Journal of Universal Computer Science, vol. 7, no. 11 (2001), 1050-1067
submitted: 7/6/01, accepted: 1/10/01, appeared: 28/11/01 © Springer Pub. Co.

ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation

Angelo Gargantini
(C.E.A.- Università di Catania
Piazza Università, 2 - 95124 Catania, Italy
a.gargantini@unict.it)

Elvinia Riccobene
(Dipartimento di Matematica e Informatica - Università di Catania
V.le A. Doria, 6 - 95125 Catania, Italy
riccobene@dmi.unict.it)

Abstract: This paper tackles some aspects concerning the exploitation of Abstract State Machines (ASMs) for testing purposes. We define for ASM specifications a set of adequacy criteria measuring the coverage achieved by a test suite, and determining whether sufficient testing has been performed. We introduce a method to automatically generate from ASM specifications test sequences which accomplish a desired coverage. This method exploits the counter example generation of the model checker SMV. We use ASMs as test oracles to predict the expected outputs of units under test.

Key Words: testing, test sequence generation, formal methods, Abstract State Machines

Category: D.2.2, D.2.5

1 Introduction

The final goal of the software development process is to produce high-quality *software*, i.e. software that satisfies the given requirements and that meets the user needs. To uncover development and coding errors, to asses its reliability and dependability, and to convince customers that the performance is acceptable, the software needs to be tested. However, software testing is extremely costly and time-consuming. It has been estimated that current testing methods consume between 40% and 70% of the software development effort [Beizer 1983]. Formal methods offer an opportunity to significantly reduce the testing costs, and recent work [Stocks and Carrington 1996, Hierons and Derrick 2000] addresses the relationship between testing and formal specifications.

In *specification-based testing*, a specification plays several roles. First, it can be used as *test oracle* [Podgurski and Clarke 1990, Richardson et al. 1992], i.e. as a means to predict the expected outputs of the system. When testing a software the correctness of its behavior should not be left to the judgment of a human, because for complex and critical software the human assessment is neither efficient nor dependable. On the contrary, the specification, as authoritative font of the expected behavior, can assess correctness of implementations.

Second, *test adequacy criteria* can be derived from a specification [Zhu et al. 1997]. They determine if a test set, also called *test suite*, is adequate to test a software system, whether enough testing has been performed or further tests are needed. For example, in the classical structural testing, the statement coverage criterion establishes that a test is adequate for a program if every statement of that program has been exercised. In the context of specification-based testing, a test adequacy criterion is formally defined as a function that takes a specification S and a test suite T and returns *true* if T is adequate, *false* otherwise [Frankl and Weyuker 1993].

A specification can also provide *selection criteria* to choose, among all possible system behaviors, an adequate test suite. Normally a selection criterion introduces some algorithms or techniques to actually generate test sequences from formal specifications [Blackburn et al. 1997, Gargantini and Heitmeyer 1999, Weyuker et al. 1994].

In this paper, we focus on *ASM-based testing*, and tackle several issues in this topic. In particular, we attempt to solve the challenging problem concerning the exploitation of "ASMs for defining and implementing methods for generating test suites from high-level specifications" [Börger 2000].

[Section 2] briefly overviews the ASM notation, while [Section 3] presents the Safety Injection System [P.J.Coutois and D.L.Parnas 1993] (SIS in brief) that we choose as case study to experiment our approach. [Section 4] provides formal definitions of test sequence and test suite, and discusses the use of ASM specification as test oracle.

We define in [Section 5] several coverage criteria for ASM specifications. We provide for each coverage criterion a set of formulas, called *test predicates*, which determine the set of states that realizes the coverage. These coverage criteria can be used as adequacy criteria to measure the degree of coverage achieved by a test set.

A method to automatically generate test sequences from ASM specification is presented in [Section 6]. This method exploits the counter example generation of the model checker SMV to build tests for a given coverage criterion. The general architecture of the tool developed for automatic test sequence generation is described in [Section 7], while [Section 8] reports our experimental results in applying our method to the SIS case study. Related work and future directions are given in [Section 9] and in [Section 10], respectively.

2 Abstract State Machines

Abstract State Machines (ASMs) represent a semantically well-defined, precise form of widely used pseudo-code over abstract structures. We provide in the rest of this section some intuitive explanations which should suffice to correctly understand and use ASMs for the purposes of this paper. We refer the reader to [Gurevich 2000] for a detailed mathematical definition.

The *states* of ASMs are arbitrary structures in the standard sense they are used in mathematical sciences, i.e. domains of objects with functions and predicates defined on them. The basic operations of ASMs are guarded destructive assignments of values to functions at given arguments, expressed in the following form, called *guarded transition rule*:

If *Cond* **then** *Updates*

where *Cond* is an arbitrary condition (Boolean expression) formulated in the given signature, *Updates* consists of finitely many *function updates*:

$$f(t_1, \ldots, t_n) := t$$

which are executed simultaneously. The terms $t_1, \ldots, t_n$ are arguments at which the value of the arbitrary function f is set to t. For technical convenience we treat predicates as Boolean-valued functions.

An ASM M is a finite set of rules for such guarded multiple function updates. The computation of an ASM is defined in the standard way transition system runs are defined. Applying one step of M to a state $\mathcal{A}$ produces as next state another state $\mathcal{A}'$, of the same signature, obtained as follows: First evaluate in $\mathcal{A}$, using the standard interpretation of classical logic, all the guards of all the rules of M. Then compute in $\mathcal{A}$, for each of the rules of M whose guard evaluates to true, all the arguments and all the values appearing in the updates of this rule. Finally replace, simultaneously for each rule and for all the locations in question, the previous $\mathcal{A}$-function value by the newly computed value if no two required updates contradict each other.

The state $\mathcal{A}'$ thus obtained differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by a rule of M which could fire in $\mathcal{A}$. The effect of an ASM M, started in an arbitrary state $\mathcal{A}$, is to repeatedly apply one step of M as long as an M-rule can fire. Such a machine terminates only if no rule is applicable any more (and if the monitored functions do not change in the state where the guards of all the M-rules are false).

ASM functions are distinguished in *basic* functions and *derived* functions (which are defined in terms of basic ones). Within derived or basic functions, *static* functions, which remain constant during computations, are distinguished from *dynamic* ones, which may change from state to state. Among the dynamic functions we distinguish the *controlled* ones from the *monitored* ones, also called *in*-functions or *inputs*. The controlled functions are subject to change by rule updates. The monitored functions can change only due to the environment or, more generally, due to actions of other agents. For a complete function classification see [Börger et al. 2000].

We distinguish for testing purposes controlled dynamic functions in *outputs* corresponding to observable controlled functions, and *internal functions* which are not observable outside the system.

3 The Safety Injection System Case Study

The system here used as case study is a simplified version of a control system for a safety injection [P.J.Coutois and D.L.Parnas 1993]. It monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold. The system operator may block this process by pressing a "Block" switch. The system is reset by a "Reset" switch. To specify the requirements of the control system, we use the monitored variables `WaterPressure`, `Block` and `Reset`, and the output controlled variable `SafetyInjection` to denote the controlled quantity which can be `on`, `off`. To specify the modes the system can be according to the `WaterPressure` values (if such values are permitted or the water pressure is risking to reach threshold values), we introduce the internal controlled variable Pressure, also called *mode*, which can take values TooLow, Normal, High. At any given time, the system must be in one and only one of these modes. A drop in water pressure below a constant Low causes the system to enter mode TooLow; an increase in water pressure above a larger constant Permit causes the system to enter mode High. Otherwise, Pressure is Normal. The boolean constant Overridden (internal controlled term) is *true* if safety injection is blocked, *false* otherwise. The controller policy is simple: SafetyInjection is on when Pressure is TooLow and Overridden is *false*; it is off otherwise.

In our initialization, the constants Low and Permit are assigned the values 900 and 1000. When the plan and the system start their activity, WaterPressure is 14 and the mode Pressure is TooLow, but Overridden is false and therefore SafetyInjection is on.

The initial state of this critical system is fundamental and fixed by the costumer, as well it is crucial that test sequences start from an initial state, otherwise they would be useless to the testing activity.

The complete ASM specification of the Safety Injection System is reported in [Appendix A].

4 Test Sequences and Test Suites

We provide in this section some basic terminology. Adapting to our purposes some definitions common in literature for state transition systems [Ammann et al. 1998, Lee and Yannakakis 1996], we define a *test sequence* or *test* as follows.

Definition 1. A **test sequence** or **test** 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 transition rules.

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 advocated in [Richardson et al. 1992] because 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.

Other approaches [Blackburn and Busser 1996] consider a test as a simple pair of two consecutive states (often called *test vector*). The generation of state vectors is easier than the generation of test sequences, but test vectors are less useful. A test vector does not give the user the complete scenario of system execution, and does not provide the tester with any information about the reachability of the pair of states from a valid initial state – a pair could be not reachable at all, and even if it is reachable, the input sequence necessary to lead the system to the first state of the pair is not provided –. Note that obtaining test vectors from a test sequence is straightforward.

When the tester has access to the internals of the unit under test (UUT), inputs, outputs, and internal functions provided by test sequences are used; when the tester can observe only UUT inputs and outputs, values of internal functions are ignored. For these reasons, our definition of test makes our approach as general as possible, widely applicable, and neutral with respect to the actual use of the generated test sequences.

We define a collection of test sequences as follows.

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

5 Coverage Criteria for ASMs

There exist several aspects of an ASM model that the designer would like to cover with a test suite: (a) how the execution of *transition rules* affects machine states; (b) how *function values* may influence the system behavior (*sub-domain partitioning testing* [Myers 1979]); (c) under which circumstances *conditions and decisions* are true or false; (d) how *non deterministic specification features* influence the system behavior.

To test any of these aspects, suitable coverage criteria have to be defined. Since most of the testable aspects are related to the syntax and semantics of transition rules, as first step we focus on rules testing.

In the sequel we use the following terminology: let $\{r_i\}$ be a set of consistent ASM rules, g_i be the r_i's guard, and $Ups_i{=}f_{i1}(\bar{t}_{i1}) := t_{i1}, \ldots, f_{in}(\bar{t}_{in}) := t_{in}$ be the r_i's function updates. $Val(t, S)$ yields the interpretation of t in the state S.

For the moment, we do not use constructors `extend` and `choose` inside rules.

5.1 Test Predicates

We formally define a coverage using a set of logical predicates, called *test predicates*. A test predicate is a formula over the state and determines if a particular testing goal is reached (e.g. a particular condition or a particular event is covered). A test suite T satisfies a coverage C if each test predicate of C is true in at least one state of a test sequence of T.

The generation of test sequences that covers test predicates is an undecidable problem [Weyuker 1979].

5.2 Rule Coverage

Definition 3. A test suite satisfies the *rule coverage* if for every rule r_i there exists at least one test sequence for which r_i fires at least once (i.e. there exists a state where the guard g_i is true) and there exists at least one test sequence for which r_i does not fire at least once (i.e. there exists a state where the guard g_i is false).

According to this definition, to test each rule r_i we search for a test sequence containing at least one state where the guard g_i is true, and a test sequence with a state where the guard g_i is false.

Test Predicates

The set of test predicates for the rule coverage criterion is simply the set of the rule guards and their negations, namely $\{g_i\} \cup \{\neg g_i\}$.
Therefore, to achieve rule coverage we require that the test suite satisfies the following property: $\forall i\,(\exists S(\mathit{Val}(g_i, S) = \mathit{true}) \wedge \exists S(\mathit{Val}(g_i, S) = \mathit{false}))$, being S a state in some test sequence.

Example

Given the rule:

```
R1) if WaterPressure >= Low & Pressure = TooLow
    then Pressure := Normal
```

a test suite satisfies the *rule coverage* for `R1` if it contains a test sequence $\{S_0, S_1, ..\,.., S_k\}$ such that Val(`WaterPressure >= Low & Pressure = TooLow`, $S_i) = \mathit{true}$ for some $i = 0, \ldots, k$, and a test sequence $\{S'_0, S'_1, \ldots, S'_w\}$ such that Val(`WaterPressure >= Low & Pressure = TooLow`, $S'_i) = \mathit{false}$ for some $i = 0, \ldots, w$.

The number of test sequences to get the rule coverage for a system of n rules is at most $2n$. This figure might be considerably lower because a single test sequence could contemporary cover several rules.

Every rule models a particular behavior or decision of the specified software. In testing, we like to recreate that behavior or that decision determining, if possible, when the decision is taken. Moreover, we want to check the system behavior when the decision is not taken (*else cases* discussed in [Gargantini and Heitmeyer 1999]). Since each rule guard models the event or the state condition under which the decision is taken or the behavior is exposed, we search for a state where the guard is true (hence the rule fires and the decision is taken), as well for a state where the guard is false (hence the rule does not fire and the decision is not taken).

We believe that testers should at minimum provide tests where every rule fires, and tests where each rule does not fire at least once. Even if this criterion guarantees only a minimal coverage, it is usually easy to achieve (manually or automatically). It can be considered a "minimal standard for testing commercial software", as advocated in [Beizer 1983] and, more recently, also in [Miller 2001]: "No one should sell software that hasn't met *at least* this requirement. Most software developers will exceed this minimal standard, but no professional software engineer should allow less".

5.3 Rule Update Coverage

If a guard models a particular event or state condition, rule updates represent the reaction of the system to particular events or conditions. Therefore, a good test set should test function updates $f_{ij}(\bar{t}_{ij}) := t_{ij}$ of each rule r_i. To this purpose, for each update, we look for a test sequence containing a state where the update is performed and it is not *trivial*, i.e. location content does change [Gurevich 2000]. Thus we introduce the following new coverage criterion.

Definition 4. A test suite satisfies the *rule update coverage* if for every rule r_i and for every j-th function update $f_{ij}(\bar{t}_{ij}) := t_{ij}$ of r_i there exists at least one test sequence for which r_i fires at least once and the the j-th update is not trivial.

Test Predicates

According to this definition, the test predicate to test the j-th update of the rule r_i, is $g_i \wedge f_{ij}(\bar{t}_{ij}) \neq t_{ij}$. We search, therefore, for a test sequence containing a state S where $Val(g_i, S) = true$ and $Val(f_{ij}(\bar{t}_{ij}), S) \neq Val(t_{ij}, S)$.

Example

Given the rule:

```
R2) if WaterPressure >= Permit & Pressure = Normal
    then Pressure := High
         Overridden := false
```

a test suite satisfies the *rule update coverage* for `R2` and the update `Pressure := High` if it contains a test sequence $\{S_0, S_1, \ldots, S_k\}$ such that *Val*(`WaterPressure >= Permit & Pressure = Normal`, S_i) $= true$ for some $i = 0, \ldots, k$, and *Val*(`Pressure`, S_i) $\neq$ `High`.

The number of test sequences to achieve the rule update coverage for a rule system of altogether m updates, is at most m. As for the previous coverage criterion, the number of test sequences really necessary to satisfy the update coverage might be considerably lower, since a test sequence might cover several updates (for example all the updates of the same rule).

5.4 Advanced Coverage Criteria

Rule coverage and update coverage are essential to achieve minimum confidence in testing adequacy; however, they test a system only weakly. In order to gain a better coverage, other criteria can be defined. To this aim we follow two different directions: one exploring the parallel rule application and the detection of inconsistent updates, and the other analyzing conditions inside rule guards.

The definition of other more powerful coverage criteria is under investigation (see [Section 10]).

5.4.1 Parallel Rule Coverage

This coverage tests the interaction between rules and can help to discover inconsistent updates.

Let $R = \{r_i\}_{i=1,\ldots,m}$ be a set of ASM rules, and $n \leq m$.

Definition 5. A n-tuple T_n of rules is *unfirable* if all the n rules of T_n can never simultaneously fire.

Definition 6. A test suite satisfies the *n-parallel rule coverage* for R if for every n-tuple T_n of rules of R, it holds

(i) either T_n is unfirable;

(ii) or there exists a test sequence containing a state for which all the n rules of T_n simultaneously fire.

Test Predicates

For each tuple T_n, the test predicate P_n is the conjunction of the n guards of the rules in T_n. A test sequence covers the tuple T_n if it contains a state S such that $Val(P_n, S) = true$.

Example

Given $n = 2$ and the tuple $T_2 = \{R2, R5\}$ with rules:

```
R2) if WaterPressure >= Permit & Pressure = Normal
    then Pressure := High
         Overridden := false
R5) if Reset = on & (Pressure = TooLow or Pressure = Normal)
    then Overridden := false
```

a test sequence $\{S_0, S_1, \ldots, S_q\}$ covers T_2 if it contains a state S_i, for some $i = 0, \ldots, q$, such that $Val($`WaterPressure >= Permit & Pressure = Normal`$, S_i)$ $= true$ and $Val($`Reset=on & (Pressure=TooLow or Pressure=Normal)`$, S_i) = true$.

Adequacy Measure

The number of the n-tuples T_n of rules of R, and then of the test predicates, is $\binom{m}{n}$. This number might be high, and therefore getting the complete n-parallel coverage could be very hard. However, several n-tuples could be unfirable, thus considerably reducing the number of test sequences required to satisfy the criterion. Consider, for example, the following two rules:

```
R2) if WaterPressure >= Permit & Pressure = Normal then ...
R1) if WaterPressure >= Low & Pressure = TooLow then ...
```

Finding a state where both guards are true is impossible. Note that the tool presented in [Section 7] is able to detect such infeasible cases.

Nevertheless, the number of firable tuples might be still high, and in this case even a partial coverage is valuable. This suggests to extend – as proposed in [Zhu et al. 1997]– the definition of test adequacy criterion (given in [Section 1]) as a function, called *test adequacy measure*, that takes a specification and a test set and returns a real number between 0 and 1 (from 0 if the test set is not adequate to 1 if the test set is totally adequate). For the n-parallel coverage criterion, the test adequacy measure of a test suite T is

$$\frac{num_{tc} + num_{ut}}{\binom{m}{n}}$$

where num_{tc} is the number of n-tuples covered by at least a test sequence in T, and num_{ut} is the number of unfirable n-tuples.

Inconsistent Update Detection

The parallel rule coverage allows to discover inconsistent updates. For example, the following two rules (slightly modified version of two SIS rules) would inconsistently update the location `Overridden`.

```
Ra) if Reset = on & Pressure = TooLow
    then Overridden = false
Rb) if Block = on & Pressure = TooLow
    then Overridden = true
```

A test sequence generated to cover the 2-tuple $T_2 = \{Ra, Rb\}$ exposes such inconsistency.

5.4.2 Strong Parallel Rule Coverage

This coverage is a natural extension of the parallel rule coverage.

Definition 7. A test suite T satisfies the *strong n-parallel rule coverage* for R if for all k, $1 \leq k \leq n$, T satisfies the k-parallel rule coverage.

Test predicates and adequacy measure definitions are straightforward.

5.4.3 Modified Condition Decision Coverage

Since often rule guards are complex logical expressions composed of several atomic literals and conditions (that cannot be split into simpler boolean conditions), in order to test the impact of every atomic condition on the guard evaluation, we introduce another coverage criterion adapting the *Modified Condition Decision Coverage* [Chilenski and Miller 1994].

Definition 8. A test suite satisfies the *modified condition decision coverage* if for every rule r_i and for every atomic condition C_j in the – possibly nested – r_i's guard, C_j has taken on all possible outcomes at least once (i.e. there exists one test sequence with C_j true in at least one state of a test sequence, and one test sequence with C_j false in at least one state of a test sequence), and each atomic condition has been shown to independently affect the guard outcome by varying just that condition while holding fixed all other possible atomic conditions.

According to this definition, each atomic condition inside a guard must be true at least in one state of a test sequence, and false at least in one other state of a test sequence of the test suite. Moreover, it must be demonstrated that the guard outcome changes as a result of changing this condition. Therefore, to test each atomic condition c, we search (a) for a test sequence containing a state S_1 where c is true and the guard is true (false); (b) for a test sequence containing a state S_2 where c is false and the guard is false (true); (c) the other atomic conditions have the same truth value in S_1 and in S_2.

Test Predicates

The computation of test predicates for MCDC is complex. There exist several standard algorithms to compute them. For example, [Chilenski and Miller 1994] suggests to use pairs tables, [Offutt and Liu 1999] to use trees, [Kuhn 1999] to use boolean derivatives.

Example

Given the rule:

```
R5) if Reset = on & (Pressure = TooLow or Pressure = Normal)
    then Overridden := false
```

every atomic condition inside the guard is tested by means of six test predicates as shown in the following table.

Test Pred. number	`Reset=On`	`Pressure=TooLow`	`Pressure=Normal`	Guard Outcome
1	*F*	$Pressure = TooLow \vee Pressure = Normal$		**F**
2	*T*	$Pressure = TooLow \vee Pressure = Normal$		**T**
3	T	*F*	F	**F**
4	T	*T*	F	**T**
5	T	F	*F*	**F**
6	T	F	*T*	**T**

The modified condition decision coverage has a strong reputation and is suggested by the FAA (Federal Aviation Agency) for certification of civil avionics software. A complete discussion about MCDC and its advantages can be found in [Chilenski and Miller 1994].

To achieve MCDC of a formula with n atomic conditions, $2n$ test predicates are needed. However, only $n + 1$ test predicates are often enough to get the coverage thanks to the repetition of the condition truth values (see for example the condition values at rows 3 and 5 in the table above). In any case, the number of test sequences necessary to get the modified condition decision coverage for a rule containing n atomic conditions is always much lower than the number of tests necessary for the *multiple condition coverage* [Myers 1979] which requires to consider all the possible 2^n combinations of the atomic condition truth values.

6 Automatic Test Sequence Generation Using Model Checking

This section introduces a method to automatically generate test sequences from ASM specifications. Test sequences are generated to achieve a desired coverage among those presented in [Section 5]. The method can be easily extended for new coverage criteria.

The method uses SMV [McMillan 1992] as subroutine and exploits SMV's counter example generation capability. SMV is a tool for checking finite state systems against specifications in the branching time temporal logic CTL (Camputation Tree Logic). A system is described in SMV as a set of initial states and a set of transition relations.

CTL formula are assertions, generally putative invariants of the system, about a single state or about paths from a state. Among the CTL temporal operators, `AG` p means *along all paths, all states satisfy p*, and `EF` p means *along some path, some states satisfy p*. In comparison with classical temporal operators, `AG` is like *henceforth* ($\Box$), while `EF` is like *eventually* ($\Diamond$).

SMV uses the OBDD-based symbolic model checking algorithm to efficiently checking a CTL formula against the system specification. If SMV analyzes all reachable states and detects no violations, then the property holds. Otherwise, if the model checker finds a reachable state that violates the property, it returns a "counterexample", namely a sequence of reachable states beginning in a valid initial state and ending with that violating the property.

Our method exploits the counter example generation feature of SMV. This particular use of a model checker has been already introduced in other approaches [Engels et al. 1997], [Ammann et al. 1998], [Gargantini and Heitmeyer 1999]. Instead of using the model checker to prove actual system properties, we check the validity of *ad-hoc* properties, called *trap properties,* whose sole scope is leading the model checker to the generation of desired counter examples.

The method consists in several steps. First, we compute for a desired coverage the test predicates set $\{tp_i\}$. Second, we encode the ASM specification in SMV following the technique described in [Del Castillo and Winter 1999]. Third, we compute for each test predicate tp_i the test sequence that covers it by running SMV on the trap property `AG(!`tp_i`)`(or equivalently `!EF(`tp_i`))` stating that tp_i is never true. If SMV finds a state where tp_i is true, it stops and prints as counter example the state sequence leading to that state. This sequence is the test that covers tp_i.

We show here how to generate a test sequence for the rule coverage of rule:

```
R1) if WaterPressure >= Low & Pressure = TooLow
    then Pressure := Normal
```

The test predicates are the `R1`'s guard and its negation. For the first one, the trap property is: `AG(!(WaterPressure >= Low & Pressure = TooLow))`. Running SMV, we get the desired test sequence:

```
-- specification AG (!(WaterPressure >= Low & Pressure = ... is false
-- as demonstrated by the following execution sequence
state 1.1:
Permit = 1000
Low = 900
Block = off
Reset = on
WaterPressure = 14
Pressure = TooLow
Overridden = 0
SafetyInjection = on

state 1.2:
Block = on
Reset = off
WaterPressure = 24
...
... WaterPressure increases
...
state 1.89:
WaterPressure = 902
Overridden = 0

resources used: ...
```

The counter example ends with a state where the guard of `R1` is true (and `R1` fires). The test sequence for the `R1`'s guard negation is generated running SMV on the trap property `AG(WaterPressure >= Low & Pressure = TooLow)`.

Note that due to the limits of model checking, this approach is applicable to specifications with finite domains, unless abstraction techniques are exploited.

6.1 Infeasible Tests

We call a test predicate *not coverable* or *infeasible* if either it is logically equivalent to false (for example $WaterPressure > 0 \wedge WaterPressure < 0$), or it is true only in an unreachable state.

If a test predicate tp_i is infeasible, the trap property `AG(!`tp_i`)` holds and the model checker, if it terminates, proves it and warns the tester that the test predicate is not coverable. The model checker, however, might not terminate and not produce any counter example, generally because of the state explosion problem. When the model checker does not end, the user does not know if either the trap property is true (i.e. the test is infeasible) but too difficult to prove, or the trap property is false and there exists a counter example but it is too hard to find. When this happens, our method simply alerts the tester that a test predicate has not been covered, but it might be feasible. The use of abstraction to reduce the likelihood of such cases is under investigation. The possible failure of our method does not surprise: the problem of finding a test that covers a particular predicate is undecidable [Weyuker 1979]. Nevertheless, in our experience this case is quite rare: for our case study it never happened.

7 Tool Architecture

Figure 1 shows the architecture of the tool we developed in order to automatically generate test suites for ASM specification. The `TEST PREDICATES GENERATOR` takes in input the ASM model and provides in output the test predicates for all the coverage criteria defined in [Section 5].

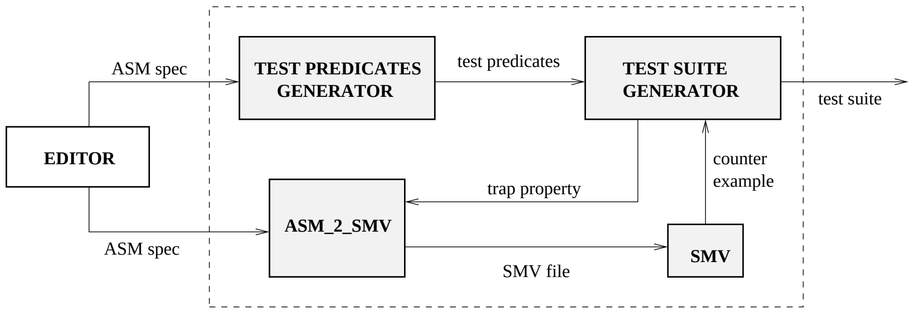

Figure 1: Tool Architecture

The component `ASM_2_SMV` receives the ASM specification and a trap property, and provides their SMV encoding. The ASM-to-SMV transformation proposed in [Del Castillo and Winter 1999] is used to this purpose.

The `SMV` model checker is used in the way described in [Section 6] to produce, if possible, a test sequence. The counterexample generated is given to the `TEST SUITE GENERATOR`.

The `TEST SUITE GENERATOR` checks (a) if the sequence covers other test predicates; (b) if the sequence can substitute some previously generated test sequences, that are discharged; (c) if there exists a test predicate not covered yet. If all the test predicates have been covered, the `TEST SUITE GENERATOR` provides in output the test suite; otherwise, it recalls the `ASM_2_SMV` supplying the trap property of an uncovered predicate.

The SMV encoding of ASM specifications is still manually done, while test predicates are automatically generated from ASM specifications written in the AsmGofer syntax [Schmid 1999]; the test suite generation is completely tool supported.

8 Experimental Results

We apply our method to the SIS example, used in other papers [Gargantini and Heitmeyer 1999, Ammann et al. 1998] as case study for testing purposes. Table 1 reports the experimental results.

Coverage criterion	**Number of test preds**	**Generated tests**	**Useful tests**	**Infeasible tests**
Rule	18	5	3	
Rule update	11	2	2	
2-parallel rule	7 / 72	4	1	3
MCDC	38	3	3	
Total	74	14	9	3

Table 1: Testing results for the SIS

In the generation of the test sequences, we start from the simplest coverage criteria, the rule coverage and the update coverage, and then we proceed trying the stronger criteria, the parallel-rule coverage and the MCDC.

For every test sequence generated for one test predicate, we check if that sequence also covers other test predicates, to avoid the generation of test sequences for test predicates already covered. This explains the reason why the test sequences generated, as indicated in table 1, are much fewer than the number of test predicates (14 against 74).

We check during generation if a test sequence is useful or can be discharged because all the test predicates it covers are already covered by other test sequence of the test suite. This explains the difference between the corresponding values of columns **generated tests** and **useful tests**, and why the final test suite includes only 9 test sequences out of the 14 originally generated.

In case of the 2-parallel rule coverage, the tool detected 3 infeasible test predicates (last column). We considered 7 test predicates over the all 72, namely `R2||R5`, `R3||R5`, `R1||R5`, `R1||R6`, `R1||R2`, `R5||R6`, `R4||R6`. The first four generate test sequences, but only the ones generated by `R1||R6` is useful, whereas `R1||R2`, `R5||R6`, `R4||R6` generate infeasible cases.

In case of rule update coverage, the test predicates not already covered by rule coverage, are derived from `R2` and `R4`.

In case of modified condition decision coverage, new test sequences are generated for rules `R5` and `R6`.

8.1 Practical Use of Test Suites

Test suites allow *conformance testing*, i.e. checking whether an implementation conforms to its specification. Our tool provides the tester with text files of test sequences (sequences of values of inputs, outputs, and internal functions). To test the conformance of a given implementation P by means of a test sequence S generated by our tool from an ASM specification of P, a tester should supply P with inputs of S, and compare outputs of P with the expected outputs in S.

Another interesting use of test suites concerns the validation of specifications through (graphical) simulation driven by test sequences. Before the implementation process takes place, simulating suitable scenarios allows the customer to observe the behavior of the specified system, and to check if the specification meets the expected requirements.

These aspects concerning the practical use of test suites are beyond the scope of this paper which focuses on coverage criteria definition and automatic test suite generation. However, these aspects are currently under investigation. We refer the reader to [Section 10] for future work.

9 Related Work

The use of formal languages in testing is tackled in [Stocks and Carrington 1996], where the Z notation is used to define test templates (similar to our test predicates) and to provide a formal framework for the entire testing process. A further problem is the derivation of such test templates from a specification [Amla and Ammann 1992] and the definition of coverage criteria for Z operational specifications, with the main goal of introducing an objective measure of coverage, independent of the implementation.

Several authors define coverage criteria or selection criteria for formal specifications, but leave still unsolved the problem of the generation of test suite. [Fujiwara et al. 1991] defines a test selection method for finite state machines, but it is not clear how to extend their method to more powerful formalisms (for example to ASMs). [Offutt et al. 1999] introduces several interesting coverage criteria for state based specifications and those might be adapted to ASM specifications. They present also a tool that semi-automatically derives test data sequences, but still some intervention of the tester is necessary.

There exist several attempts to avoid any human effort and develop completely automatic tools. To this purpose, several works exploit the counter example generation of the model checking algorithms. One of the earliest approaches using a model checker to generate test sequences can be found in

[Engels et al. 1997], where a test sequence is generated for a particular test predicate defined by the designer (called *testing purpose*). A major weakness of this approach is the reliance on the designer for introducing testing purposes and for manually translating the specification in Spin. [Ammann et al. 1998] uses the model checker SMV in combination to the mutation analysis to generate tests from mutated specifications. The coverage is measured in terms of the number of incorrect mutations that can be detected (they call this criterion *mutation adequacy*). In [Gargantini and Heitmeyer 1999] tests are generated using model checkers (both SMV and Spin) from SCR specifications to achieve a coverage similar to the well known branch coverage for programs, or to cover particular system requirements.

To the best of our knowledge, the only methods for generating test suites from ASM specifications are those recently developed by the Microsoft group in Redmond [Barnett et al. 2001, Grieskamp et al. 2001]. In the former, in order to find a test suite, they extract a finite state machine from ASM specifications and then use test generation techniques for FSMs [Lee and Yannakakis 1996]. However, the problem of reducing ASMs to FSMs is hard and might be even undecidable. In the latter, they check an implementation against its ASM specification by comparing at runtime the observed behavior with the specified one. To assess the quality of the testing activity, they still need to define some coverage criteria.

10 Future Work

As future work we plan to define more powerful coverage criteria, for example some criteria defined in [Offutt and Liu 1999, Offutt et al. 1999], missing condition coverage [Kuhn 1999], the *meaningful impact strategy* [Weyuker et al. 1994], and the *complete condition coverage*. These criteria are adequate to discover greater classes of errors (e.g. missing or wrong conditions inside guards).

We want to investigate the use of test suites generated by our method for conformance testing. To drive real programs with the inputs of a test suite, the tester might have to build an ad-hoc layer of software or a driver, or directly modify (if possible) the original program.

We also want to study the fault detection capability of coverage criteria to provide evidence that they lead to good and efficient test suites suitable to find errors in programs. Testing both mutated programs and programs with injected faults is a classical way for assessing the fault detection capability of tests. We plan to apply this technique to evaluate our approach. This activity could lead to the definition of a hierarchy among criteria.

We still continue working to improve the tool and to add an automatic translation in the language of the model checker, and to explore the use of other model checkers, e.g. Spin.

Finally, we are investigating the use of abstractions to avoid the model checker state explosion problem.

Acknowledgements

We are grateful to the anonymous referees for their insightful and constructive comments.

References

[Amla and Ammann 1992] Amla, N. and Ammann, P. (1992). Using Z specifications in category partition testing. In *Compass'92: 7th Annual Conference on Computer Assurance - Building the Right System Right*, pages 3–9, Gaithersburg, Maryland. National Institute of Standards and Technology.

[Ammann et al. 1998] Ammann, P., Black, P., and Majursk, W. (1998). Using model checking to generate tests from specifications. In *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia.

[Barnett et al. 2001] Barnett, M., Campbell, C., Schulte, W., and Veanes, M. (2001). Specification, simulation and testing of com components using abstract state machines (extended abstract). In *Formal Methods and Tools for Computer Science - Eurocast 2001*, pages 266–270.

[Beizer 1983] Beizer, B. (1983). *Software Testing Techniques.* Van Nostrand Reinhold.

[Blackburn and Busser 1996] Blackburn, M. R. and Busser, R. D. (1996). T-VEC: A tool for developing critical systems. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, Gaithersburg, Maryland. National Institute of Standards and Technology.

[Blackburn et al. 1997] Blackburn, M. R., Busser, R. D., and Fontaine, J. S. (1997). Automatic generation of test vectors for scr-style specifications. In *Compass'97: Twelfth Annual Conference on Computer Assurance*, pages 54–67, Gaithersburg, Maryland. National Institute of Standards and Technology.

[Börger 2000] Börger, E. (2000). Abstract state machines at the cusp of the millenium. In Gurevich, Y., Kutter, P. W., Odersky, M., and Thiele, L., editors, *Abstract State Machines: Theory and Applications*, number 1912 in LNCS, pages 1–8. Springer-Verlag.

[Börger et al. 2000] Börger, E., Riccobene, E., and Schmid, J. (2000). Capturing requirements by Abstract State Machines: The Light Control case study. *Journal of Universal Computer Science*, 6(7).

[Chilenski and Miller 1994] Chilenski, J. J. and Miller, S. P. (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200.

[Del Castillo and Winter 1999] Del Castillo, G. and Winter, K. (1999). Model Checking Support for the ASM High-Level Language. Technical Report TR-RI-99-209, Universität-GH Paderborn.

[Engels et al. 1997] Engels, A., Feijs, L., and Mauw, S. (1997). Test generation for intelligent networks using model checking. In Brinksma, E., editor, *TACAS'97, Lecture Notes in Computer Science 1217*, pages 384–398. Springer.

[Frankl and Weyuker 1993] Frankl, P. G. and Weyuker, E. J. (1993). A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213.

[Fujiwara et al. 1991] Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603.

[Gargantini and Heitmeyer 1999] Gargantini, A. and Heitmeyer, C. (1999). Using model checking to generate tests from requirements specifications. In Nierstrasz, O. and Lemoine, M., editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, volume 1687 of *LNCS*.

[Grieskamp et al. 2001] Grieskamp, W., Gurevich, Y., Schulte, W., and Veanes, M. (2001). Testing with abstract state machines (extended abstract). In *Formal Methods and Tools for Computer Science - Eurocast 2001*, pages 257–261.

[Gurevich 2000] Gurevich, Y. (2000). Sequential Abstract State Machines capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1.

[Hierons and Derrick 2000] Hierons, R. and Derrick, J. (2000). Special issue on specification-based testing. *Software testing, verification & reliability (STVR)*, 10(4):201–262.

[Kuhn 1999] Kuhn, R. (1999). Fault classes and error detection capability or specification-based testing. *ACM Transactions on Software Engineering and Methodologies*, 8(4):411–424.

[Lee and Yannakakis 1996] Lee, D. and Yannakakis, M. (1996). Principles and methods of testing finite state machines - A survey. In *Proceedings of The IEEE*, pages 1090–1123. Published as Proceedings of The IEEE, volume 84, number 8.

[McMillan 1992] McMillan, K. L. (1992). The SMV system. Technical report, Carnegie-Mellon University, Pittsburgh, PA. DRAFT.

[Miller 2001] Miller, K. (2001). A modest proposal for software testing. *IEEE Software*, 18(2):96–98.

[Myers 1979] Myers, G. J. (1979). *The Art of Software Testing*. John Wiley and Sons, IBM Systems Research Institute.

[Offutt and Liu 1999] Offutt, A. J. and Liu, S. (1999). Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62.

[Offutt et al. 1999] Offutt, A. J., Xiong, Y., and Liu, S. (1999). Criteria for generating specification-based tests. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131.

[P.J.Coutois and D.L.Parnas 1993] P.J.Coutois and D.L.Parnas (1993). Documentation for safety critical software. In *Proceedings of 15th IEEE International Conference on Software Engineering (ICSE'93), Baltimore, MD*, pages 315–323.

[Podgurski and Clarke 1990] Podgurski, A. and Clarke, L. A. (1990). A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79.

[Richardson et al. 1992] Richardson, D. J., Aha, S. L., and O'Malley, T. O. (1992). Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. Springer.

[Schmid 1999] Schmid, J. (1999). ASM Gofer. `http://www.tydo.de/AsmGofer/`.

[Stocks and Carrington 1996] Stocks, P. and Carrington, D. (1996). A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793.

[Weyuker et al. 1994] Weyuker, E., Goradia, T., and Singh, A. (1994). Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363.

[Weyuker 1979] Weyuker, E. J. (1979). Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598.

[Zhu et al. 1997] Zhu, H., Hall, P. A. V., and May, J. H. R. (1997). Software unit text coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427.

A SIS ASM Specification

```
MONITORED VARIABLES
  Block : {off,on};  Reset : {off,on};  WaterPressure: 0..2000;

CONTROLLED VARIABLES
  Pressure : {TooLow, Normal, High};
  Overridden : boolean;
  SafetyInjection : {on,off};

STATIC VARIABLES
  Low = 900;  Permit = 1000;

RULES
R1: WaterPressure becomes greater than Low,
    then Pressure from TooLow to Normal
 if WaterPressure >= Low and Pressure = TooLow
 then Pressure := Normal

R2: Pressure from Normal to High
 if WaterPressure >= Permit and Pressure = Normal
 then Pressure := High
      Overridden := false

R3: Pressure from TooLow to Normal
 if WaterPressure < Low and Pressure= Normal
 then Pressure := TooLow

R4: Pressure from High to Normal
 if WaterPressure < Permit and Pressure = High
 then Pressure := Normal
      Overridden := false

R5: The controller resets the SIS
 if Reset = on and (Pressure = TooLow or Pressure = Normal)
 then Overridden := false

R6: The controller ovverrides the SIS
 if Block = on and Reset = off and Pressure = TooLow
 then Overridden := true

R7+R8: When Pressure is TooLow, SafetyInjection is on
    unless the system is Overridden
 if Pressure = TooLow
 then if Overridden
      then SafetyInjection := off
      else SafetyInjection := on

R9: When Pressure is Normal or High, SafetyInjection is always off
 if Pressure != TooLow
 then SafetyInjection := off
```