A Logic-based Approach to Combinatorial Testing with Constraints

Andrea Calvagna[1] and Angelo Gargantini[2]

[1] University of Catania - Italy
andrea.calvagna@unict.it
[2] University of Bergamo - Italy
angelo.gargantini@unibg.it

Abstract. Usage of combinatorial testing is wide spreading as an effective technique to reveal unintended feature interaction inside a given system. To this aim, test cases are constructed by combining *tuples* of assignments of the different input parameters, based on some effective combinatorial strategy. The most commonly used strategy is two-way (*pairwise*) coverage, requiring all combinations of valid assignments for all possible pairs of input parameters to be covered by at least one test case. In this paper a new heuristic strategy developed for the construction of pairwise covering test suites is presented, featuring a new approach to support expressive constraining over the input domain. Moreover, it allows the inclusion or exclusion of ad-hoc combinations of parameter bindings to let the user customize the test suite outcome. Our approach is tightly integrated with formal logic, since it uses test predicates to formalize combinatorial testing as a logic problem, and applies an external model checker tool to solve it. The proposed approach is supported by a prototype tool implementation, and early results of experimental assessment are also presented.

1 Introduction

Verification of highly-configurable software systems, such as those supporting many optional or customizable features, is a challenging activity. In fact, due to its intrinsic complexity, formal 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 only the inputs and require they are sufficiently covered by tests. On the other hand, unintended interaction between optional features can lead to incorrect behaviors which may not be detected by traditional testing [22,33].

A combinatorial testing approach is a particular kind of functional testing technique consisting in exhaustively validating all combinations of size t of a system's inputs values. This is equivalent to exhaustively testing t-strength interaction between its input parameters, and requires a formal modeling of just the system features as input variables. In particular, 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, which has been experimentally shown to be really effective in revealing software defects [21]. A test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [27,9]. Other experimental work shown that 100% of faults are usually triggered by a relatively low degree of features interaction, typically 4-way to 6-way combinations [22]. For this reason combinatorial testing is used in practice and supported by many tools [26].

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 [18]. Covering arrays are combinatorial structures which extend the notion of *orthogonal arrays* [2]. A covering array $CA_\lambda(N; t, k, \boldsymbol{g})$ is an N x k array with the property that in every N x 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 $\boldsymbol{g} = (g_1, g_2, ...g_k)$ 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 this paper we present our approach to combinatorial testing, which is tightly integrated with formal logic, since it uses test predicates to formalize combinatorial testing as a logic problem. The paper is organized as follows: section 2 gives some insight on the topic and recently published related works. 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 presents some early results of experiments carried out in order to assess the validity of the proposed approach. Finally, section 6 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 [15]. There is also a web site [26] devoted to this subject. We would like to classify them according to Cohen et al. [7], as:

a) *algebraic* when the CA is given by mathematical construction as in [20]. These theoretic based approaches generally leads to optimal results, that is minimal sized CA. Unfortunately, no mathematical solution to the covering array generation problem exists which is generally applicable. Williams and Probert [31] showed that the general problem of finding a minimal set of test cases that satisfy $t-$wise coverage can be NP-complete. Thus, heuristic approaches, producing a sub-optimal result are widely used in practice.

b) *greedy* when some search heuristic is used to incrementally build up the CA, as done by AETG [5] or by the In Parameter Order (IPO)[27]. This approach is always applicable but leads to sub-optimal results. Typically, only an upper bound on the size of constructed CA may be guaranteed. The majority of

existing solutions falls in 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 [6,25].

Besides 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 the previous work on this topic by Lott et al. [23]:

Ability to state complex constraints. This issue has been recently investigated by Cohen et al. [7] and recognized as a highly desirable feature of a testing method. Still according to Cohen et al., just one tool, PICT [8], was currently found 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. Most tools require the user to re-write the specification in a way that the inputs are separated and unconstrained, but when combined the satisfy the constraints. AETG [5] and the TestCover [28] service follow this approach. Other tools, like the IBM Whitch [16], require the user to explicitly list all the forbidden combinations. Note that if constraints on the input domain are to be taken into account then finding a valid test case becomes an NP-hard problem [3]. In our work, not only we address the use of full constraints as suggested in [7] but we feature the use of generic predicates to express constraints over the inputs (see section 4 for details). 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.

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* [15] 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.

Integration with other testing techniques Combinatorial testing is just *one* testing technique. The user may be interest to integrate results from many testing techniques, including those requiring very complex formalisms (as in [14,12,11,13]). 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 pairwise interaction testing, our approach gives to the user the easy of having just one convenient and powerful formal approach for both uses.

Recently, several papers investigated the use of verification methods for combinatorial testing. Hnich et al. [19] 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. Conversely, Cohen et al. [7] exclusively focuses on handling of with constraints and present a SAT-based constraint solving technique that has to be integrated with external algorithms for combinatorial testing like IPO. Kuhn and Okun [21] try to integrate combinatorial testing with model checking (SMV) to provide automated specification based testing, with no support for constraints. In this work we investigate the integration of model checkers with combinatorial testing in the presence of constraints while supporting all of the additional features listed above.

3 A logic approach to pairwise coverage

We now describe our approach to combinatorial testing which we can classify as *logic-based* and which is supported by the *ASM Test Generation Tool* (ATGT)[3]. ATGT was originally developed to support structural [14] and fault based testing [13] of *Abstract State Machines* (ASMs), and it has been extended to support also combinatorial testing. Since pairwise testing aims at validating each possible pair of input values for a given system under test, we then 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 enumerative or boolean domain and v_1 and v_2 are two possible values of $p1$ and $p2$ respectively. The easiest way to generate test predicates for the pairwise coverage of an ASM model is to employ a combinatorial enumeration algorithm, which simply loops over

[3] A preview release of the tool is available at the following URL: http://cs.unibg.it/gargantini/projects/atgt/.

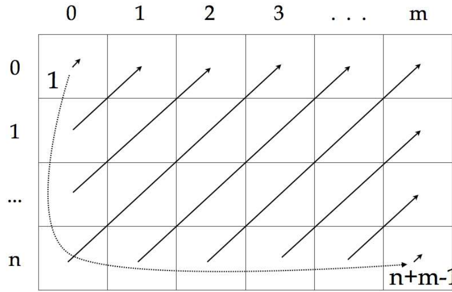

Fig. 1. antidiagonal order in combinatorial indexing of the values pairs, n and m being the ranges of two input parameters.

the variables and their values to build all the possible test predicates. Another variation of the test predicate generation algorithm we support is the *antidiagonal* algorithm, which instead has been specially devised to output an ordered set of logic test predicates (tp) 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 *antidiagonal* order, see Fig.1. Thus, generating their sequence of pair assignments such that both values always differ from previous ones. [4] This alternative way of ordering of the pairs combinations to be covered is motivated by a performance improvement it produces on the execution of our covering array generation algorithm, as will be explained later in Sect. 3.3.

In order to correctly derive assignment pairs required by the coverage we assume the availability of a formal description of the system under test. This description includes at least the listing of input parameters and respective domains (finite and discrete). The description has to be entered in the ATGT tool as an ASM specification in the AsmetaL language [29]. The description is then parsed and analyzed by our tool in order instantiate a convenient corresponding data structure. As an example, consider the following, which declares two parameters both with domain size three, without constraints:

```
asm simpleexample signature :
   enum domain D = {V1 | V2 | V3 }
   dynamic monitored p1 : D
   dynamic monitored p2 : D
```

The ASM model has to declare the domains, which currently must be either boolean or an enumeration of constants, like in the given example. The keyword *monitored* alerts the tool that the following parameter is in the set of input variables under test. Non monitored variables and variables of other types are ignored.

[4] Apart from the set's first and last pairs special cases.

3.1 Tests generation

A *test case* is a set of assignments, binding each monitored (input) variable to a value in its proper domain. It is easy to see that a test case implicitly covers as many *t-wise* test predicates as $\binom{n}{t}$, where n is the number of system's input parameters and $t = 2$ (for pairwise interaction testing) is the *strength* of the covering array. A given test suite satisfies the *pairwise* coverage iff *all* test predicates are satisfied by at least one of its test cases. Note that the smallest test suite is that in which each test predicate is covered by *exactly* one test case. Note that a test predicate in pairwise coverage binds only two variables to their values, while a test case assigns values to all the monitored variables.

By formalizing the pairwise testing by means of logical predicates, finding a test case that satisfy a given predicate reduces to a logical problem of satisfiability. To this aim, many logical solvers, like e.g. constraint solvers, SAT algorithms, SMT (Satisfiability Modulo Theories) solver, or model checkers can be applied. Our approach exploits a well known model checker tool, namely the bounded and symbolic model checker tool SAL [10]. Given a test predicate *tp*, SAL is asked to verify a *trap property* [11] which is the logical negation of *tp*: $\mathsf{G}(\mathsf{NOT}(tp))$. The *trap property* is not a real system property, but enforces the generation of a counter example, that is a set of assignments falsifying the trap property and satisfying our test predicate. The counter example will contain bindings for all monitored inputs, including those parameters missing (free) in the predicate, thus defining the test case we were looking for.

A first basic way to generate a suitable test suite consists in collecting all the test predicates in a set of *candidates*, extracting from the set one test predicate at the time, generating the test case for it by executing SAL, removing it from the candidates set, and repeating until the candidates set is empty. This approach, which according to [15] can be classified as iterative, is very inefficient but it can be improved as follows.

Skip already covered test predicates Every time a new test case s is added to the test suite, s always covers $\binom{n}{t}$ test predicates, so the tool detects if any additional test predicate *tp* in the candidates is covered by s by checking whether s is a model of *tp* (i.e. it satisfies *tp*) or not, and in the positive case it removes *tp* from the candidates.

Randomly process the test predicates Randomly choosing the next predicate for which the tool generates a test case makes our method *non deterministic*, as the generated test suite may differer in size and composition at each execution of the algorithm. Nevertheless, it is important to understand the key role played on the final test suite outcome by just the order in which the candidate test predicates are choose for processing. In fact, each time a *tp* is *turned* into a corresponding test case it will dramatically impact on the set of remaining solutions which are still possible for the next test cases. This is clear if we consider the following: the ability to reduce the final test suite size depends on the ability to *group* in each test case the highest possible number of uncovered tps.

The *grouping* possibilities available in order to build a test case starting from the tp currently selected for processing are directly proportional to the number and ranges of involved input variables, and limited by input constraint relations. Thus, for a given example, they can vary from *tp* to *tp*, and since each processing step will actually subtract to the next grouping possibilities, eventually the first step, that is the choice of first tp to process, will be the most influent, as it will indirectly impact on the whole test suite composition process.

Ordered processing of test predicates A different policy is to order the *tps* in the candidates 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 1. *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 variables assignments of t_1 have been already tested in T less times than the assignments 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.

Despite deterministic processing of the *tps* has the advantage of producing repeatable results, and we also included this option in our tool, it requires additional computational effort in order to guess the correct processing order of the test predicates, that is, that producing the best *groupings*. On the other hand, random processing strategy accounts for average performance in all suite of practical applications, and the rather small computation times easily allows for several trials to be shoot, and the best result to be statistically improved without significant additional effort.

3.2 Reduction

Even if one skips the test predicates already covered, the final test suite may still contain some test cases which are redundant. We say that a test case is *required* if contains at least a test predicate not already covered by other test cases in the test suite. We then try to reduce the test suite by deleting all the test cases which are not required in order to obtain a final test suite with fewer test cases. Note, however, that an unnecessary test case may become necessary after deleting another test case from the test suite, hence we cannot simply remove all the unnecessary test predicates 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
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 and add t to Op
2. remove all the test predicates covered by t from Tp
3. if Tp is empty then return Op else goto 1

3.3 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. Besides 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 consist in *composing* more test predicates into an *extended*, or *composed* test predicate,which specifies the values for as many variables as possible. We define a *composed* test predicate the conjoint of one or more test predicates. When creating a composed test predicate, we must ensure that we will be still 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 the propositional logic: since a sentence is *consistent* if it has a model, we can define consistency among test predicates as follows.

Definition 2. ***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 variables values so that the composition will take into account just the variables values of the test predicates we compose. The case where constraints over the model are defined will be considered in Sect. 4.

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

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.

Theorem 2. *A conjoint TP of test predicates is consistent with a test predicate tp if and only if every t in TP is consistent with tp*

Algorithm 2 Pseudo code for the main algorithm

```
C = the set of logical test predicates of the form (p1=v1 AND p2=v2), to be covered
T = the Test Suite initially empty
0. reset usage counts for bindings of all parameters.
1. if C is empty then return T and stop
2. (optional) sort the tps in C according to their novelty or shuffle C
3. pick up the first tp, P, from C
4. try composing P' by joining P with other consistent test predicates in C
5. run SAL trying to prove the trap property G(not(P'))
6. convert resulting counter example into the test case tc, and add tc to T
7. remove from C all tps in P' and all additional tps covered by tc
8. update usage frequencies for all covered tps.
9. goto step 1
```

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 still uncovered tps as possible. The heuristic stage of this approach is in the process of extracting from the pool of *candidate* tps the best sub-set of consistent tps to be joined together into TP. Than, the resulting composed test predicate is in turn is used to derive a new test case by means of a SAL counterexample.

3.4 Composing and ordering

The initial ordering of the predicates in the candidate pool may influence the later process of merging many pairwise predicates into an extended one. In fact, the candidates tps for merging are searched sequentially in the candidates pool. The more diversity there will be among subsequent elements of the pool and 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 pairwise predicates have been successfully compacted into the same test case and the less number of test cases will be needed to have a complete pairwise coverage.

There are more than one strategy we tested in order to produce a effective ordering of the predicates, to easy the merging process. In the implemented tool one can choose the test predicate at step 2 by the following several *pluggable* policies which impact on the efficiency of method. By adopting the *random* policy, the method randomly chooses at step 2 the next test predicate and check if it is consistent. By *novelty* policy the method chooses the most novel test predicate and try to combine it with the others already chosen. The resulting whole process is described in Alg. 2.

4 Adding Constraints

We now introduce the constraints over the inputs which we assume are given in the specification as axioms in the form of boolean predicates. For example for the well known asm specification example Basic Billing System (BBS) [23], we introduce an axiom as follows:

axiom inv_calltype **over** billing, calltype :
  billing = COLLECT implies calltype != INTERNATIONAL

To express constraints we adopt the language of propositional logic with equality (and inequality)[5]. Note that most methods and tools admit only few templates for constraints: the translation of those templates in equality logic is straightforward. For example the **require** constraint is translated to an *implication*; the **not supported** to a *not*, and so on. Even the method proposed in [7] which adopt a similar approach to ours prefer to allow constraints only in a form of forbidden configurations [17], 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. Our approach allows the designer to state the constraints in the form he/she prefers. For example, the model of mobile phones presented in [7] has 7 constraints. The constraint number 5 states that *"Video camera requires a camera and a color display"*. In [7], 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)

A constraint may not only relate two variable values (to exclude a pair), but it can contain generic bindings among variables. Any constraint models an explicit binding, but their combination may give rise to complex implicit constraints. 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 our specification. While others [4] distinguish between forbidden combinations and combinations to be avoided, we consider only forbidden combinations, i.e. combinations which do satisfy the axioms. Finding a valid test case becomes with the constraints a challenge similar to finding a counter example for a theorem or proving it. For this reason verification techniques are particularly useful in this case and we have investigated the use of the bounded and symbolic model checkers in SAL.

To support the use of constraints, they must be translated in SAL and this requires to embed the axioms directly in the trap property, since SAL does not support assumptions directly. The trap property must be modified to take into account the axioms. The general schema for it becomes:

[5] SAL, as other SMT solvers, has decision theories for linear arithmetic, uninterpreted functions, etc.. However, since we consider only inputs with enumerative domains, users can only write constraints as logic propositions with equality at most.

```
G(<AND axioms>) => G(NOT(<test predicate>))           (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 consistent 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. Inconsistent axioms must be considered as a fault in the specification and this must be detected and eliminated. For this reason when we start the generation of tests, if the specifications has axioms, we check that the axioms are consistent by trying to prove:

```
G(NOT <AND axioms>)
```

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 *unfeasible.*

Definition 3. *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 unfeasible. Let tp be the pair of assignments* $v_1 = a_1 \wedge v_2 = a_2$*, we say that this pair is unfeasible.*

An unfeasible pair of assignments represents a set of invalid test cases: all the test cases which contain this pair are invalid. Our method is able to detect infeasible pairs, since it can actually prove the trap property derived from it. The tool finds and marks the infeasible pairs, and the user may derive from them invalid test cases to test the fault tolerance of the system.

For example, the following test predicate results infeasible for the BBS example:

calltype = INTERNATIONAL and billing = COLLECT −−−> unfeasible

Note that since the BMC is in general not able to prove a theorem, but only to find counter examples, it would be not suitable to prove unfeasibility of test

predicates. However, since we know that if the counter example exists then it has length 1, if the BMC does not find it we can infer that the test predicate is unfeasible.

4.1 Composition and constraints

By introducing constraints, Theorems 1 and 2 are no longer valid and the composition method presented in Sect. 3.3 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 *Axs* and the conjoint *TPs*, we can try to prove by using SAL:

```
G(<TPs>) AND G(<Axs>) => G(NOT(tp))
```

If this is proved, we skip *tp* since it is inconsistent with *TPs*, otherwise we can add *tp* to *TPs* and proceed.

4.2 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 pairwise assignments. Sometimes these combinations are *n* assignments to *n* variables (for example with *n=3* one could specify a 3-wise coverage) but this is not the most general case. We assume that the user defined test goals are given as generic logical predicates, allowing the same syntax as for the constraints. The user wants to obtain a test case which covers these test goals. For example, we allow the user to write the following test goal:

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

which requires to test a combination of inputs such that access is LOOP but the billing is not the CALLER and the calltype is not LOCALCALL. A counter example for the trap property derived from the test goal loop is again a test case that covers the test goal.

Besides user defined test goals, we allow also user defined test cases (sometimes called *seeds*) The user may have already some tests cases to be considered, 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;
```

		one tp at the time								collect + reduction		
spec	mc	no opt	*time*	skip	+rnd	+antDg	+nov	**red**	*time*	no rnd	rnd	*time*
TCAS	SMC	837	*310*	352	113	300	280	241	*113*	107	100	*45*
TCAS	BMC	837	*352*	452	120	452	420	419	*200*	110	101	*48*
three_four	SMC	48	*22*	37	20	37	30	10	*15*	19	10	*10*
three_four	BMC	48	*16*	37	23	37	28	10	*18*	20	10	*10*

Table 1. Test suite size and time (in sec.) using several options

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 in the set of test predicates to be covered. Than 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.

5 Early evaluation

We have experimented our method in three different ways. First we explored the impact of the run-time configuration options on the tool itself. The second set of experiments aimed at exploring the tool's combinatorial algorithm performance. And the last set of experiment assessed the validity of our approach in the presence of constrained models. Experiments were executed on a PPC G4 1,5Mhz processor, equipped with 1Gbyte of physical memory.

We report in Tab. 1 the results of the experiments regarding the use of all the options presented in this paper applied to the case study *TCAS*, which models a Traffic Collision Avoidance System described in [21] and to the benchmark model *three_four* which contains three variables with four possible values each. If no optional features are selected (no opt column) the test suite will contain as many tests as the test predicates. Still covering one test predicate at the time, if one applies the skip policy and either the random, or the anti diagonal or the novelty technique, the size of the test suite and the time taken is reduced. However, if one applies the reduction algorithm (**red** column) we found no difference among which other technique is applied before the reduction. The best results are obtained applying the collect and the reduction. In this case we found the best results when applying the random strategy (rnd column). While it is widely recognized that the Bound Model Checker (BMC) performs better then the Symbolic Model Checker (SMC) when searching for counter example, we found the opposite: the SMC generally performed better than BMC.

In Table 2 we compared the size of the test suites obtained applying our best method with results from several tools available in the literature [8][18]. This new set of experiments was designed in order assess the scalability of the combinatorial algorithm we implemented. Note that we adopt below the exponential symbolic notation used in [18] to represent the problem domain size. Reported

Task	ATGT	AETG [5]	PairTest [27]	TConfig [32]	CTS [16]	Jenny [30]	AllPairs [24]	PICT [8]
3^4	11	9	9	9	9	11	9	9
3^{13}	23	15	17	15	15	18	17	18
$4^{15}3^{17}2^{29}$	62	41	34	40	39	38	34	37
$4^1 3^{39} 2^{35}$	65	28	26	30	29	28	26	27
2^{100}	25	10	15	14	10	16	14	15
4^{10}	37		31	28	28	30		
4^{20}	54		34	28	28	37		
4^{30}	68		41	40	40	41		
4^{40}	88		42	40	40	43		
4^{50}	104		47	40	40	46		
4^{60}	114		47	40	40	49		
4^{70}	127		49	40	40	50		
4^{80}	136		49	40	40	52		
4^{90}	143		52	43	43	53		
4^{100}	151		52	43	43	53		
10^{20}	367	180	212	231	210	193	197	210

Table 2. Combinatorial performance comparison.

results clearly show that our algorithm performed worse than the others for every benchmark. Despite the performance is still reasonable for simpler tasks, it decays rapidly with the increase of the task size. Also, the time to generate the tests (which are not reported but are in the order of few tens to many hundreds of seconds) are significantly greater than the average time taken by other tools, mainly due to the fact that we iteratively call an external program (SAL) by exchanging files. However, this problem could be alleviated easily with an hardware upgrade. As far as the time taken by the generation of tests is kept within minutes, we believe that it is not an issue, since this test suite generation is done only once. Note that the pure numeric performance of the combinatorial algorithm was never meant to be an objective of primary importance in our intentions, being it really to explore the viability of using model checkers for testing purposes. The current ATGT combinatorial test generation algorithm has been devised purposely to support us to this aim only, that is, being more flexible and integrated with other testing techniques, as explained earlier in this paper. We are very confident that its combinatorial efficiency could still be improved significantly if desired, although we intentionally left this issue outside the scope of this paper.

In table 3 results for constrained asm specifications are reported. All the example's domains used in this case were subject to a number of restrictions in the form of asm axioms, quantitatively reported in the third column. Computed test suite sizes with and without constraints are reported. In this set of experiments we considered three example specifications taken from the literature. BBS is a basic billing system presented in [23] that processes telephone call data

with four call properties, and in which every property has three possible values. 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 [7]. In all the computed test suites the tool was able to correctly handle the axioms restrictions in order ensure complete coverage of all non-forbidden pairs, without the need to enumerate those pairs explicitly. This has been particularly helpful in the last example, involving many explicit and also a few implicit (to be derived) constraints. Size of computed test suite is also the least possible in the presence of the constraints, and equals the size of the test suite computed in [7]. Note that in two of the considered cases the test suite size increased with respect to their unconstrained equivalent, while it decreased in the last one, where constraints where more pervasive. Figure 2 reports all the AsmetaL axioms translating the constraints for this model.

Name	Task size	# of constraints	constrained size	unconstrained size
BBS	3^4	1	13	11
Cruise Control	$4^1 3^1 2^4$	2	8	6
Mobile Phone	$3^3 2^2$	7	9	11

Table 3. Test suite sizes for constrained models

```
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)
```

Fig. 2. constraints for mobile phone example

6 Conclusions and future work

In this paper we presented a logic based approach to combinatorial testing, supporting a number of original features, to the best of our knowledge, which have been also implemented in the software tool ATGT. These contributions include: support for Asm specifications, support for expressing constraints on

the input domain as formal predicate expression on the input variables, integrated support for multiple types of coverages evaluation over the same system specification, support for combinatorial test case generation through selectable random or deterministic strategies, and support for user-level customization of the derived combinatorial test suite by import or banning of specific set of test cases. This work is currently on going and early evaluation results have been presented in this paper. We believe that our approach satisfies, even though not completely, the three goals stated in the introduction: ability to state complex constraints, ability to deal with user specific requirements on the test suite, and integration with other testing technique.

We plan to improve our technique along these directions. We already support enumerations and boolean, 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. We plan to extend the language of the constraints by allowing generic temporal logic expressions, which may specify how the inputs evolve. For this reason, we chose the model checker SAL instead of a simple SMT solver in the first place: it is able to deal with temporal constraints and transition systems. Moreover, further improvements can 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 [12] and fault based testing [13], which, however, require a specification complete of outputs, controlled variables, and transition rules.

References

1. Joanne M. Atlee and Michael A. Buckley. A logic-model semantics for SCR software requirements. In *International Symposium on Software Testing and Analysis*. ACM, 1996.
2. R. C. Bose and K. A. Bush. Orthogonal arrays of strength two and three. *The Annals of Mathematical Statistics*, 23(4):508–524, 1952.
3. Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.
4. Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.
5. 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), 1997.
6. 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.

7. Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM Press.
8. J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.
9. 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.
10. Leonardo de Moura, Sam Owre, Harald Rueß, John Rushbyand N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
11. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99*, number 1687 in LNCS, 1999.
12. A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *JUCS, Volume 10 Number 8 (Nov 2001)*, 2001.
13. Angelo 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.
14. Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using spin to generate tests from ASM specifications. In *ASM 2003 - Taormina, Italy, March 2003. Proceedings, LNCS 2589*, 2003.
15. Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
16. Alan Hartman. Ibm intelligent test case handler: Whitch, http://www.alphaworks.ibm.com/tech/whitch.
17. Alan Hartman. *Graph Theory, Combinatorics and Algorithms Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005.
18. Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.
19. Brahim Hnich, Steven David Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.
20. Noritaka Kobayashi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Non-specification-based approaches to logic testing for software. *Journal of Information and Software Technology*, 44(2):113–121, February 2002.
21. D. Richard Kuhn and Vadim Okum. Pseudo-exhaustive testing for software. In *SEW '06: IEEE/NASA Software Engineering Workshop*, volume 0, pages 153–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
22. D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.
23. 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.
24. A. McDowell. All-pairs testing, http://www.mcdowella.demon.co.uk/allpairs.html http://www.mcdowella.demon.co.uk/allpairs.html.

25. K. Nurmela. Upper bounds for covering arrays by tabu. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.
26. Pairwise web site. http://www.pairwise.org/.
27. K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.
28. TestCover tool. http://www.testcover.com/.
29. The asmeta project. http://asmeta.sourceforge.net.
30. Jenny Combinatorial Tool. http://www.burtleburtle.net/bob/math/jenny.html.
31. Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *AICCSA*, pages 304–312. IEEE Computer Society, 2001.
32. A.W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the 13th International Conference on the Testing of Communicating Systems (TestCom 2000)*, August 2000, pp. 59-74.
33. Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.