.
36. 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.
37. 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.
38. D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In I. Society, editor, *27th NASA/IEEE Software Engineering workshop*, pages 91–95, 2002.
39. 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.

40. 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.
41. A. McDowell. All-pairs testing, http://www.mcdowella.demon.co.uk/allpairs.html.
42. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA,, 1993. ISBN 0792393805.
43. K. Nurmela. Upper bounds for covering arrays by tabu. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.
44. Pairwise web site. http://www.pairwise.org/.
45. G. Seroussi and N. H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.
46. B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In C. Breckenridge, editor, *Proceedings of the Fifth International conference on Artificial Intelligence Planning Systems (AIPS)*, 2000.
47. K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.
48. A. 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.
49. 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.
50. 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.