Using Model Checking to Generate Fault Detecting Tests

Angelo Gargantini

Department of Management and Information Technology
Università di Bergamo
angelo.gargantini@unibg.it

Abstract. We present a technique which generates from Abstract State Machines specifications a set of test sequences capable to uncover specific fault classes. The notion of *test goal* is introduced as a state predicate denoting the detection condition for a particular fault. Tests are generated by forcing a model checker to produce counter examples which cover the test goals. We introduce a technique for the evaluation of the fault detection capability of a test set. We report some experimental results which validate the method, compare the fault adequacy criteria with some classical structural coverage criteria and show an empirical cross coverage among faults.

1 Introduction

Specification-based testing aims to reduce the cost of testing and to increase the reliability of safety critical systems. One benefit of a formal method is that the high-quality specification it produces can play a valuable role in software testing. For example, the specification may be used to automatically construct a suite of test sequences. These test sequences can then be used to automatically check the implementation software for faults. However, specification-based testing is not widely adopted [34], while white box or program based testing is well known and used in practice: many tools support it and software developers and testers are familiar with it. In the wake of the success of program based testing, specification-based testing criteria that mimic the coverage criteria for programs have been proposed. They are generally called structural criteria because they analyse the structure of the specification and require the coverage of particular elements (like states, rules, conditions, and so on). Examples are the Modified Condition Decision Coverage (MCDC), one of the most powerful criteria used in practice, applied to Abstract State Machines [18] or the coverage of properties and assertions for a program given by using the Assertion Definition Language (ADL) as proposed by [10].

Since the aim of software testing is to demonstrate the existence of errors, selecting tests that can reveal faults is of paramount importance. The fault detection capability of structural criteria is not definitely assessed though. Recent works hypothesize some classes of faults and analyze the fault detection capability of most used criteria with respect to these classes of faults. The analysis can be formal [28,30,31,33] and/or empirical [38]. The main result is that many coverage criteria cannot assure the detection of several fault classes. For instance, MCDC is unable to detect faults due to missing brackets in boolean expressions. Stronger coverage criteria have been introduced (as in [28]) with the aim to detect more faults, but still the relationship between coverage criteria and faults is not well established and it is infeasible to evaluate the effectiveness of a test criterion in general [22]. For example, it can be shown "that the fact that criterion C_1 subsumes criterion C_2 does not guarantee that C_1 is better at detecting faults [16]".

Other papers define testing criteria focusing on certain classes of faults, which model commonly committed mistakes. For instance, Weyuker et al. [38] introduce the meaningful impact strategies for boolean expressions to target specifically the variable negation fault that occurs when a boolean variable is erroneously substituted by its negation. Chen and Lau develop three more powerful testing strategies capable to detect several fault classes [11]. These criteria specify also the algorithms (with some possible non determinism) which can be used for test generation. Within this framework, assessing the fault detection capability of a criterion with respect to other criteria is important, since one should choose one criterion and generate the tests from it in accordance with the expected faults, although experimental data show that resulting tests are generally effective for detecting faults in other classes. The introduction of a new fault class would require the definition of new criteria capable to detect it or the investigation (formal or empirical) of the capability of existing criteria to detect it.

In this paper we introduce a novel approach which specifically aims at detecting faults in an implementation given its specification. Specifications are Abstract State Machines which are explained briefly in Section 2. We assume (as [14]) that implementations contain only relatively simple faults (*competent programmer hypothesis*) of the kinds introduced in Section 3 and that a test set which detects all simple faults will detect more complex faults (*fault coupling effect*). Our approach could appear similar to the *mutation analysis* [8], but it does not require any mutation at all. Instead, we introduce in Section 4 the detection condition for a fault and define adequacy criteria in terms of these detection conditions. In Section 5 we present a method which uses the detection conditions to generate and to evaluate fault detecting tests. This method is based on the counter example generation of the model checker SPIN [23]. Our approach makes the introduction of a new fault class, the generation of tests detecting these faults, and the evaluation of other tests easy to realize. In Section 6 we discuss experimental results, some of which were unexpected. Related work is presented in Section 7.

2 Preliminaries

2.1 Abstract State Machines

Even if the Abstract State Machines (ASM) method comes with a rigorous scientific foundation [9], the practitioner needs no special training to use the ASM method since Abstract State Machines are a simple extension of Finite State Machines, obtained by replacing unstructured "internal" control states by states comprising arbitrarily complex data, and can be understood correctly as pseudocode or Virtual Machines working over abstract data structures. A complete introduction on the ASM method can be found in [9], together with a presentation of the great variety of its successful application in different fields as: definition of industrial standards for programming and modelling languages, design and re-engineering of industrial control systems, modelling e-commerce and web services, design and analysis of protocols, architectural design, language design, verification of compilation schemes and compiler back-ends, etc. ASM theory is the basis of several languages and tools including the Abstract State Machine Language by Microsoft [6] and the AsmGofer [36].

An ASM *state* models a machine state, i.e. the collection of elements and objects the machine "knows", and the functions and predicates it uses to manipulate them. Mathematically, a *state* is defined as an algebraic structure, where data come as abstract objects, i.e. as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions*) and *predicates* (attributes or relations).

In this paper we consider only *single agent basic ASMs*, whose behavior is specified by a finite sets of so-called *transition rules* of the form

$$\mathrm{R} = \textbf{if } \varphi \textbf{ then } \textit{updates} \tag{1}$$

which model the actions performed by the machine to manipulate elements of its domains and which result in a new state. The guard φ under which a rule is applied is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to true or false. *updates* is a finite set of assignments of the form $f(t_1, t_2, \ldots, t_n) := t$, whose execution is to be understood as changing (or defining, if there was none) in parallel the value of the occurring functions f at the indicated arguments $t_1, t_2, \ldots, t_n$ to the indicated value t. A update is not *trivial* if the value of $f(t_1, t_2, \ldots, t_n)$ had a value different from the value of t before the update.

A more general schema is the conditional rule of the form:

$$\textbf{if } \varphi \textbf{ then par } \textit{R1,..,Rn} \textbf{ endpar else par } \textit{Q1,..,Qn} \textbf{ endpar endif} \tag{2}$$

The meaning is: if φ is true then execute *R1,..., Rn* in parallel, otherwise execute *Q1 ..., Qn*. If *Q1,..., Qn* are omitted (since they are optional), from a semantic view it is assumed that the else part is equal to *skip*, which is the empty rule whose meaning is: do nothing

2.2 Test Sequence

Adapting to ASMs some definitions common in literature for state transition systems [3,32], we define a *test sequence* or *test* as follows.

Definition 1. *A* ***test sequence*** *or* ***test*** *for an ASM $\mathcal{M}$ is a finite sequence of states (i) whose first element belongs to a set of initial states, (ii) each state follows the previous one by applying the transition rules of $\mathcal{M}$.*

A test sequence ends with a state, which might be not final, where the test goal is achieved. Informally, a test sequence is a partial ASM run and represents an expected system behavior.

Definition 1 assumes the use of ASM specification as test oracle, since states supply expected values of outputs. The importance of test oracles is well known, since the generation of the sole inputs (often called *test data*) does rise the problem of how to evaluate the correctness of the observed system behavior.

We define a collection of test sequences as follows.

Definition 2. *A* ***test set*** *or* ***test suite*** *is a finite set of test sequences.*

Given a predicate P over an ASM state we say that a test sequence t covers P, if t contains a state such that P is true in that state.

2.3 Structural Coverage Criteria for ASMs

We summarize the following coverage criteria, originally presented in [18]. They are compared in Section 3 with the new fault based coverage criteria.

Basic rule (BR) coverage requires that for every guard (decision) there exists a test which covers the case when the decision is taken (the guard is evaluated true at least in one state belonging to a test sequence) and when the decision is not taken (the guard is evaluated false).

MCDC requires the classical modified condition decision coverage in the masking form [12] to every guard in the ASM.

Complete rule (CR) coverage requires that for every rule, its guard is evaluated true at least once and at least one update in the rules is not trivial.

Update coverage (UC) requires that for every update (in every rule) there exist a test sequence in which the update is applied and is not trivial.

BR and MCDC can be classified as (model-based) *control oriented* coverage criteria [39] as they consider only the control flow of the model, while CR and UC can be classified as data flow coverage criteria, since they consider the value of variable before its assignment to a possibly new value (this kind of an update can be considered a new *definition*). Note that MCDC implies BR and UC implies CR.

3 Fault Classes

While test coverage criteria like the CR and UC, presented in Section 2.3, aim to detect faults in updates, in this paper we focus only on faults which may occur in the guards of the ASM specification under test. Note that a *fault* in a implementation is a cause that results in a failure [26], which is an erroneous evaluation of a guard in the implementation in our approach. We consider only faults originated by typical programmer mistakes like use of incorrect control predicates, missing conditions, and the incorrect use or order of boolean and relational operators in rule guards. These types of mistakes result in faults that regard the conditions and their operators, where with *condition* we intend atomic boolean expressions which cannot be further decomposed in simpler boolean expressions. A condition can be a boolean variable, like `overridden`, or a relational expression like `pressure > TooLow`. We exclude faults inside conditions except the incorrect use of relational operators (for instance the use of $>$ instead of $<$). We follow the notation proposed in [31]: a *literal* [1] is an occurrence of a condition inside a guard (note that a condition or a boolean variable may occur several times in the same guard). While many papers [30,37,31] assume that the boolean expressions are given in disjunctive normal form (DNF), we remove such restriction (as in [33]). We study the following fault classes:

- Operand faults (i.e. regarding the literals or sub expressions):

LNF *Literal negation fault.* An occurrence of a condition (i.e. one literals) is replaced by its negation. For example, from $a \wedge b \wedge (a \vee b)$ we obtain the following four faulty expressions: $\neg a \wedge b \wedge (a \vee b)$, $a \wedge \neg b \wedge (a \vee b)$, $a \wedge b \wedge (\neg a \vee b)$, and $a \wedge b \wedge (a \vee \neg b)$

ENF *Expression negation fault.* It consists of replacing a sub expression (but not a condition or literal) with its negation. For example, from $a \wedge b \wedge (a \vee b)$ [2] we obtain the following three faulty expressions: $\neg(a \wedge b \wedge (a \vee b))$, $\neg(a \wedge b) \wedge (a \vee b)$, and $a \wedge b \wedge \neg(a \vee b)$

MLF *Missing literal fault.* It causes the absence of one literal or condition in the formula. If the same condition occurs several times in the formula, we introduce several faults and not just one. For example, from $a \wedge b \wedge (a \vee b)$, we obtain the following four faulty expressions: $b \wedge (a \vee b)$, $a \wedge (a \vee b)$, $a \wedge b \wedge b$, and $a \wedge b \wedge a$.

ST0/1 *Stuck at 0/at 1 fault.* This is a classical hardware fault, which consists in replacing an input with 0 or with 1. In our case, it causes a replacement of a literal by *false* (ST0) or *true* (ST1). For example, from $a \wedge b \wedge (a \vee b)$, we obtain for the ST0 the following four faulty expressions: $\mathit{false} \wedge b \wedge (a \vee b)$, $a \wedge \mathit{false} \wedge (a \vee b)$, $a \wedge b \wedge (\mathit{false} \vee b)$, and $a \wedge b \wedge (a \vee \mathit{false})$. ST0U1 denotes the union of ST0 and ST1, i.e. the replacement of a literal by false and true.

- Boolean Operator faults:

ASF *Associative Shift fault.* This fault is due to the misunderstanding about operator evaluation priorities and missing brackets. For example from $a \wedge b \wedge (a \vee b)$ we would obtain by deleting the brackets $(a \wedge b \wedge a) \vee b$.

ORF *Operator Reference fault.* $'\wedge'$ is replaced by $'\vee'$ and vice-versa. $a \wedge b \wedge (a \vee b)$ would be implemented as $a \wedge b \vee (a \vee b)$, $(a \vee b) \wedge (a \vee b)$, and $a \wedge b \wedge (a \wedge b)$.

[1] A literal is sometimes called *clause* as in [33], a condition is often called *variable* especially in papers dealing with boolean specifications [31,30,37].

[2] We assume that logical binary operators are left associative, hence $a \wedge b \wedge (a \vee b)$ must be read as $(a \wedge b) \wedge (a \vee b)$

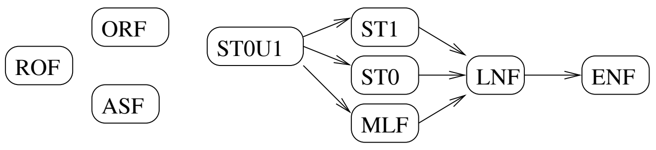


Fig. 1. boolean fault hierarchy

Furthermore we add the following fault class, which introduces faults in relational expressions with pattern $E\, op\, F$, where E and F are either arithmetic expressions or expressions of enumerative type and *op* is one of $<$, $\leq$, $=$, $>$, $\geq$, and $\neq$.

ROF *Relational Operator Fault.* Replace a relational operator by any other relational operator (note that replacing an operator with its opposite is equal to LNF). If the expression is an enumeration, then replace only $=$ with $\neq$ and vice-versa (we allow only "equals" and "differs" comparison between two enumerative values). For example, from $x \leq c$ one would obtain the following faulty expressions: $x > c$, $x \geq c$, $x = c$, $x \neq c$, and $x < c$.

We have chosen LNF, ENF, MLF, ASF, and ORF because they are the most studied faults in the literature [30,37,31]. ST0/1 faults are commonly considered a realistic model of manufacturing faults in hardware circuit testing. ROF is introduced and studied in [33] (called Relational Operator Reference Fault) and models a typical software fault.

The hierarchy among fault classes for boolean expressions have been intensively studied. For instance, empirical work [38] showed that tests generated to detect variable negation fault (our LNF) always detected expression negation faults. Kuhn proposed a rigorous approach to formally prove the existence of a hierarchy among faults in boolean specifications given in normal form [30]. The initial hierarchy proposed by Kuhn was first enriched by [37] and then by [31]. Okun et alt. [33] developed a novel analytic technique to find the hierarchy among faults of arbitrary boolean expressions, not just those in disjunctive normal form. According to the results presented in the literature, the hierarchy among the fault classes used in this paper is presented in Figure 1, where $C_1 \rightarrow C_2$ means that every test suite able to detect C_1 detects C_2 as well. In this case, we say that C_1 is *stronger* than C_2.

Note that a pair of fault classes C_1 and C_2 is proved to be independent in the hierarchy when there exists a test suite which guarantees to detect faults of C_1 but not those of C_2 and vice-versa. In our case, ORF, ROF, and ASF are independent of each other and with all the other fault classes, and MLF, ST0, and ST1 are independent of each other. This fact has practical consequences: since a test set T_1 which detects a fault C_1 does not guarantee to detect C_2 and vice-versa, one should generate a test suite for C_1 and a test suite for C_2. Therefore, one should generate a test suite for every independent fault class. However, T_1 may detect C_2 as well for the particular specification under test and the generation for C_2 may be skipped. To assess the actual fault detection capability of a test suite, we introduce in Section 5.1 a method to evaluate tests with respect to possible faults in the specification under test and regardless of the way such tests have been generated.

4 Discovering Faults

The erroneous implementation of a boolean expression φ as φ' can be discovered only when the expression $\varphi \oplus \varphi'$, called *detection condition,* evaluates to true, where $\oplus$ denotes the logical *exclusive or* operator. Indeed, $\varphi \oplus \varphi'$ is true only if φ' evaluates to a different value than the correct predicate φ. The detection condition is also called *boolean difference* or *derivative* [1].

Consider a simple rule R of an ASM specification $\mathcal{M}$:

R = **if** φ **then** *updates*

Let $\mathcal{M}'$ be the faulty implementation of $\mathcal{M}$. Assume that the guard φ of R in $\mathcal{M}$ is erroneously implemented as φ' in $\mathcal{M}'$ due to the fault F, and that rule *updates* are not all trivial. F can be detected during testing only if there exists a test sequence t containing a state s in which $\varphi \oplus \varphi'$ is evaluated to true, i.e. φ and φ' have different values in s for $\mathcal{M}$ and $\mathcal{M}'$. In this case, when we apply t, the rule R fires in $\mathcal{M}$ and performs its updates but it does not fire in $\mathcal{M}'$ or vice-versa. The predicate $\varphi \oplus \varphi'$ is the detection condition of F and it is called *test predicate* or test goal. For example, if the guard $x \leq c$ in the specification is implemented as $x < c$, then the test goal is $x \leq c \oplus x < c$ which is equivalent to $x = c$. Only a test sequence containing a state s in which $x = c$ can uncover the fault.

Let φ be a guard and C be a fault class. We denote with $F_C(\varphi)$ the set of all the possible faulty implementations of φ according to the fault class C (as explained in Section 3). The test predicates to discover the fault C in φ are the expressions $\varphi \oplus \varphi'$ with φ' in $F_C(\varphi)$. For example, if the guard is $a \wedge b$ and C is the MLF, then $F_{MLF}(a \wedge b) = \{a, b\}$ and the test predicates are the following two expressions: $(a \wedge b) \oplus a$ (which is $a \wedge \neg b$) and $(a \wedge b) \oplus b$ (which is $\neg a \wedge b$).

In case of a nested rule of kind (2), test predicates must include the guards of outer rules. Let φ be the guard of an inner rule R and $g_1, \ldots, g_n$ be the guards of the outer rules or their negation (in case of else) such that if $g_1 \wedge \ldots \wedge g_n$ holds, then R is executed (and its updates fired if φ is true). We call $g_1, \ldots, g_n$ *outer guards* of R. The test predicates to discover the fault C in φ are the expressions $g_1 \wedge \ldots \wedge g_n \wedge (\varphi \oplus \varphi')$ with φ' in $F_C(\varphi)$.

Definition 3. ***Test Predicates.*** *Let R be a rule in an ASM $\mathcal{M}$, φ be its guard, $g_1, \ldots, g_n$ be the outer guards of R, and C be a fault class. The set $\Gamma_C(R)$ of test predicates is given by the expressions $g_1 \wedge \ldots \wedge g_n \wedge (\varphi \oplus \varphi')$ with φ' in $F_C(\varphi)$.*

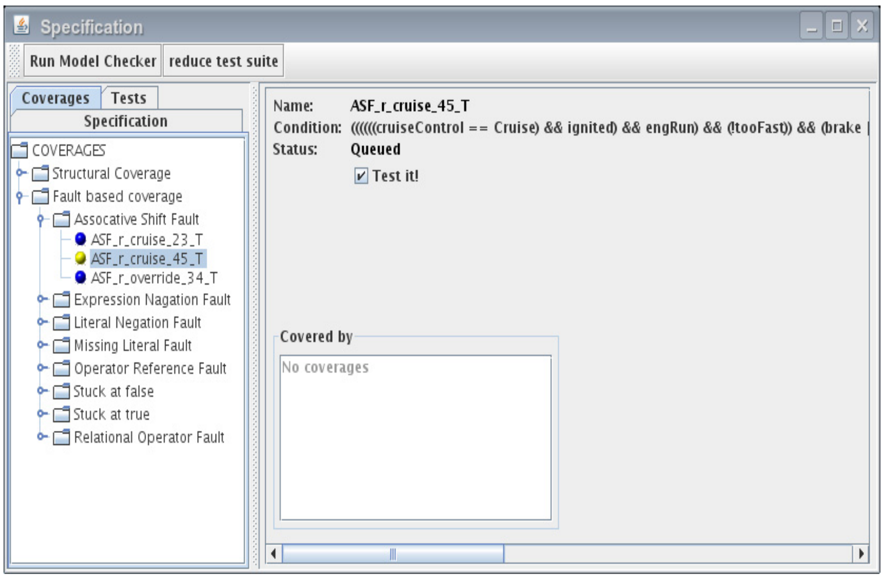
A test suite is adequate to test the guard φ of a rule R with respect to a fault class C if it covers every test predicate generated for R and C:

Definition 4. ***Fault Detecting Adequacy Criteria.*** *A test suite $\mathcal{T}$ is adequate with respect to the fault class C and the ASM $\mathcal{M}$, if for every rule R of $\mathcal{M}$*

and for every test predicate `tp` *in* $\Gamma_C(R)$ *there exists a state* s *in a test sequence of* $\mathcal{T}$ *such that the test predicate* `tp` *evaluates to true in* s*.*

5 Generation of Tests

To automatically generate the test sequences which cover a set of test predicates, we exploit the capability of the model checker Spin [23] to produce counter examples. Model checkers have been successfully applied to formal verification of properties, normally given in temporal logic, for systems modeled by means of automata. They automatically perform the proof of a desired property p by analyzing every possible system behavior, checking that p is true, and producing a counter example in case the property p does not hold in the model. The counter example is a possible system behavior that shows a case where the property p is falsified.



Name: ASF_r_cruise_45_T
Condition: ((((((cruiseControl == Cruise) && ignited) && engRun) && (!tooFast)) && (brake |
Status: Queued
[✓ Test it!]
No coverages

Fig. 2. An ATGT screen-shot.

The method presented in this section has been implemented in a prototype tool ATGT[3] - a screen-shot is reported in Fig. 2 - and consists in the following steps as illustrated in Figure 3.

- First, denoted by ①, a *Test Predicate Generator* computes the test predicate set $\Gamma_C = \{tp_i\}$ for the desired fault classes C introduced in Section 3 and for

[3] available at http://cs.unibg.it/gargantini/project/atgt/

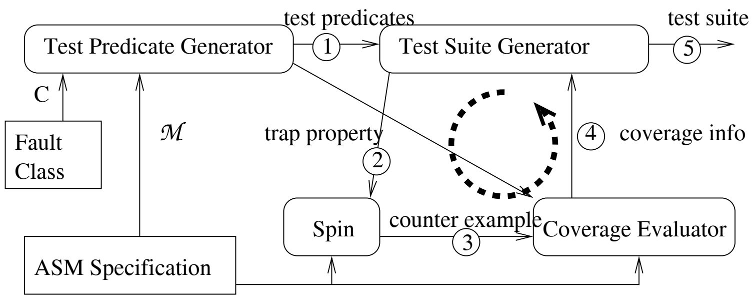


Fig. 3. Steps in the proposed generation method.

all the rule guards in the ASM specification under test given in the syntax of the AsmGofer [36].

– Second ②, the *Test Suite Generator* selects a test predicate tp_i, either randomly or according to the user request, and computes the *trap property* stating that tp_i is never true, i.e. $\mathtt{p} = \mathbf{never}(tp_i)$ which is translated in PROMELA, the language of Spin, as the statement `assert(!tp_i)`. The trap property `p` is not a desired property of the system; on the contrary, we look for a system behavior which falsifies `p`, i.e. where tp_i becomes true. This method has been introduced in [17,19].
– Third ③, the model checker is used to find the test sequence, by encoding the ASM specification in PROMELA, following the algorithms described in [19] and trying to prove the trap property `p`. If the model checker finds that `p` is false, i.e. a state where the tp_i is true, it stops and prints as counter example the state sequence leading to that state (plus the updates generated by the last update). This sequence represents the test that covers tp_i.
– Fourth ④, the *Coverage Evaluator* reads the counter example to produce the actual test sequence and to evaluate its coverage as explained in the following section against all the test predicates generated in the first step and provides the coverage information back to the Test Suite Generator.
– The process is iterated starting from the second step for each test predicate that has not been already covered. In the end, ⑤, a complete test suite is generated, except for the cases where the model checker fails to find a counter example as explained in Section 5.2.

5.1 Evaluating the Fault Coverage

A test sequence that is generated to cover a particular test predicate likely covers also many other test predicates, i.e. it contains a state where other test predicates are true and it can, therefore, discover other faults as well. Finding which predicates are covered can reduce the time and the resources to obtain a complete test suite - because we can decide to skip the generation of tests for test predicates already covered - and it can reduce the size of the test suite -

because we could decide to discard a test if the test predicates that it covers are covered also by other tests -.

To evaluate the coverage for a test, we give an unique identification (ID) to each test predicate and we add in the PROMELA file an instruction which prints a particular message if the test predicate is covered. For example, we introduce for the test predicate $(a \wedge b) \oplus a$ with identification tc_*ID* the following statement:

```
printf("_Covered: tc_ID %d \n",((a || b ) ^ a));
```

This instruction will print the ID for the test predicate followed by 1 or 0 whether the test predicate has been covered or not. The `printf` instruction is actually computed only during the last phase (④) and it does not complicate the model nor introduces new state variables since it is ignored during phase ③.

The proposed method for test evaluation can be used to evaluate any test sequence, regardless the way it has been generated. As we show in Section 6, we use this technique to get valuable insights over the fault detection capability of the structural coverage criteria presented in Section 2.3.

5.2 Undetectable Faults

When the model checker terminates, one of the following three situations occurs. The best case occurs when the model checker stops finding that the trap property is false, and, therefore, the counter example that covers the test predicate is generated.

The second case happens when the model checker checks every possible behavior without finding any state where the trap property is false, and, therefore, it actually proves the trap property **never**(tp_i). A test predicate, in our case, has always the pattern $A \wedge \varphi \oplus \varphi'$ (where A is a conjunction of outer guards), and **never**$(A \wedge \varphi \oplus \varphi')$ is equivalent to **always**$(A \rightarrow \neg(\varphi \oplus \varphi'))$, i.e. **always**$(A \rightarrow (\varphi \leftrightarrow \varphi'))$. Therefore, SPIN proves that when A holds, φ is always equivalent to its mutation φ' and that the fault does not introduce an actual change in the behavior of the system. In this case we say that such a fault is undetectable and we can safely ignore tp_i and simply warn the user that its model is insensitive in that rule guard to that fault.

In the third case, the model checker terminates because it finishes the maximum time or memory allocated for the search (set by the user or decided by model checker itself) but without completing the state space search and without finding a violation of the trap property, and, therefore, without producing any counter example (generally because of the state explosion problem). In this case, we do not know if either the trap property is true (i.e., the fault cannot be discovered) but too difficult to prove, or it is false but a counter example is too hard to find (i.e. the fault could be discovered if an appropriate test sequence could be found). When this case happens, our method simply warns the tester that the test predicate has not been covered, but it might be feasible.

Model Checking Limits. Model checking applies only to finite models. Therefore, our method works for ASM specifications having variables and functions with finite domains. The problem of abstracting models with finite domains from models with infinite domains such that some behaviors are preserved, is under investigation. Moreover, since model checkers perform an exhaustive state space (possibly symbolic) exploration, they fail when the state space becomes too big and intractable. This problem is known as *state explosion problem* and represents the major limitation in using model checkers. Note, however, that we use the model checker not as a prover of properties we expect to be true, but to find counter examples for trap properties we expect to be false. Therefore, our method does generally require a limited search in the state space and not an exhaustive state exploration. However, undetectable faults require a complete state search.

Model Checking Benefits. Besides its limits, model checking offers several benefits. For instance, SPIN adopts sophisticated techniques to compute and explore the state space, and to find property violations. It represents a state and the state space in a very efficient way using state enumeration, hashing techniques, and state compression methods. Moreover, SPIN explores the state space using practical heuristics and other techniques like partial order reduction methods and on-the-fly state exploration based on a nested depth first search. For these reasons, we have preferred existing model checkers instead of developing our own tools and algorithms for state space exploration. Moreover, the complete automaticity of model checkers allows to compute test sequences from ASM specifications without any human interaction.

6 Experiments

We report the result of applying our method to two case studies, the Cruise Control (CC) specification [31,5] and a simple model for a Safety Injection System (SIS) of a nuclear plant [13,18,17]. The CC has one monitored (i.e. modified only by the environment) enumerative variable, 4 monitored boolean variables and one controlled (i.e. modified only by the system) variable. It has 9 rules with rather complex boolean expressions as guards, which admit numerous boolean operator faults. The SIS includes three monitored variables (one integer in the interval [0,2000] and two switches), two internal variables (a boolean and an enumerative) and an output (boolean). It has 7 transition rules with guards which contain several relational operators and hence admit numerous ROFs. The number of test predicates is shown in Table 1. Note that 20 test predicates in the CC for the ROF were proved unfeasible by the model checker, which completed the search without finding any violation of the trap property, therefore actually proving that the faults are undetectable as explained in the second case of Section 5.2.

#tp	ENF	LNF	MLF	ASF	ORF	ST0	ST1	ROF	/unfeasible
for SIS	9	16	16	1	9	23	24	32	0
for CC	24	33	33	3	24	54	54	33	20

Table 1. Test Predicates and Tests for SIS and CC

strategy	#runs	time (sec)	#test	#states
1 - BFS, weak to strong	59	116	22	635
2 - BFS, strong to weak	59	102	22	637
3 - DFS, strong to weak	42	258	8	11760

Table 2. Runs for test generation

6.1 Generation of Tests

We have applied three strategies for test generation. In strategy 1 and 2 we use the breath first search (BFS) algorithm of Spin, which normally requires more time and memory than the default nested depth first search (nDFS) algorithm, but it guarantees that the shortest counter example is found. In strategy 3 we use the nDFS which is faster but finds long counter examples. Furthermore, in the first strategy we start from weaker fault classes and then we increase the fault detection capability of the tests by choosing stronger faults, while in the second and third strategy we start from strong coverage classes. Results are shown in Table 2, in which we report the number of runs, the total time required[4], the number of tests (some tests are discarded because they cover only test predicates covered by other test sequences in the test suite), and the total number of states in the test sequences.

Although several papers [30,31,27] suggest that hierarchical information about fault classes can be useful during test generation and that starting the test generation from the strongest coverage would require less time and fewer test cases than starting from the weakest coverage, we found no evidence of this fact. Indeed, strategy 2 (strong to weak) performed as well as strategy 1 (weak to strong). This result can be explained by considering that our method is iterative (it produces a test sequence at a time) and that we perform test evaluation at the end of every cycle. If the criterion S is stronger than the criterion W, any test set T_S adequate according to S includes any set of test T_W adequate according to W. The test generation starting from S produces a test suite T_S, whose evaluation stops the test generation because T_S covers W as well. The test generation starting from W initially produces T_W which still requires the generation of $T_S - T_W$ and not of the complete T_S. In both cases the number of test cases is the same (except for some non determinism in the generation and in the optimization of the test suites). However, our examples are too small to draw the definitive conclusion that hierarchical information about fault classes are useless during test generation.

[4] We have used a PC with an AMD Athlon 3400+ and 1 GB of RAM

	BR	MCDC	CR	UC
ENF	66	41	0	26
LNF	91	67	50	63
MLF	94	82	50	74
ASF	59	20	0	11
ORF	78	52	0	42
ST0	*100*	76	*100*	84
ST1	94	73	50	68
ST0U1	*100*	84	*100*	84
ROF	78	53	50	42

	#tp	ENF	LNF	MLF	ASF	ORF	ST0	ST1	ROF
BR	32	*100*	96	63	0	88	90	79	58
MCDC	98	*100*	*100*	*100*	75	*100*	*100*	*100*	65
CR	2	27	29	20	0	27	19	24	37
UC	19	*100*	96	67	25	88	91	81	58

↑(b) Fault detection capability of structural coverage

←(a) Structural coverage of fault criteria

Table 3. Structural vs fault coverage (in %)

Another unexpected result was that the test generation with the DFS algorithm performed worse than the others, although the DFS proved to be more efficient per visited state than the others: it explored around 11760 states (18 times more then the others) but it took only about twice as much time. By analyzing the runs, we found that the sole model checker execution (step ③ in Fig. 3) actually took less time than the same step in other strategies, but the other steps which analyze the results to evaluate the coverage took much more time, since the DFS produces very long counter examples. We believe that strategy 3 may perform better than the others for complex specifications, since in complex cases the model checker execution is the most critical step in the proposed test generation method. Moreover, strategy 3 is useful when one prefers very few test cases (for example if resetting the system is expensive) and because long test sequences may discover more faults (like extra states) [32,21].

6.2 Comparison with Structural Coverage Criteria

We have compared our new fault based adequacy criteria and the structural criteria presented in Section 2.3. Tests for structural criteria are generated following the technique introduced in [18,19]. Table 3 (a) reports the structural coverage of tests generated to cover faults. The ST0U1 has covered most structural parts in our specification, but not all. No fault based test set has been able to achieve the complete MCDC and Update Rule Coverage. Table 3 (b) reports the fault detection capability of tests generated by using the structural criteria. MCDC performed better then the others, but no structural coverage has been able to achieve the ASF and ROF criteria. These data suggest that fault based criteria and structural criteria are complementary to each other.

6.3 Cross Coverage Among Fault Classes

We have also analyzed the *cross coverage* among the fault based criteria and results are reported in Table 4, which must be read as follows. The tests generated for a fault class in a row (cross) covers also the shown percentage of the test

	ENF	LNF	MLF	ASF	ORF	ST0	ST1	ROF
ENF	-	78	41	*0*	85	53	63	32
LNF	*100*	-	73	25	*100*	79	94	63
MLF	*100*	*100*	-	75	*100*	94	95	63
ASF	82	33	31	-	67	10	53	34
ORF	*100*	90	57	*0*	-	65	72	49
ST0	*100*	*100*	84	25	*100*	-	87	62
ST1	*100*	*100*	82	50	*100*	82	-	63
ST0U1	*100*	*100*	*100*	75	*100*	-	-	65
ROF	*100*	96	59	25	97	64	81	20

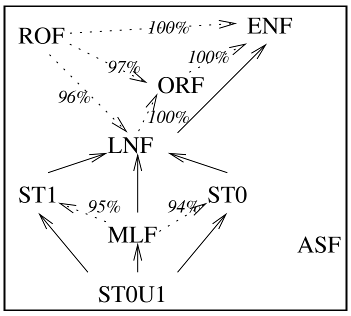


Table 4. Cross Coverage (%)

predicates for other fault classes displayed in the columns. Besides the confirmation of the theory (continuous arrows in the figure of Table 4), we have found some empirical relationships among fault classes (dotted arrows in the figure). For instance, ROF covered all the ENFs, ORF covered all the ENFs, and LNF covered all ORFs. MLF seems stronger than ST1 and ST0 individually, and ROF seems stronger than LNF and ORF. Although this empirical extended hierarchy may not hold in general, we believe that for most boolean expressions these relationships are likely to be true. This information may be useful in practice if one has a test suite that targets a specific fault and want to approximately judge the test suite's fault detection capability. We found that ASF is really complementary with respect all other criteria.

7 Related Work

Many papers tackle the problem of tests generation or selection. The subject of using model checkers for test generation starting from models has been studied for many years. For a (not so recent) survey see [2]. In [15] the authors used Spin to generate test sequences for a protocol augmented by a test predicate, called test purpose, written by the designer by hand. Classical control oriented tests generation is presented for SCR in [17] and for ASMs in [18]. Several recent papers apply the same concepts to UML state diagrams [29], to StateCharts [25], and to Stateflow [20] specifications. [35] presents state coverage, decision coverage and MCDC (not masked) for specifications written in RSML^{-e}. They all share the same approach. They introduce some control oriented coverage, derive the test predicate from decision points in the model and then use the model checker to obtain the test sequences.

A first attempt to introduce data flow oriented coverage criteria can be found in [18] where the rule update criterion (presented in Section 2.3) covers the real update of a variable. A novel approach is presented in [24], which shows how the classical data flow coverage criteria can be translated in terms of the Computation Tree Logic (CTL).

The combined use of model checking and mutation testing is presented in [3,8]. Their approach, that we could classify as fault oriented [39], is very similar to ours, but the technique is completely different. Differently from us, they do not use test predicates derived from the specifications by using the boolean difference. Instead, they directly apply mutation techniques to models. The original specification, written for the model checker SMV, is initially augmented by many temporal logic properties (*constraints*) that represent the correct behavior. In the extraction of these properties (also called *expounding*) there are several "subtle issues that require attention [4]" and may reduce the fault detection capability of the tests. Afterwards, the specification or the constraints are repeatedly modified applying mutation operators (more general than our fault classes), that introduce faults in the models or in the constraints. Counter examples are automatically generated by SMV either (*approach 1*) trying to prove the original properties in faulty models to obtain wrong behaviors that implementations must not exhibit or (*approach 2*) proving mutated properties for the correct model to obtain tests sequences that discover particular faults (or *kill* mutants).

We can compare their approach 2 with ours as follows. In the extraction of constraints, they build a set of safety properties which are always true in the original model. Given a safety property $\textbf{always}(P)$, they look for a counter example by trying to prove $\textbf{always}(P')$ where P' is a possible mutation of P. If a counter example is found, they have found a state where the mutated property is false, i.e. $\neg P'$. They actually find a state where P is true (safety property) while P' is false, i.e. $P \wedge \neg P'$, which is a particular case of $P \oplus P'$, the boolean difference of P. Our approach does not require the extraction of safety properties, since test conditions are defined as boolean differences over guards, which are not always true.

Ammann et alt. tackle also the problem of evaluation of test sequences against specification-based coverage criteria [4]. They show how the model checker SMV can be used to evaluate a test sequence with respect to the capability to discover (or *kill*) mutations of the original specification. The test sequence (regardless the way it has been generated) is transformed in a SMV model to run together with the mutated specification. This requires a run for every test and every mutation, rising the problem how to reduce (*winnow*) the number of mutations really necessary to evaluate the coverage of a test. Tests which kill a subset of mutations of other tests, can be discarded. In our approach, we can evaluate the capability of a test sequence to detect all faults in one run by using test conditions. Furthermore we are able not only to discard duplicated test cases, but also to avoid the generation of tests for test predicates already covered.

Model checkers can be used to generate tests in program based testing too: the model checker BLAST is used in [7] to generate test suites and to detect dead code in C programs.

8 Conclusions and future work

Although we have shown how to generate tests to detect several fault classes, we plan to introduce other fault classes, possibly involving not only boolean expressions but also integers (like off-by 1 fault or at the boundaries faults). Moreover, while this paper focuses on faults in the rule guards, we plan to define other fault classes involving the rule updates. Our method has been applied to the generation and evaluation of tests for several case studies, but more experiments with real specifications are needed to assess its real applicability. Abstract State Machines are chosen as formal method, but our approach can be easily adapted to any formalism based on guarded state transitions. We have discovered that the hierarchy among faults is useless in the prioritization during test generation, but further experiments and theoretical research is needed to definitely prove that.

References

1. S. B. Akers. On a theory of boolean functions. *Journal Society Industrial Applied Mathematics*, 7(4):487–498, December 1959.
2. P. Ammann, P. E. Black, and W. Ding. Model checkers in software testing. Technical Report NIST-IR 6777, National Institute of Standards and Technology, 2002.
3. P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, page 46, Brisbane, Australia, Dec. 1998.
4. P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):275–300, December 2001.
5. J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 280–292, New York, NY, USA, 1996. ACM Press.
6. M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, 2001.
7. D. Beyer, A. J. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. International Conference on Software Engineering (ICSE), Edinburgh*, pages 326–335. IEEE CS Press, May 2004.
8. P. E. Black, V. Okun, and Y. Yesha. Mutation of model checker specifications for test generation and evaluation. In W. E. Wong, editor, *Mutation Testing for the New Century, proc. of Mutation 2000*, pages 14–20. Kluwer Academic Publishers, October 2000.
9. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.
10. J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering – ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 285–302. Springer-Verlag, Nov. 1999.
11. T. Y. Chen and M. F. Lau. Test case selection strategies based on boolean specifications. *Softw. Test., Verif. Reliab.*, 11(3):165–180, 2001.

12. J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (mcdc). Technical report, Boeing, Seattle WA, 1997.
13. P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD, 1993.
14. R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Testing, Analysis, and Verification*, pages 142–151. IEEE Computer Society Press, 1988.
15. A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*, number 1217 in Lecture Notes in Computer Science, pages 384–398. springer, 1997.
16. P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, Mar. 1993.
17. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, volume 1687 of *LNCS*, pages 6–10, Sep 1999.
18. A. Gargantini and E. Riccobene. ASM-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7(11):1050–1067, Nov. 2001.
19. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from ASM specifications. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines, 10th International Workshop, ASM 2003*, pages 263–277, March 3-7 2003.
20. G. Hamon, L. M. de Moura, and J. M. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 261–270, 2004.
21. M. P. Heimdahl and D. George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Automated Software Engineering*, Linz, Austria, September 2004.
22. R. M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Trans. Softw. Eng. Methodol.*, 11(4):427–448, 2002.
23. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
24. H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03, Portland, Oregon, May 3-10*, 2003.
25. H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. Automatic test generation from statecharts using model checking. In *Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, August 2001. BRICS Notes Series, NS-01-4, pp. 15-30.*, 2001.
26. IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* Institute of Electrical and Electronics Engineers, 610.12.
27. K. Kapoor and J. P. Bowen. Ordering mutants to minimise test effort in mutation testing. In *Formal Approaches to Software Testing, 4th International Workshop, FATES*, pages 195–209, 2004.

28. K. Kapoor and J. P. Bowen. A formal analysis of MCDC and RCDC test criteria. *Softw. Test. Verif. Reliab.*, 15(1):21–40, 2005.
29. Y. Kim, H. Hong, S. Cho, D. Bae, and S. Cha. Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187–192, Aug 1999.
30. D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, Oct. 1999.
31. M. F. Lau and Y.-T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Trans. Softw. Eng. Methodol*, 14(3):247–276, 2005.
32. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of The IEEE*, pages 1090–1123, Aug. 1996. Published as Proceedings of The IEEE, volume 84, number 8.
33. V. Okun, P. E. Black, and Y. Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 46:525–533, 2004.
34. A. Pretschner. Model-based testing in practice. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 537–541. Springer, 2005.
35. S. Rayadurgam and M. P. Heimdahl. Generating MC/DC adequate test sequences through model checking. In *28th Annual NASA Goddard Software Engineering Workshop (SEW 03)*, 2003.
36. J. Schimd. Executing ASM specifications with AsmGofer. http://www.tydo.de/AsmGofer.
37. T. Tsuchiya and T. Kikuno. On fault classes and error detection capability of specification-based testing. *ACM Trans. Softw. Eng. Methodol.*, 11(1):58–62, 2002.
38. E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.
39. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.