Model Checkers for Test Case Generation: An Experimental Study

Muralidhar Talupur

Carnegie Mellon University

Abstract. In this paper we study the performance of various model checkers in test case generation scenario. The model checkers we consider are Cadence SMV, NuSMV, NuSMV's Bounded Model Checker (BMC), the explicit state model checker SPIN and CMU's SMV. The test cases are generated from specifications written in the SCR language. We suggest a strategy for minimizing the test generation time while ensuring the quality of the test cases. We also demonstrate through experiments that our tests achieve significant code coverage over the actual code generated from SCR specifications.

1 Introduction

Test case generation based on software specification [4], is a formal method for generating test cases. Test cases generated using this method are guaranteed to achieve specified levels of code coverage. We will briefly describe the method below, for a full treatment of the topic see [4].

The test case generation method makes crucial use of the ability of model checkers to return counter examples when a model does not satisfy a specified property. Given an SCR specification, $\mathcal{S}$, it is first translated into the language of the model checker. Call the resultant model $\mathcal{M}$. A set of *test predicates*, Φ, is then picked out from the specification $\mathcal{S}$ depending on the coverage criterion used. These test predicates are usually conditions/predicates occurring in the specification. Then a model checker is run on the model, $\mathcal{M}$, to see if the negations of test predicates Φ are ever violated. If some of the test predicates are violated then the corresponding counter example, produced by the model checker, can be used to generate a test case. This test case is guaranteed to cover the corresponding test predicate in the specification. In the following, we will refer to negation of a test predicate as a *trap property*.

The schematic representation of the process is shown below.

Once the test cases are generated they can be run on the real code corresponding to the specification to see how well the code is covered. This report describes our experiments with this test generation method. The goals of experiments are as follows:

Feasibility. Finding out whether the test generation method is really useful. That is, are we able to generate test cases in reasonable time?

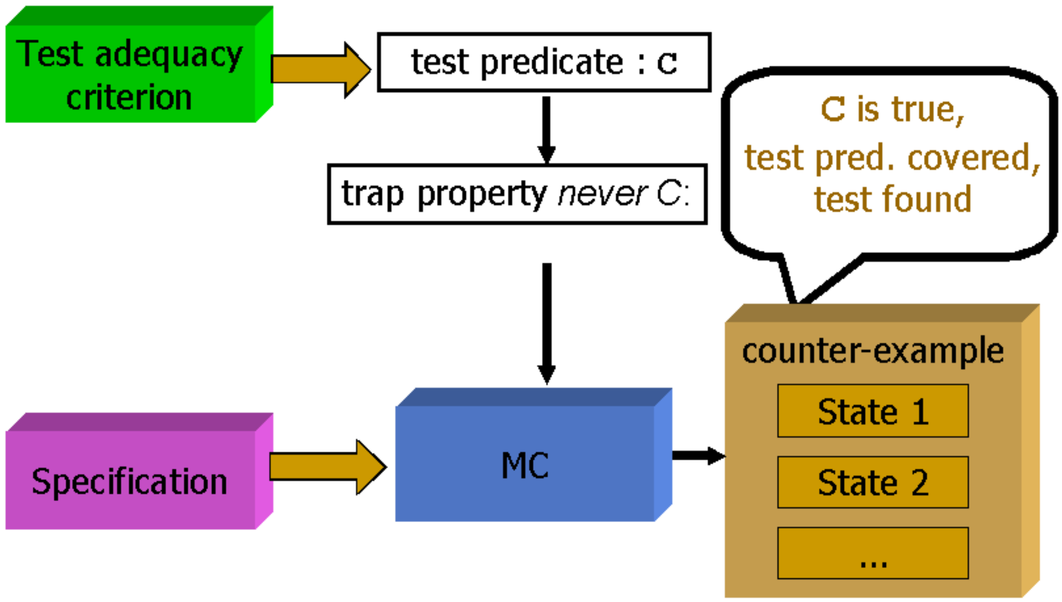

Fig. 1. Schematic representation of the Test Generation Process

Performance of Model Checkers. Studying the performance of various model checkers. It is well known that performance of model checkers changes depending on the size of the model, length of the counterexample, [1]. Our goal was to find out how model checkers perform in the test generation scenario.

Best Strategy. What is the best strategy to generate tests given a specification? That is, how should the different model checkers be used to produce the test cases in the least possible time?

Code Coverage. Once the test cases are generated, how well do these test cases cover C-code generated from the specification?

The rest of the report is organized as follows: In the next section, we describe the different model checkers we used. The section after that briefly talks about trap properties. Section 4 describes our experiments and results. Section 5 concludes the paper.

2 Model Checkers

We study the performance of the following Model Checkers in the test generation scenario:

– BDD based Model Checkers: Cadence's SMV (Cad SMV), NuSMV, CMU SMV

– SAT based Bounded Model Checker: NuSMV's BMC (NuBMC for short)
– Explicit State Model Checker: SPIN

We give a brief description of each of the different types of Model Checkers below.

2.1 BDD based model checkers

BDD based Model Checkers use BDDs to symbolically represent the state space of a model. Given a property to be verified, the Model Checkers in effect perform a breadth first search over the symbolic state space. Because of this, if any counterexample is found then it will be the shortest counterexample. If no counterexample is found then the Model Checker concludes the property to be verified is true on the model.

The three symbolic model checkers that we consider are: Cad SMV, NuSMV, and CMU SMV. Of the three, the last was the slowest and was left out of the experiments.

2.2 Explicit state

Explicit state Model Checkers explore the explicit state space of a model. SPIN is the only explicit state Model Checker considered in our experiments. SPIN first takes the input specification and generates an equivalent C-code. This C-code is compiled and executed. The state space exploration thus takes place while execution.

SPIN performs a nested depth first search of the state space. Because of this, counterexamples found by SPIN are not guaranteed to be the shortest. In fact, in our experiments, some of counterexamples generated by SPIN are extremely long, in the order of 10,000 states.

There are switches in SPIN which can be turned on to make SPIN look for the shortest counterexample. We considered three switches during our experiments, `REACH`, `i`, and `I`. `REACH` combined with `i` gives the shortest counterexamples but takes lots of time. `REACH` with `I` looks for short counterexamples but not the shortest. In our experiments, `REACH` with `I` did not help much in shortening the counterexamples whereas `REACH` with `i` took much too long to find counterexamples.

`SPIN` does an exhaustive search (unless `BITSTATE` is turned on), so it is able to prove a property correct.

2.3 Bounded Model Checker

SAT based Bounded Model Checkers (BMCs) search for counter examples of a specified length. The search problem is converted in to a SAT instance such that the SAT problem is satisfiable if and only if there is a counterexample of the specified length. So a BMC looks for counter examples of increasing length starting with length 1. It stops when a counterexample is found. This procedure is not complete in that the BMC does not stop if there is no counterexample. This is a significant disadvantage. In fact during our experiments, BMC got stuck several times during test case generation.

The BMC we used was NuBMC. We could not use Cadence SMV's BMC because it accepts only the stutter-free fragment of LTL and many of our trap properties need the next state operator.

The SAT solver zChaff was the SAT engine for NuBMC.

3 Trap Properties

Trap properties are derived from predicates/conditions for various branches in the specification. The particular set of trap properties used depends on the coverage criterion being used. In our experiments we used four coverage criteria: `table coverage`, `split mode coverage`, `full predicate coverage` with and without expanded events. For detailed descriptions of these coverage criteria see the appendix.

The branch conditions in SCR are over the current state variables and the next state variables. Trap properties are converted to CTL, LTL or other format depending on the Model Checker used. For SPIN, the trap property is modified by introducing new variables for the next versions of the state variables. The names of the new variables were usually the names of the original variables with a 'P' as suffix. For NuSMV BMC, the trap property is converted to the corresponding LTL formula. This conversion makes use of the next time operator 'X'. For this reason we cannot use Cadence's BMC which only takes stutter-free fragments of LTL as input. We could use Cadence's BMC when trap properties were one state properties (i.e when no next time variables were involved).

For NuSMV, CadSMV and SMV we convert the SCR property to CTL. This conversion is described in [5]. This conversion is possible because SCR properties deal with at most two consecutive time steps.

While translating SCR specification in to the language of Model Checkers, we could not translate @C-type events to CTL and LTL. We could do this translation easily for SPIN. An @C-type event, for example @C(X), is an event denoting change in the value of variable X, where X could be a non-boolean variable. Since @C event could involve comparison between two non-boolean variables and CTL, LTL do not allow such a comparison, the translation of @C events to CTL, LTL is not straightforward. So, for test predicates involving @C BMC and symbolic Model Checkers could not generate any test case. Also for test predicates involving conditions of the form X'= Y (where X' denotes the next time-step value of variable X), we could not obtain the corresponding CTL, LTL condition. We now have a solution to this problem, but the experiments described below were done prior to solving this problem.

4 Experiments and Results

We did our experiments on 7 SAL specs. Their names, the number of variables and approximate state space size are given below:

– Autopilot: 9 dependent vars, 10 monitored vars, 10ˆ12
– Cruise Control: 1 dependent vars, 5 monitored vars. 10ˆ2
– SIS: 3 dependent vars, 3 monitored vars, 10ˆ4
– Bombrel: 3 dependent vars, 9 monitored vars, 10ˆ7
– TTCP: 35 dependent vars, 19 monitored vars, 10ˆ8
– FPEspec: 27 dependent vars, 5 monitored vars, 10ˆ17

As can be seen from the above the size of the specs varied from 6 variables to 54 variables. These specifications are good representatives of the kind of specifications encountered in real situations.

We also used modifications of some of the above specifications. We changed the domain of certain variables to obtain larger specifications. This was done primarily to compare performance of SPIN and BDD-based Model Checkers.

As mentioned previously, the coverage criteria we used were the following:

1. `Table coverage`
2. `Split mode coverage`
3. `Full predicate coverage`
4. `Full predicate coverage with event expansion`

Each coverage criteria leads to a different set of test predicates (and trap properties). Not all trap properties are necessarily false on the model corresponding to the specification. Only those test predicates that are false lead to a counterexample and hence to a test case.

The coverage criteria are arranged in the order of increasing strengths. That is, a test suite satisfying a certain coverage criteria will also satisfy all the lower criteria. For example, a test suite satisfying `Full predicate coverage` with event expansion will satisfy all the other criteria as well. For further description see the appendix.

We have a tool, written in Java, that takes the test criteria and the specification and generates the test cases. The tool, called `scrtgtool`, also needs to be told which model checker to use. Given a test criteria, the tool first generates all the test predicates and then for each test predicate it tries to find a counterexample.

All the experiments were performed on a 3 GHz Pentium 4 processor, with 1 GB of main memory.

To measure the time we used the system command `time`. The test generation tool uses `time` command to measure the time taken by each to call to the model checker. Time measured by the command is the elapsed time and not CPU time. Since we were not running any other big programs, this is a good approximation of the CPU time.

4.1 Results

Our first goal is to check the feasibility of the test generation method, that is, if we could generate test cases using reasonable resources. We could generate test cases for all the specifications using one or the other Model Checker. With SPIN we could generate test cases for all specifications. With BDD-based MCs, we could generate test cases for all but two specifications (a modification of Autopilot and FPEspec). So the test case generation method is indeed a viable method for generating test cases.

Our second goal was to study the performance of different model checkers. Our results are described below:

Except for SIS, on all other specifications Cad SMV out-performs NuSMV, SPIN and NuBMC. The table below shows the time taken by each model checker on different specification. The time shown here is the average of time taken for all trap properties. We restricted NuBMC to look for counterexamples of length at most 12. Otherwise, NuBMC would not stop on trap properties that are true.

	C-SMV	NuSMV	BMC	SPIN
Autopilot	0.035	0.065	1.5	0.7
Cruise	0.02	0.02	0.1	0.4
SIS	4.4	9.7	60.0	0.44
Bombrel 1	0.43	0.40	100.0	2.1
Bombrel 2	0.13	0.18	–	1.2
TTCP	0.08	0.3	440.0	1.7
FPEspec	–	–	–	21.3

Table 1. Performance of different Model Checkers

As can be seen from the above table, for Bombrel and FPEspec some entries are blank, that is, we could not obtain results for those cases. For FPEspec we could not get any result for NuBMC or BDD-based Model Checkers. For Bombrel, BMC could not stop on some test predicates even after 48 hrs.

For SIS, we used four modifications. The modifications we made were increasing the domains of certain variables. For all the four variations the BDD and BMC-based model checkers were slower than SPIN. In fact, as we increased the domain sizes the time taken by SPIN changed only slightly whereas the time take by the other model checkers increased rapidly. The following table shows how the time taken by Model Checkers varies with the size of the variable domains. The variations were obtained by changing the range of a variable named `PressureRange`. The values of `PressureRange` in different specifications were

	PressureRange
SIS1	0..200
SIS2	0..800
SIS3	0..8000
SIS4	0..9000

The rest of the variable domains stayed the same.

	C-SMV	NuSMV	SPIN
SIS1	0.95	1.8	0.44
SIS2	4.4	9.7	0.44
SIS3	108.0	168.0	1.9
SIS4	170.0	217.0	2.3

For large domain sizes BDD and BMC based methods could not give any results whereas SPIN produced test cases where possible. We used an enlarged version of Autopilot to see how the Model Checkers perform on large models. SPIN gave us results, where as BDD and BMC based Model Checkers could not handle the large model.

Cad SMV was faster than NuSMV on all specifications except the very small ones. This is probably because of the heuristics that are used by Cad SMV. Moreover, Cadence SMV is optimized for performance whereas in NuSMV the emphasis is more on open-source development and less on optimizing the code. There are several places where NuSMV code can be optimized, but have been left unchanged for better readability.

For small specs with small counterexamples performance of NuSMV, NuBMC,Cad SMV were comparable. The main time component in SPIN execution was the compilation time. All the model checkers took less than 1.00s per test predicate for small specs with small counterexamples.

As the length of the counter example increases NuBMC started taking more time, as expected. For test predicates with long counterexamples, NuBMC took several hours. In some cases, NuBMC did not terminate even after 2 days. For the larger specifications, NuBMC is again slow even if the counterexamples are short.

Another disadvantage of Bounded Model Checkers is that they cannot prove a property correct, they can only find counterexamples. So for trap properties that are true, NuBMC does not stop. NuBMC's performance is comparable to Cadence SMV or NuSMV only if the spec is small and counterexamples are short (around 3 states long). So NuBMC and Bounded Model Checkers in general are not suited for test case generation.

Another crucial aspect is the length of the counterexamples produced. BDD-based Model Checkers and NuBMC always produced the shortest counterexample. SPIN, which does a nested depth first search on the state space, produces very long counterexamples. In our experiments SPIN produced many counterexamples running into tens of thousands of states. We tried turning on the I switch (in combination with REACH). Some counterexamples became shorter but many remained very long. The option `-i` with `REACH` seems to produce the shortest counterexample. But, it takes lots of time. For example, for SIS3 and for just one test predicate, SPIN with `REACH` and `i` turned on took more than 9 hours to find the shortest counterexample(for the same test predicate Cad SMV took less than 5 mins). Since there are around 50 test predicates per coverage criteria for larger specifications, the time to generate all test cases might go into several days for SPIN with `i` and `REACH` on.

We now summarize briefly the advantages and disadvantages of each Model Checker.

Cad SMV: Fastest on all most all specs and produces the shortest counterexamples. Disadvantage: Cannot handle specs with large domains.
NuSMV: Fast and produces the shortest counterexamples. Has the same disadvantage as CAD SMV
NuBMC: Good for small specs with small counterexamples. Major disadvantage is that it is not complete. So it gets stuck on trap properties that are true.
SPIN: If used without `-i` switch, it is fast. The disadvantage is it produces extremely long counterexamples. If used with `-i` and `REACH` switches it produces the shortest counterexamples but it takes very long time. SPIN has other switches too, like `I`, which provide other trade-offs between execution time and counterexample length.

In the light of these experimental results, the best strategy is to try using Cad SMV first. If the model is too large (that is SMV does not even finish building the model in reasonable time, say 4-5 mins) then use SPIN. Alternatively, one can wait till SMV suffers from state explosion and then switch over to SPIN with just `REACH` option.

4.2 Running the test cases on C-code

The fourth goal of our experiments was to see the coverage achieved by our test cases for a real code written based on the specification. We could obtain a C-code for autopilot specification and we ran our experiments on it. The coverage achieved by our tests is shown in the table below for each coverage criteria. In these experiments, we used the test cases generated by NuSMV (for autopilot specification NuSMV and Cad SMV took almost the same time and generated the same counter examples). Since we could not translate test predicates involving conditions like @C and conditions of the form X' = Y, the set of test cases is not complete (for the given criteria). The reason we could not use SPIN test cases was that the test cases were very long and running them through the code took more than 2 days to finish. SPIN with `I` switch turned on suffers from the same problem. SPIN with `i` turned on produces short counterexamples but it takes days to finish as mentioned earlier. The table below gives the code coverage achieved by different criteria.

As can be seen from the table, even an incomplete set of test cases achieves significant code coverage. In fact using the code coverage we figured out a mistake we made while running the experiments. There are different versions of autopilot available, we used one version to generate the tests and the C code was based on another version. By looking at the coverage of our tests on the C-code we could

	statement	Decisions	Basic cnds	Modified cnds	Multiple cnds
Table	76.4	76.5	64.8	32.8	37.5
Split	76.4	76.5	64.8	32.8	37.5
Full Pred	74.9	76.1	76.1	61.6	57.1
Full Ev	77.4	78.2	73.2	47.2	46.4

Table 2. Code Coverage for different criteria

figure out that the versions do not match. Some of the conditions in C-code were not covered, whereas we had test cases that covered exactly those conditions in the specification. This led us to the difference in the versions.

5 Conclusion

In this report we presented a study of the performance of various model checkers in test case generation scenario. Our results indicate that where Cadence SMV on realistic specifications produces the best test cases and in least possible time. If a specification is too large to be handled by BDD-based methods then we can use SPIN. Our experiments also demonstrate the overall viability of the test generation method. We have also included a brief section on the coverage achieved by our test cases on a real C-code.

A Basic Notation

In this section we provide some basic terminology. Adapting to our purposes some of the definitions common in literature on state transition systems [2, 6], we define a *test sequence* or *test* as follows:

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 function defined by tables.

Informally, a test sequence is a partial possible behavior of the system as specified and generally ends with a state where the test goal is achieved.

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

B Coverage criteria

In this section we define and explain several test coverage criteria for SCR. Formally [9, 3] a coverage criterion is a function that given a program P, its specification S, and a test set T returns true iff T is adequate to test P according to S. In specification-based approach coverage criteria are defined regardless the program P. Therefore, in the following a test adequacy criterion is a function that takes a specification S and a test suite T and return true (if T is adequate) or false (if T is not adequate). In this framework test coverage criteria can be used to evaluate a test set, regardless the way or the method it has been generated or selected.

Test adequacy criteria can be used to select adequate test sequences and as guidelines during the selection process. A test criterion may introduce a method or an algorithm to actually generate the tests, but it is normally defined regardless the precise algorithm used to select such test cases. Test cases can be selected by hand, or automatically generated using some algorithm and technique. In this context, formally speaking, a test selection criterion is a function that given a specification S returns a set of test sequences T that is adequate with respect to S. The actual definition of such function may involve some non determinism (e.g. some random sampling).

In the following sections we will provide several test criteria for SCR specifications.

B.1 Table Coverage

We start from the basic coverage, that we call table coverage, originally presented in [4] in a similar (but less precise) manner. Informally, with the table coverage criterion we want to cover all the cells (i.e. events and conditions) in SCR tables in one of modes that regard the cell???.

We say that a test suite T is adequate according to the table coverage criterion, iff

1. for every condition c, not equivalent to *false*, in each condition table, there exists a test sequence in T containing a state where c is true and the mode is equal to a mode in the c row.

2. for every event e at this stage `@T(X) or @T(Y)` is considered one event, not equivalent to *never*, in each event and mode table, there exists a test sequence in T containing a state where e happens and the mode is equal to a mode in the e row.

3. for each event and mode table T, there exists a test sequence in T containing a state where no event in T happens.

With the table coverage we simply test every condition and every event in the table.

To formalize test criteria we use *test predicates*. A test predicate is a formula over a state or over a pair of states (current and next one) 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 [8].

B.2 Split Mode Coverage

Table coverage generally does not distinguish modes, unless, of course, modes are already distinct in the table. Modes play an important role in the specification as well in the system and therefore deserve more attention. In order to overcome this limit, we analyze every predicate introduced for the basic coverage. If the predicate refers to one mode, the predicate is not changed. If the predicate refers to several modes or does not contain modes, it is split in several predicates, each for a distinct mode.

B.3 Full Predicate Coverage

Full predicate coverage is based on the idea that each atomic condition or event in a complex boolean expression should be tested separately, i.e. choosing the values of other atomic conditions and events in the expression in a way that only the atomic condition under observation influences the final value of the expression. The same goal can be found in the code-based testing criterion MCDC. For the detailed definition of the full predicate expression, please refer to [7]. We report and adapt for SCR only the main definitions.

Given a test predicate tp, we say that an atomic condition c (an atomic event e) determines the value of tp, if the remaining atomic conditions and events in tp have values that changing the truth value of c (e) changes the value of tp.

For each condition or event A in each table, and each atomic condition or atomic event a_i in A, the test set T must include test sequences containing a state where A is true and one state where A is false and a_i determines the value of A.

With the full predicate we test every atomic condition in conditions and how it affects the condition final value. For every atomic condition we search a couple of test cases, each containing a state, let s_1 and s_2 be the couple of states, such as:

- the atomic condition is true in s_1, and false in s_2
- the value of compound condition is different in s_1 and s_2 (true and false or vice-versa)
- the value of the other atomic conditions dose not affect the value of C

B.4 Expanding events

Mode splitting and other coverage criteria can be combined to get deeper coverage. Moreover events can be explicitly expanded in conditions according to their meaning (e.g. @T(X) = X' ∧¬ X).

Consider the event:

$\mathbf{C}_{22}$ `Pressure = Permitted` ∧ `@T(Block=On)` ∧ `Reset=Off`

It can be expanded in the following condition:

$\mathbf{C}_{22}$ `Pressure = Permitted` ∧ `Block'=On` ∧ `Block` ≠ `On` ∧ `Reset=Off`

We apply FullPredicate to this condition to get five test predicates, one where every atomic condition is true and four where each conjoint of the compound condition is false:

C_{221} `Pressure = Permitted` $\wedge$ `Block' = On` $\wedge$ `Block` $\neq$ `On` $\wedge$ `Reset = Off`
C_{222} `Pressure` $\neq$ `Permitted` $\wedge$ `Block' = On` $\wedge$ `Block` $\neq$ `On` $\wedge$ `Reset = Off`
C_{223} `Pressure = Permitted` $\wedge$ `Block'` $\neq$ `On` $\wedge$ `Block` $\neq$ `On` $\wedge$ `Reset = Off`
C_{224} `Pressure = Permitted` $\wedge$ `Block' = On` $\wedge$ `Block = On` $\wedge$ `Reset = Off`
C_{225} `Pressure = Permitted` $\wedge$ `Block' = On` $\wedge$ `Block` $\neq$ `On` $\wedge$ `Reset` $\neq$ `Off`

Note that C_{222} models the occurrence of the same conditioned event `@T(Block=On) WHEN Reset=Off` in a different mode. In C_{223} and C_{224} the event does not occur (since `Block` does not change C_{224}, and since `Block` does not become equal to `On` in C_{223}). In C_{225} the event `@T(Block=On)` occurs but the condition is false, hence the conditioned event does not occur.

References

1. N. Amla, R. Kurshan, K. McMillan, and R. Medel. Experimental analysis of different techniques for bounded model checking. *TACAS*, 2003.
2. Paul Ammann, Paul Black, and William Majursk. Using model checking to genrate tests from specifications. In *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, December 1998.
3. Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.
4. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel 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*, September 6–10 1999.
5. Ralph D Jeffords. Translating scr properties into ltl and ctl. Technical Report 5540-293A:rdj, NRL, 1998.
6. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of The IEEE*, pages 1090–1123, August 1996. Published as Proceedings of The IEEE, volume 84, number 8.
7. J. Offutt, S. Liu, A. Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, 13(1), March 2003.
8. Elaine J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, 1979.
9. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.