Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints

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

Angelo Gargantini
Dip. Metodi Matematici e Ingegneria dell'Informazione
University of Bergamo - Italy
angelo.gargantini@unibg

ABSTRACT

In this paper we describe an approach to use formal analysis tools in conjunction with traditional testing to improve the efficiency of the test generation process. We have developed a technique for the construction of combinatorial test suites, featuring expressive constraints over the models under test and cross coverage evaluation between multiple coverage criteria: combinatorial, structural and fault based. Our approach is tightly integrated with formal logic, since it uses formal logic to specify the system inputs (including the constraints), test predicates to formalize testing as a logic problem, and applies the SAL model checker tool to solve it, and hence to generate combinatorial test suites. Early results of experimental assessment are presented, supported by a prototype tool implementation.

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 [27, 35]. To this aim, combinatorial interactive testing (CIT) techniques [12, 20, 27] can be effectively applied in practice [2, 31, 26]. CIT consists in 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 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.

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. 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 [25]. A test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [32, 13]. Other experimental work shown that usually 100% of faults are already triggered by a relatively low degree of features interaction, typically 4-way to 6-way combinations [27], and that the testing of all pairwise interactions in a software system finds a significant percentage of the existing faults [13]. Dunietz *et al.* [15] 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 [6] 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 [29]. However, as explained in Section 2, 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 investigate the use of CIT in the presence of constraints, and in particular with constraints over how input values can change over time, or briefly, temporal constraints. Our approach is particularly useful (but not limited to) if one wants to apply CIT to reactive systems, for which temporal constraints play a fundamental role.

We argue that a mixed approach, where both testing and formal analysis (model checking) tools are used in conjunction, could be of advantage in order to balance the required efforts over time. Specifically, we devise an approach where formal modeling of 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. At start-up stage, formal modeling of just the input domain allow for exhaus-

tive exploration of features interactions, using for instance combinatorial testing. This lets us achieve a high degree of confidence on the system correctness with relatively little effort. Meanwhile, or later in time, the same model can be extended to include the actual system's behavioral description. At this second stage, temporal properties which express constraints over sequences of inputs over time can also be checked. That is, feedback from the model checker can be used in order to customize the combinatorial test suites, allowing only valid tests, with respect to the requirements on the dynamics of the system parameters. Thus, still improving the significance of the resulting test process. Moreover, the behavioral model of the system can be also used as an oracle to compute expected outputs to each test, thus enabling also fully automated evaluation of the test process. In this context, we present a technique to express constraints over the dynamics of a system and to use them to build a valid combinatorial test suite. This technique has been implemented in a tool (ATGT)[1] designed in order to exploit model checkers to generate tests. Considering models possibly with their complete behavioral specification, allowed us to derive a combinatorial test suite and then evaluate its cross coverage with respect to structural and fault-based criteria.

The paper is organized as follows: section 2 gives some insight on the topic and recently published related works. Section 3 presents our approach, how we deal with propositional constraints, how we use the SAL model checker to generate combinatorial tests, and how we evaluate the tests. Section 4 explains how we incorporate temporal constraints over the input domain. Section 5 evaluates early results on some case studies carried out in order to assess the correctness of the proposed approach. Finally, section 6 draws our conclusions and points out some ideas for future extension of this work.

2. RELATED WORK

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 [20]. There is also a web site [29] devoted to this subject and several automatic tools are commercially [8] or freely available [32]. 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 methods really useful in practice in more general situations, like i.e. the ability to handle constraints on the input domains. In a previous paper [7] we have identified the following requirements for a effective combinatorial testing tool, extending the previous work on this topic by Lott et al. [28]:

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* [20] 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 interest to integrate results from many testing techniques, including those requiring very complex formalisms (as in [19, 18, 17, 16]). 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.

C. Constraints support. A third 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. [9] and recognized as a highly desirable feature of a testing method. Note that the general problem of finding a minimal set of test cases that satisfies t-wise coverage can be NP-complete [34, 30]. If constraints on the input domain are to be taken into account, even the generation of a single test can be NP-complete, since it can be reduced in the most general case to a satisfiability problem.

Although no one has considered constraints over the evolution of monitored variables during the time, there are already few approaches to deal with the non temporal (or propositional) constraints over the inputs.

In order to deal with constraints, some methods require to remodel the original specification. For instance, AETG [8, 28] requires to separate the inputs in a way they become unconstrained, and only simple constraints of type **if then else** (or **requires** in [9]) can be directly modeled in the specification. Other methods [21] require to explicitly list all the forbidden combinations. As the number of input grows, the explicit list may explode. In [3] the authors introduce the concept of *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 the constraints. Cohen et al. [9] found that just one tool, PICT [11], was able to handle *full* constraints specification, that is, without requiring remodeling of inputs or explicit expansion of each

[1]ATGT tool is available for download at: http://cs.unibg.it/gargantini/projects/atgt.

forbidden test cases. However, there is no detail on how the constraints are actually implemented in PICT, limiting the reuse of its technique.

Cohen et al. [9] propose a framework to incorporating constraints into established greedy and simulating annealing combinatorial testing algorithm. Their framework is general and fully supports the presence of constraints, even if they can be modeled only as forbidden tuples.

Recently, several papers investigated the use of verification methods for combinatorial testing. Hnich et al. [24] 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. Kuhn and Okun [25] try to integrate combinatorial testing with model checking (SMV) to provide automated specification based testing, with no support for constraints.

In our previous work [7] we have investigated the integration of model checkers with combinatorial testing in the presence of (propositional) constraints while supporting all of the additional features listed above. In [7], not only we address the use of full constraints as suggested in [9] but we feature the use of predicates to express constraints over the inputs (see section 3 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.

In this paper we extend and integrate our previous work with some modifications by considering t-wise coverage, dealing with temporal constraints over the dynamics of inputs, which has not yet been investigated, to the best of our knowledge, and evaluating combinatorial tests against structural and fault-based test suites (cross coverage evaluation).

3. LOGIC-BASED APPROACH

In this section we present the logic-based approach presented in [7] with some integrations and extensions, like the n-wise coverage. The technique is supported by the *ASM Test Generation Tool* (ATGT). ATGT was originally developed to support structural [19] and fault based testing [16] 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. Similarly, the n-wise coverage can be modeled by a set of test predicates, each of the type:

$$p_1 = v_1 \wedge p_2 = v_2 \wedge \cdots \wedge p_n = v_n$$

where $p_1, p_2 \ldots p_n$ are n inputs and $v_1, v_2 \ldots v_n$ are their values, such that every possible combination of the n input variables with their values is taken into account. Please note that to reach complete n-wise coverage this has to be true for each n-tuple of input parameters of the considered system.

The easiest way to enumerate the test predicates required for the n-wise coverage of an ASM model is to employ a combinatorial enumeration algorithm, which simply loops over the variables and their values to build all the possible test predicates.

In order to correctly generate the test predicates required by the coverage we assume the availability of a formal description of the system under test. This description should include at least the input parameter domains[2]. The description has to be entered in the tool as an ASM specification in the AsmetaL language [33]. We use as case study the well known example Cruise Control (CC) [1], whose AsmetaL specification in shown in Listing 1. The CC has 4 boolean monitored variables, one monitored variable with 3 possible values, and, for instance, the collection of test predicate for the pairwise coverage count 48 predicates. These are the combinatorial explosion of all assignments for each of the five possible subsets of two distinct parameters of CC. The the four-wise coverage set for the same example count 112 test predicates. They can be obtained by enumerating all the possible assignments for the following parameter subsets:

fast	igOn	brake	engRun
fast	igOn	brake	lever
fast	igOn	engRun	lever
fast	brake	engRun	lever
igOn	brake	engRun	lever

Table 1: parameters combinations

This activity (step 1) is carried out by the *test predicate generator* of Fig. 1 showing the process proposed by our method and implemented in ATGT.

By formalizing the n-wise testing by means of logical predicates, finding a test that satisfies a given predicate reduces to a logical problem of finding a complete [3] model for a logical formula. To this aim, many techniques like constraint solvers, can be applied. Our approach exploits a well known model checker tool, namely the bounded and symbolic model checker tool SAL [14]. Given a test predicate *tp*, SAL is asked to verify a *trap property* [17] which 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 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 we were looking for.

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

[3]We say that a model is *complete* if it assigns a value to every input variable. We informally call this model *assignment*

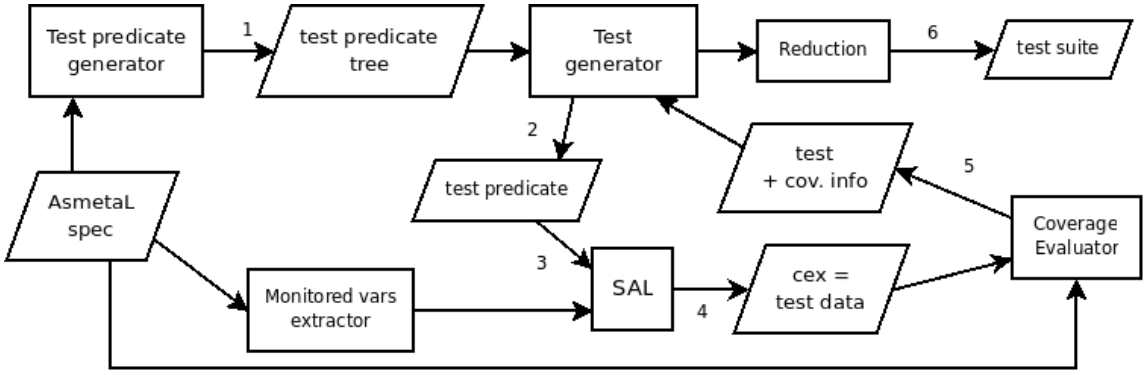

Figure 1: the process of test suite generation based on SAL MC

```
asm cruiseControl
import StandardLibrary
//UNIVERSES and FUNCTIONS
signature:
  enum domain CCMode =
        {OFF| INACTIVE|CRUISE|OVERRIDE}
  enum domain CCLever =
        {DEACTIVATE| ACTIVATE|RESUME}
  dynamic controlled mode : CCMode
  dynamic monitored lever : CCLever
  dynamic monitored igOn : Boolean
  dynamic monitored engRun : Boolean
  dynamic monitored brake : Boolean
  dynamic monitored fast : Boolean
definitions:
// AXIOMS: ADDED LATER
// RULES:
main rule r_CruiseControl =
 if not igOn then mode := OFF
 else if not engRun then mode:= INACTIVE
 // igOn and engRun
 else par
  if mode = OFF then mode := INACTIVE endif
  if mode = INACTIVE and not brake and not fast
      and lever = ACTIVATE then
        mode := CRUISE
  endif
  if mode = CRUISE then
    if fast then mode := INACTIVE
    else if brake or lever = DEACTIVATE then
                  mode := OVERRIDE endif endif
  endif
  if mode = OVERRIDE and not fast and not brake
      and (lever = ACTIVATE or lever = RESUME) then
        mode := CRUISE
  endif endpar
endif endif

default init s1:
// INITIAL STATE: ADDED LATER
```

Listing 1: AsmetaL specification of Cruise Control

```
cruiseControl: CONTEXT = BEGIN

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

 monitored : MODULE = BEGIN

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

 TRANSITION igOn' IN {true, false};
      fast' IN {true, false};
      engRun' IN {true, false};
      brake' IN {true, false};
      lever' IN {DEACTIVATE, ACTIVATE, RESUME};
 END;

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

Listing 2: SAL specification of Cruise Control

The steps we actually perform to generate a suitable test suite are depicted in Fig. 1. We randomly extract a test predicate *tp* (step 2) from the set of all the test predicates previously generated. The user may select only a subset of the test predicates for generation or include some extra tps (as explained in [7]): we call *candidates* all the test predicates to be considered. Then (step 3) we build the SAL specification by considering the inputs of the original model and the trap property. The SAL translation of CC is shown in Listing 2.

We run SAL to obtain a counter example, i.e. an assignment of every input which satisfies *tp*. Without constraints such counter example always exists, it represents the test, and it is called *test data* (step 4). The test data produced by SAL is then completed to compute the expected values for the controlled variables to obtain a real test. Indeed, since the SAL model ignores the rules and the controlled variables and it considers only the inputs and their domains, the counter example does not contain the expected values for the controlled variables. The test is also evaluated to check if it covers other candidates, i.e. if it satisfies other test predi-

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

Figure 2: Test suite reduction algorithm

cates (step 5). Finally the test is added to the test suite and the test predicates covered are removed from the candidates until the set becomes empty. This approach, according to [20], can be classified as iterative, since the test suite is built one test at the time.

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 than 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 Fig. 2, which finds a test suite with the minimum number of required test cases by simply looking at which test predicates are covered by each test in the original test suite. This reduction technique is applied in step 6 of the Fig. 1.

3.1 Propositional 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[4]. 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 [9] which adopt a similar approach to ours prefer to allow constraints only in a form of forbidden configurations [22], since it relies for the actual tests generation on external tools. Our approach allows the designer to state the constraint of a forbidden combination as a *not* statement. Moreover, we support constraint that not only relate two variable values (to exclude a pair), but can contain generic bindings among variables. Note that 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 the specification. While others [4] distinguish between forbidden combinations and combinations to be avoided, we consider only forbidden combinations, i.e. combinations which do not satisfy the axioms. 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:

$$G(a_1 \wedge a_2 \wedge .. \wedge a_n) \Rightarrow G(\neg tp) \quad (1)$$

A counter example of the trap property 1 is still a valid test data. 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. 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 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:

[4]To be more precise, we use propositional calculus, boolean and enumerative types for variables, and equality

$$G(\neg(a_1 \wedge a_2 \wedge .. \wedge a_n)) \quad (2)$$

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 1. *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. Let tp be the n-wise test predicate* $p_1 = v_1 \wedge p_2 = v_2 \ldots p_n = v_n$*, 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 this combination are invalid. Our method is able to detect infeasible assignments, since it can actually prove the trap property derived from it. The tool finds and marks the 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 results infeasible for the CC example, since the engine cannot run when the ignition is off:

engRun = true AND igOn = false −−−> 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 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, if the BMC does not find it then we can infer that the test predicate is infeasible. Note that a test 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. 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.

4. TEMPORAL CONSTRAINTS

Until now, we have considered only constraints which bind the input values at the same time, e.g. a variable cannot have a value if another has another value. Now we consider systems which evolve during their operations in a discrete way, step by step. At every step every variable can have a different value from the value it had before. In particular, the monitored variables are free to non deterministically change from one step to the next one. However, some axioms may limit how inputs evolves due some external constraints of the environment. For these systems, a test is no longer a simple assignment to every input variable to one of its possible values, but it becomes a sequence of assignments, where every assignment denotes a state. In this case we refer to a test as *test sequence*. We consider in this paper three kinds of temporal constraints: *initial value*, *next value*, and *one input assumption* (OIA). These three kinds of constraints are the typical types of assumptions most formal notation for reactive systems permit about the models under test. The OIA is useful to model asynchronous systems, which process an input event at the time.

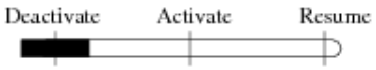

Figure 3: the lever of a Cruise Control

Initial and next value. The first two constraints state how a *single* input can evolve during the normal operation of the system. Consider for example the lever for a cruise control system depicted in Fig. 3. The lever is initially in the DEACTIVATE position and from that position it can only become ACTIVATE. From the ACTIVATE position it may become RESUME or DEACTIVATE again.

If the tester wants to use the test generated by our method to test the actual system for conformance with the requirements, he/she must use the actual interface to enter the input values of the test cases. For example, if a test case requires that lever = RESUME, the user must start with lever = DEACTIVATE, than lever = ACTIVATE and finally lever = RESUME. The application of *one* test case has required three steps, but the testing method considered so far is not aware of this and it cannot take advantage: if another test requires the user to switch lever = ACTIVATE, the tester must go back to the initial state and start again the application of the new test case, although this kind of combination of inputs has already been tested. To consider this kind of constraints we extend our method as follows. First, the tester will add the constraints in the specification. Initial values are set by the following initialization in the AsmetaL specification.

```
default init s1:
  function mode = OFF
  function lever = DEACTIVATE
  function igOn = false
  function engRun= false
  function brake = false
  function fast = false
```

The next value constraint can be specified by using the special library function *next*. For example, for CC a constraint is:

```
axiom lever_deact : lever = DEACTIVATE implies
  next(lever = DEACTIVATE) or next(lever = ACTIVATE)
```

While the initial values are translated into SAL by the INITIALIZATION clause, the next value constraints are translated in SAL in *Linear Temporal Logic* (LTL)[5]. The trap

[5]The SAL model checkers use LTL (Linear Temporal Logic)

property 1 will contain in $a_1 \ldots a_n$ not only the propositional axioms, but also the temporal axioms. For example, to express the constraint when the lever is in the DEACTIVATE position, the trap property will contain the following axiom:

```
G( ... AND lever = DEACTIVATE =>
  (X(lever = DEACTIVATE) OR X(lever = ACTIVATE))) ....
```

One Input Assumption. The One Input Assumption constraint, taken form [23], allows at most one monitored variable to change from one state to the next. The expression in AsmetaL of such constraint in terms of next values is very complex and it can be error prone. We decided to allow the user to simply set this constraint as a preference of the generation method. The translation in SAL differs from the next constraints too: instead adding a complex axiom, we modify the TRANSITION clause in the SAL specification as follows, thus allowing only one input to change at the time:

```
TRANSITION
  [ true --> igOn' IN {true, false};
  [] true --> fast' IN {true, false};
...
  ]
```

In the presence of temporal constraints, the counter example produced by SAL will contain several states. We extend the definition of test: a **test data sequence** is a sequence of assignments, whose first assignment is compatible with the initial value constraint and every pair of consecutive assignments is compatible with the temporal constraints.

We have now to take the whole counter example produced by the model checker as a test sequence: a test case is now a test data sequence with a possible number of states greater than 1. Furthermore, adding this kind of constraints limits (but it does not exclude) the use of the Bounded Model Checker (and any other constraint solver or SAT based technique). Indeed the BMC is no longer able to prove that a test predicate is infeasible, since not finding a counter example of a given finite length does not necessarily imply that the trap property is true: it can be the case that the counter example is longer than the BMC depth. We are investigating the use of induction and a conservative use of the bound for counter examples to be able to induce infeasibility from the non existence of a counter example by using BMC. Meanwhile, if the BMC does not find a counter example for a test predicate, we do not mark that test predicate as unfeasible, we warn the user, and we invite to run the SMC.

5. EXPERIMENTS

We have applied our technique to two case studies: the Cruise Control (CC) , the Safety Injection System (SIS), a simplified version of the system described in [10]. Table 2 reports some significant data about them. The *input size* is the product of the input domain sizes. The notation n^m means that the specification contains m variables each with n possible values. The table reports the number of controlled variables (# C), the number of rules (# R), the number of axioms (*#axs*: propositional and temporal constraints, which include for SIS the OIA), and the number of the test predicates obtained from the n-wise combinatorial coverage. For CC some test predicates were proved unfeasible.

	input size	# *C*	# *R*	# *axs*	*#tp n-wise*			
					2	*3*	*4*	*5*
CC	$2^4 \cdot 3$	1	6	5	48	104	112	48
	unfeasible				3	19	38	24
SIS	$2^2 \cdot 3$	3	4	4	16	12	-	-

Table 2: Case Studies

The first goal of our experiments has been the validation of our technique by generating the test suites for the combinatorial testing of the two case studies. We have used the two different model checkers SAL provides: the symbolic model checker (smc) and the bounded model checker (bmc). We wanted also to study the effects of the constraints over the test suites. We have modified the original specifications in order to obtain two versions: one with the constraints (denoted by +) and one without (temporal and propositional) constraints (denoted by -). Table 3 reports the total time required to generate the test suite (A) for each n-wise covering test suite, the total number of tests in the suite, the total number of states (as sum of the number of states in all the test sequences - *#sts*) before and after the reduction algorithm is applied (if the reduction has actually reduced the size of the test suite). The column (B) shows also the average time taken for every test in the test suite and it is equal to (A)/#tests. Note that the number of tests and the number of states are equal for specifications without constraints since each test has only one state.

Observing the data in Table 3 we can make the following preliminary observations. The BMC proved to be faster than the SMC, especially in the presence of constraints. The total time to generate the test suite (column A) is always lower for the BMC than for the SMC, while the average time taken per every single generated test (column B) is always lower for the BMC in the presence of constraints, while it is comparable without constraints. The number of tests and the total length is again lower for the BMC than for the SMC (without constraints the sizes are comparable again). This suggests that the BMC is more suitable to deal with constraints than the SMC: it is faster and produces smaller test suites. This is particularly true in case of short tests as in our case studies. However, the average length of the tests (not reported in Table) produced by the SMC is lower: since the counter example produced by the SMC is as short as possible, the SMC produces very short tests, which cover only few test predicates. This, on the other hand, causes the SMC to run more times than the BMC and the start-up time for SMC is longer than the BMC since it must build the complete BDD representation of the problem. SMC works better when one needs very short tests while the number of tests is not so important. The reduction technique reduced the test suite only in a few cases (column *after red.*).

The constraints caused the number of tests to decrease, but the total number of states to increase. We never found a test

as their assertion language but they also accept CTL syntax on the common fragment

		(A)	no red.		*after red.*		(B)
n-wise	mc	time secs	# tests	# sts	# tests	# sts	time x test
CC + (with constraints)							
pair	smc	10.4	6	27	5	22	1.73
	bmc	5.2	4	27	-	-	1.3
3	smc	30.9	14	58	13	53	2.21
	bmc	6.3	5	46	4	39	1.26
4	smc	50.2	19	77	-	-	2.64
	bmc	10.5	8	72	-	-	1.31
5	smc	45.2	21	83	20	81	2.15
	bmc	10.4	8	81	-	-	1.3
CC - (without constraints)							
pair	smc	18.8	21	21	-	-	0.9
	bmc	18.9	21	21	-	-	0.9
3	smc	34.8	37	37	-	-	0.94
	bmc	33.7	37	37	-	-	0.91
4	smc	41.9	46	46	-	-	0.91
	bmc	41.9	46	46	-	-	0.91
5	smc	44.5	48	48	-	-	0.93
	bmc	45.3	48	48	-	-	0.94
SIS + (with constraints)							
pair	smc	5.5	6	29	-	-	0.92
	bmc	2.5	3	19	-	-	0.83
3	smc	4.4	5	28	-	-	0.88
	bmc	3.3	4	33	-	-	0.83
SIS - (without constraints)							
pair	smc	4.8	10	10	-	-	0.48
	bmc	5.5	10	10	-	-	0.55
3	smc	6.0	12	12	-	-	0.5
	bmc	6.2	12	12	-	-	0.52

Table 3: Test suite with and without constraints

suite complaint with the constraints able to cover all the test predicates in a total number of states equal to the number of states of the test suite without constraints. Although considering the temporal constraints notably increases the total length of the test suite, we believe that it augments the usability and applicability of the test suite, as observed in Sect. 4.

5.1 Cross coverage evaluation

We have compared the coverage obtained by the combinatorial n-wise testing with the coverage obtained by the structural criteria presented in [18] and the fault based criteria presented in [16]. The results for the CC and SIS examples are reported in Tables 4 and 5, where BR is the basic rule coverage (similar to the branch coverage), CR is the complete rule coverage, UR is the update rule coverage, MCDC is the modified condition decision coverage, ASF is the associative shift fault, ENF is the expression negation fault, LNF is the literal negation fault, ORF is the (logical) operator reference fault, ST0 is stuck at false, ST1 is stuck at true, and ROF is the relational operator fault.

Table 4 shows the number of (feasible) test predicates for all the structural and fault based coverage criteria (row $\#tp$) and the number of these test predicates covered by each n-wise combinatorial test suite.

The table shows that the pairwise coverage implied a very low coverage of structural and fault based criteria for the CC example. Small improvements were obtained by increasing n of the the n-wise coverage. However, combinatorial testing was not even able to cover all the rules (BR), although the number of combinatorial test predicates is much higher than the number of test predicates for BR. We have investigated and found that the rules contain guards like the following one (see Listing 1):

if mode = INACTIVE and not brake and
not fast and lever = ACTIVATE **then** ...

which requires to be covered a particular value of the controlled variable *mode* and a particular combination of inputs. The desired input combination is covered in at least a test in the test suite, but since combinatorial testing ignores the outputs, such combination may not activate that particular rule because the controlled variable has a different value from that desired. By ignoring the controlled variables, combinatorial testing may be not able to drive the system to a critical state where a controlled variable takes a particular value. These results prove that the combinatorial testing does not always imply good structural coverage, contrary to the experiments presented in [6, 15], where, however, the structural coverage was computed against the implementations and not against the specifications as in our approach. Our experiments would confirm the results presented in [5], where the interaction test suites provided little benefit over the randomly generated tests and did not improve coverage of the requirements-based tests. On the contrary, we obtained that the combinatorial coverage implies a very high structural and fault based coverage for the SIS, as reported again in Table 4. We observed that the rules in SIS were not so dependent on controlled variables as those in CC, and this is the reason why we obtained better structural coverage for SIS than for CC. These results raise concerns on the application of combinatorial testing in the model-based domain for embedded and reactive systems, where the information about the system state is needed to build test suites achieving good structural coverage. However, the combinatorial testing may be the only model-based testing technique applicable in case the model contains only the specification of the inputs together with their domains and constraints.

On the other hand, structural and fault based criteria do not imply combinatorial testing either, as reported in Table 5, which shows the number of combinatorial test predicates covered by structural and fault based testing. For this reason we believe that combinatorial, structural and fault based criteria are complementary each other.

6. CONCLUSION AND FUTURE WORK

In this paper we have described an integrated approach to use formal analysis in conjunction with traditional testing in order to improve the efficiency of the verification&validation process. We presented an approach which is tightly integrated with formal logic, since it uses a formal notation to specify the system under test, test predicates to formalize combinatorial testing as a logic problem, and applies a model checker to solve it. We developed and presented a technique for the construction of n-wise combinatorial test suites, featuring not just simple constraints over the set of inputs but

		Structural Coverage				Fault Based							
		BR	CR	UR	MCDC	ASF	ENF	LNF	MLF	ORF	ST0	ST1	ROF
CC	*#tp*	*7*		*7*	*9*	*0*	*22*	*29*	*28*	*22*	*49*	*49*	*29*
	pairwise	3		4	5		19	11	9	19	10	28	5
	3-wise	3		3	5		19	12	10	19	10	29	5
	4-wise	3		3	5		19	13	11	19	10	30	6
	5-wise	4		4	6		19	15	11	19	20	31	6
SIS	*#tp*	*7*	*2*	*11*	*7*	*1*	*9*	*16*	*16*	*9*	*24*	*24*	*16*
	pairwise	7	2	9	7	1	9	16	15	9	24	21	15
	3-wise	7	2	10	7	1	9	16	16	9	24	24	15

Table 4: Cross coverage of n-wise combinatorial testing

			Structural Coverage				Fault Based							
		#tp	BR	CR	UR	MCDC	ASF	ENF	LNF	MLF	ORF	ST0	ST1	ROF
CC	pairwise	*45*	26	-	26	26		14	31	40	40	40	40	40
	3-wise	*85*	34	-	34	34		16	48	64	64	64	64	64
	4-wise	*74*	21	-	21	21		9	32	47	47	47	47	47
	5-wise	*24*	5	-	5	5		2	7	13	13	13	13	13
SIS	pairwise	*16*	9	5	10	9	6	3	9	10	10	10	10	10
	3-wise	*12*	4	2	5	4	2	1	4	5	5	5	5	5

Table 5: Combinatorial test predicates covered by structural and fault based testing

also over the evolution of inputs, which is the major and original contribution. Moreover, we were able to evaluate the cross coverage between combinatorial, structural and fault based coverage criteria, with very limited effort. Early results of experimental assessment have also been presented, supported by a prototype tool implementation. We found some interesting and unexpected results, since in one case combinatorial testing did not imply structural testing, in spite of different results in the literature suggest the contrary. We believe the whole proposed approach can be very successful since it still allows the tester/developer to interface to an easy black-box test tool - where only combinatorial testing can be used - and, at the same time, advantage of the (available) formal specification in order to implement a more effective test process - where structural and fault based testing can be applied.

Since our approach is based on the use of model checkers, it suffers from the state explosion problem. However, for the combinatorial testing, we consider only the inputs together with their constraints and this should keep the problem tractable. Other approaches may take advantage of a limited expressiveness of the constraints language, but with a loss of usability. We plan to investigate the combined use of the model checker for the constrained part of the model with classical algorithms for combinatorial testing for the unconstrained part to minimize the total complexity of the method.

7. 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] 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.

[3] 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.

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

[5] R. C. Bryce, A. Rajan, and M. P. E. Heimdahl. Interaction testing in model-based development: Effect on model-coverage. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 259–268. IEEE Computer Society, 2006.

[6] 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.

[7] 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.

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

[9] 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.

[10] P.-J. Courtois and D. L. Parnas. Documentation for

safety critical software. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 315–323. IEEE Computer Society Press, May 1993.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *JUCS, Volume 10 Number 8 (Nov 2001)*, 2001.

[19] 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.

[20] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

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

[22] A. Hartman. *Graph Theory, Combinatorics and Algorithms Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005.

[23] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

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

[33] The ASMETA project. http://asmeta.sourceforge.net.

[34] 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.

[35] 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.