Using Spin to Generate Tests from ASM Specifications

Angelo Gargantini[1], Elvinia Riccobene[2], and Salvatore Rinzivillo[3]

[1] CEA - Università di Catania (Italy)
[2] DMI - Università di Catania (Italy)
[3] DI - Università di Pisa (Italy)

Abstract. In this paper we introduce an algorithm to automatically encode an ASM specification in PROMELA, the language of the model checker Spin. Exploiting the counter example generation feature of Spin, we present a method to automatically generate from ASM specifications test sequences which accomplish a desired coverage. ASMs are used as test oracles to predict the expected outputs of units under test. A prototype tool that implements the proposed method is presented. Experimental results in evaluating the method are reported. The experiments include test sequence generation, tests execution, code coverage measurement for a case study implemented in Java, and comparison with random tests generation. Benefits and limitations in using model checking are discussed.

1 Introduction

Along the process of software development, testing is of extreme importance to produce high-quality products. According to the recent NIST report [16], the cost of inadequate software testing infrastructure for the US economy is estimated to be $59.5 billions. Software needs to be tested to uncover development and coding errors, to asses its reliability and dependability, and to convince customers that the performance is acceptable. However, software testing is extremely costly and time-consuming. *Specification based testing* [10] offers an opportunity to significantly reduce the testing costs.

In specification based testing, a specification can be used as *test oracle* [13], i.e. as authoritative font of the expected system behavior, and as a means to assess correctness of implementations. Moreover, *test adequacy criteria* can be derived from a specification [18]. They determine if a test suite is adequate to test a software, whether enough testing has been performed or further tests are needed. A specification can also provide *selection criteria* of adequate test suites. Normally a selection criterion introduces some algorithms or techniques to actually generate test sequences from formal specifications.

In [8] we showed how to use an ASM as test oracle, we introduced and motivated several test criteria for ASMs, and we briefly explained how to use such test criteria to generate test suites exploiting the model checker SMV. In this paper we investigate further the use of model checkers for test suite generation.

Section 2 introduces the model checker Spin [11] and a novel algorithm to translate (a particular class of) ASMs in PROMELA, the language of Spin. This translation also allows using Spin to prove properties of ASM models, but formal verification is outside the scope of this paper. Section 3 explains the method of generating test sequences using model checkers and how, in particular, the model checker Spin is exploited to generate tests. Benefits and limitations in using model checkers for test generation are discussed. Section 4 briefly presents the tool we have developed. In Section 5, interesting experimental results in applying our method to real cases and to real code are presented. For the Safety Injection System (SIS) [5], test sequences are generated, and the coverage of the SIS Java code provided by these tests is measured through a code coverage analyzer. Moreover, these tests generated from the ASM specification are compared with test sequences randomly generated. Finally, we present the generation of test sequences for the OpenDoor example, overcoming some problems discussed in [9]. Section 6 concludes discussing related work and future directions.

2 Encoding ASMs in Spin

The model checker Spin [11] has been successful used for software design and verification, and to trace logical design errors in distributed systems design. It uses a high level language, called PROMELA (PROcess MEta LAnguage), to specify systems. The tool checks the logical consistency of a specification and is able to prove system properties. It works *on-the-fly*, which means that it avoids the need to construct of a global state graph as a prerequisite for the verification of any system properties.

Spin supports random, interactive and guided simulation, and both exhaustive and partial proof techniques. The tool is meant to scale smoothly with problem size, and is specifically designed to handle even very large problem sizes.

Spin can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed and verified using assertions instead of LTL.

A property is verified by creating an on-the-fly verifier program that can be compiled and then executed to eventually find, if the property is proved to be false, a counterexample. To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques.

Each PROMELA model has an `init` process, that represents the entry-point for the model simulation and verification. The other processes are instantiated and executed by this main process. All these processes are executed asynchronously. A model can contain only one `init` process.

Two processes can communicate through a *buffered channel*. Each channel has a fixed size and a tuple of types $T_1, \ldots, T_n$ for messages that can be re-

ceived on that channel. Each message contains n fields, where the ith fields is of type T_i. A process sends a message to a channel `cname` with the statement `cname!expr1,...,exprn`. A message is read from a channel `cname` with the instruction `cname?var1,...,varn`, where `var1`,..., `varn` are the variables where to store the actual values of the message. When a process reads a message from a channel, this message is removed from it. If a process tries to write on a full channel it is suspended until there is space for a new message. Analogously, if a process tries to read from an empty channel, it is suspended until a message arrives. *Rendezvous* channels are a special type of channels that permit synchronization between processes. These channels are zero-sized. When a process tries to write into a rendezvous channel, it is suspended until another process tries to read from it. Spin supports process communication also through *shared memory*.

In this section we introduce a method to translate ASM specifications to PROMELA. We consider only a particular subset of sequential ASMs, that we call ASM_0. ASM_0 specifications use only nullary dynamic functions (variables) and static functions that are those built-in functions of Spin (like $+$, &,). ASM_0s have only *finite integer*, *boolean* and *enumeration* domains, and do not allow the construct *choose* (non determinism is only in monitored variables). We assume ASM_0 specifications to be consistent with respect to the updates.

2.1 States

We show here how an ASM_0 state is encoded in PROMELA.

Universes. Integer and boolean domains of ASM_0 are simply mapped into PROMELA `int` and `bool` types, respectively. Enumerated domains can be modeled as integers through the definition of suitable constants. For example, an enumeration $EnumName = Enum_0, \ldots, Enum_n$ is encoded as a sequence of constants declaration:

```
#define Enum_0 0
#define Enum_1 1
...
#define Enum_n n
```

Variables. ASM_0 dynamic nullary-functions are encoded as PROMELA variables. Each ASM_0 controlled variable is translated into a pair of variables one yielding the *current value* and one yielding the *next value*. A similar method is used also in [14]. For example, for the integer variable *aVariable* we have:

```
int aVariable;
int aVariable__P;
```

where the suffix `__P` is used to distinguish the variable yielding the next value. The current value `aVariable` is used in guards and in expressions evaluation, while `aVariable__P` is updated by rules. After all rules are executed, each new value is copied in the current one, so to obtain the new state.

Monitored variables updating. An ASM_0 monitored variable does not need to be encoded as a pair of variables, because it is changed only by the environment and never updated by any rule. Monitored variables might have a completely random behavior, i.e. at each step they can take any value in their domain, or they may obey to some constraints (for example, the pressure in a boiler can change in a limited range from one state to the next one). ASM_0 allows to specify the interval and the maximum variation between two consecutive values for integer monitored variables. If a variable can have only values in an interval [a,b], we say that it is of type $\texttt{int}([a, b])$. Moreover, an integer variable may have a fixed *delta*, i.e. a maximum variation of value from the current state to the next one. When a *delta* is provided, we say the location is of type $\texttt{int}(interval, delta)$. If f is a monitored variable of type $\texttt{int}([min_f, max_f], \delta_f)$ whose current value is t, then in the next state the variable value will be in $[t - \delta_f, t + \delta_f]$. For example, let x be a variable of type int([0,200],5). Then the minimum value is 0, the maximum is 200; if the value of x in the current state is 120 then in the next state it could be in $[115, 125]$.

In Spin is possible to use a non-deterministic statement to choose one of these values. For example, the monitored location `aMonVariable` of type int([1, 10],2) is simulated as:

```
if
  :: skip
  :: (aMonVariable - 1 >=  1) -> aMonVariable = aMonVariable - 1;
  :: (aMonVariable + 1 <= 10) -> aMonVariable = aMonVariable + 1;
  :: (aMonVariable - 2 >=  1) -> aMonVariable = aMonVariable - 2;
  :: (aMonVariable + 2 <= 10) -> aMonVariable = aMonVariable + 2;
fi;
```

If the variable has no *delta*, then the next value is chosen randomly in the interval definition. For example, if the monitored location `aMonVariable` has type int([1, 10]) in PROMELAwe have:

```
if
  :: aMonVariable = 1;
  :: aMonVariable = 2;
  ...
  :: aMonVariable = 10;
fi;
```

Variables of enumerated type can assume one of the possible values in type declaration. In Spin, a monitored variable `enumVariable` of type *EnumType* = *firstItem*, *secondItem*, *thirdItem*, is simulated as:

```
if
  :: enumVariable = firstItem;
  :: enumVariable = secondItem;
  :: enumVariable = thirdItem;
fi;
```

Constants. Constants are encoded with constant declarations.

2.2 Rules

ASM_0 rules are unfolded so to have only rules of the form

$$R_s = \texttt{if } G \texttt{ then } U \texttt{ else skip}$$

where U is a sequence of updates. The unfolding method is explained in the following. Let $R = \texttt{if } G \texttt{ then } R_T \texttt{ else } R_E$ be a generic ASM_0 rule, where:

$$R_T = \begin{cases} \texttt{if } G_T^1 \texttt{ then } R_T^1 \\ \cdots \\ \texttt{if } G_T^n \texttt{ then } R_T^n \end{cases} \quad R_E = \begin{cases} \texttt{if } G_E^1 \texttt{ then } R_E^1 \\ \cdots \\ \texttt{if } G_E^m \texttt{ then } R_E^m \end{cases}$$

The unfolding method showed in the following schema (similar to that presented in [4]) allows to translate R in a set of simpler rules like R_s:

$$R = \texttt{if } G \texttt{ then } R_T \texttt{ else } R_E \rightarrow \begin{cases} R_T^1 = \texttt{if } G \wedge G_T^1 \texttt{ then } R_T^1 \\ \cdots \\ R_T^n = \texttt{if } G \wedge G_T^n \texttt{ then } R_T^n \\ R_E^1 = \texttt{if } \neg G \wedge G_E^1 \texttt{ then } R_E^1 \\ \cdots \\ R_E^m = \texttt{if } \neg G \wedge G_E^m \texttt{ then } R_E^m \end{cases}$$

In the following we present two methods for the translation of rules like R_s.

Flat Translation. The first method uses only one process `init` to execute the model. For each rule we introduce an `if ... fi` statement. So each rule R_i: $R_i = \texttt{if } G_i \texttt{ then } U_i$ is translated into:

```
if
  :: G_i -> U_i;
  :: else -> skip;
fi;
```

The final model for the model checker is shown in Figure 1. After enumeration, variables and constant declarations, the `init` process begins. It contains an unique `do` cycle that performs monitored variable updating, executes the rules and updates current variable values. Each `do` cycle corresponds to a step of the original ASM_0 .

Chan Translation. This method uses a process for each rule, so that all the rules are executed asynchronously. The main process has the task of synchronizing the execution of all the rules. This synchronization is possible through *rendezvous* channels. Each process (rule) has one channel and the rule execution is suspended until a message arrives in the channel. For a rule $R_i = \texttt{if } G_i \texttt{ then } U_i$, we have in PROMELA:

```
mtype={start}
chan r_i_chan = [0] of {mtype}

proctype r_i(){
  do :: atomic{
          r_i_chan?start;
          if
            :: G_i-> U_i;
            :: else -> skip:
          fi
        }
  od
}
```

The statement `atomic` ensures that the statements in the brackets are executed atomically, without interruptions. The rule body is encoded in the same way as in the *flat* method. The statement `r_i_chan?start` tries to read a message `start` from the channel `r_i_chan`. The keyword `mtype` define a special type for messages in channels.

In Figure 1 is shown a schema of a Spin model obtained with chan translation. The `init` process starts all the processes for the rules. Inside the main `do` cycle, for each rule `r_i`, the statement `r_i!start` fires the rule `r_i`:

```
/* init function */
init{                /* main processing loop */
  run r_i();         /* start rule processes */
  do ::
    ...              /* Simulation of monitored variables */
  atomic{            /* Rules execution */
     r_i!start;      /* Fire rule r_i */
     ...
     ...             /* Locations update */
  } od
}
```

Comparison We have compared the two translation methods for the specification of the case study SIS [5]. Execution of PROMELA specifications obtained

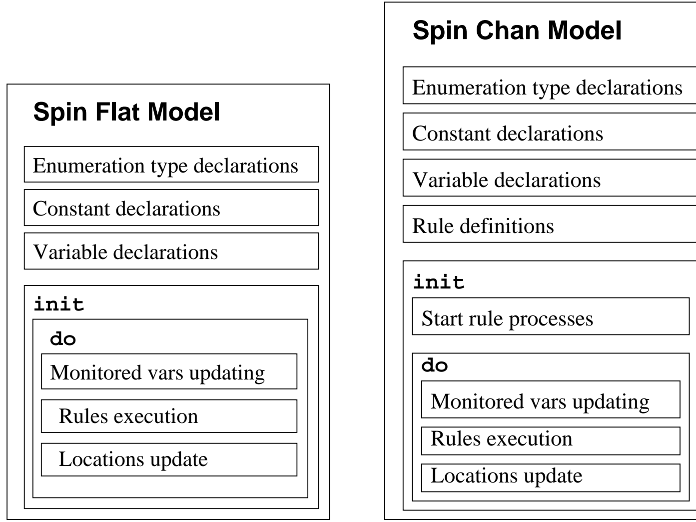

Fig. 1. Translation structure

by chan translation requires much more memory than those obtained by flat translation. However, chan models run slightly faster than flat models.

Although we assume ASMs to be sequential, the proposed chan translation method could be easily extended to deal with distributed ASMs.

3 Automatic Generation of Test Sequences from ASM Specifications

In this Section we briefly review some fundamental definitions and concepts originally presented in [8]. Moreover, we introduce a new coverage criteria, explain the use of Spin for tests generation and discuss its benefits and limitations.

Testing criteria for ASMs. In [8], several structural coverage criteria for ASM specifications are defined. They are: *basic rule* coverage, *rule update* coverage, *parallel rule* coverage, and *MCDC*. Definitions and motivations for these criteria are not reported here for the sake of brevity; the reader can refer to [8] for more details. These criteria represent the coverage of particular behaviors of the specified system, and besides them, new criteria can be introduced to achieve other coverage. In this section, we define and explain the *full guard coverage* criterion, which is an adaptation of the full predicate coverage criterion defined in [12].

Definition 1. *A test suite T satisfies the Full Guard coverage criterion if for each rule guard G, T includes tests that cause each atomic condition c in G to*

result in a pair of outcomes where the value of G is directly correlated with the value of c.

In this definition, "directly correlated" means that c controls the value of G, that is, one of the following two situations occurs: either c and G have the same value (c is true implies G is true and c is false implies G is false), or c and G have opposite values (c is true implies G is false and c is false implies G is true). This criterion is very similar to the MCDC, but when testing a particular atomic condition, it does not require to hold fixed all other atomic conditions. For example, let the guard G be $a \wedge (b \vee c)$. Consider the atomic condition a. In order to test a in G according to the full guard coverage, we need two tests: a test t_1 with a true and $b \vee c$ true and a second test t_2 with a false and $b \vee c$ true. Therefore, $t_1 = \{a, b, \neg c\}$ and $t_2 = \{\neg a, \neg b, c\}$ are sufficient to cover a in G according the full guard coverage, but they are not adequate to cover a according to the MCDC. For the MCDC, indeed, b and c must have the same value in t_1 and in t_2.

Test Predicates. We formally define a coverage criterion 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 of the specification 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.

For example for the full guard coverage of a in $G = a \wedge (b \vee c)$, we introduce two test predicates: $tp_1 = a \wedge (b \vee c)$ and $tp_2 = \neg a \wedge (b \vee c)$. Test predicate tp_1 implies G, while tp_2 implies $\neg G$ and the value of G depends only on the value of a. We search tests which contain a state where tp_1 is true and a state where tp_2 is true.

Automatic Test Sequence Generation Using Model Checking. The method proposed for automatic test generation is based on the model checker Spin. This technique is explained in [8], where the model checker SMV is used instead of Spin, and it is here only briefly sketched.

To automatically generate test sequences, we exploit the model checker capability to discover counter examples for properties that do not hold in the model. The method consists in several steps. First, we compute for the specification and a desired coverage criteria the test predicates set $\{tp_i\}$. Second, we encode the ASM specification in PROMELA, the language of Spin, following the algorithm described in Section 2. Third, for each test predicate tp_i, we compute the test sequence that covers tp_i by trying to prove with Spin the *trap property* $\neg tp_i$. If Spin 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.

Infeasible Tests. Spin always terminates and one of the following three situations occurs. The best case is when Spin stops finding that the trap property is false, and, therefore, the counter example to cover the test predicate is generated.

The second case happens when Spin explores the whole state space without finding any state where the trap property is false, and, therefore, Spin proves $\neg tp_i$. In this case, we say that the test predicate is *infeasible* or *not coverable*. Infeasible test predicates are to be ignored for testing purposes. We define *feasible* [18] versions of our adequacy criteria requiring that a test suite T satisfies a coverage C if each *feasible* test predicate of C is true in at least one state of a test sequence of T. In the following, we always refer to feasible versions of coverage criteria.

In the third case, Spin terminates without exploring the whole state space 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, the user does not know if either the trap property is true (i.e. the test is infeasible) but too difficult to prove, or it is false but a counter example is too hard to find. When this case happens, our method simply warns the tester that the 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 should not surprise: the problem of finding a test that covers a particular predicate is undecidable [17]. Nevertheless, in our experience the third case is quite rare: for our case study it never happened.

Model Checking Limits. Model checking applies only to finite models. Therefore, our method works for ASM specifications having variables and functions with finite domains. However, this limitation does not preclude the application of our approach to models with infinite domains, because an ASM A with finite domains can be extracted from an ASM B with infinite domains, such that a test generated for A is still a valid test for B. 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 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.

Model Checking Benefits. Besides its limits, model checking offers several benefits. Existing model checkers, and Spin in particular, adopt sophisticated techniques to compute and explore the state space, and to find property violations. They represent a state and the state space in a very efficient way using state enumeration, hashing techniques and symbolic representations. Moreover, model checkers explore the state space using practical heuristics and other techniques. For example, Spin uses partial order reduction methods and on-the-fly state exploration. 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 Spin allows to compute test sequences from PROMELA specifications without any human interaction.

4 A Tool for Automatic Generation of Test Sequences

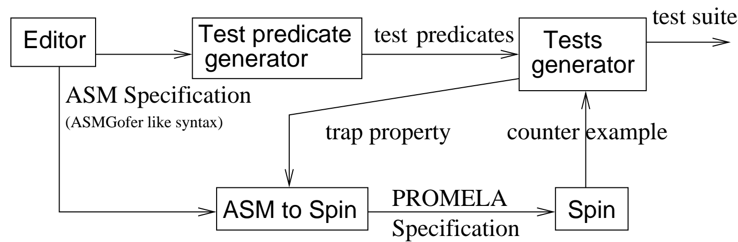

Fig. 2. Tool Architecture

We have developed a prototype tool[4] that applying the method presented in the previous section, automatically produces a complete test suite for several coverage criteria. The tool architecture is depicted in Figure 2, while some screen shots are showed in the Appendix. The Editor allows the designer to edit an ASM_0 specification. The ASM_0 grammar is a subset of the AsmGofer [15] language, extended to allow the definition of intervals and delta as explained in Section 2.1. The Test predicate generator computes the test predicates for the following coverage criteria: basic rule, complete rule, rule update, rule guard, full guard coverage. The Tests generator controls the generation of the test suite: (a) it takes each test predicate selected by the user and generates the corresponding trap property; (b) it calls the ASM_to_Spin component to translate the specification, together with the trap property to PROMELA; (c) it runs Spin; (d) it takes the counter example generated by Spin (if any), and decides, according to the user preferences, to stop or to continue test generation till each required coverage of test predicates has been satisfied. At the end a set of tests, or test suite, is produced. The test suite can be displayed and saved into a text file.

Remark 1. During the test generation for a test predicate, it also possible to check if other test predicates $p_1, ..., p_n$ are covered, adding to the PROMELA specification a statement like:

```
if
  :: P_1 -> printf("Covered: caseP_1");
```

[4] The tool will be available in a few weeks at the following URL address: www.dmi.unict.it/garganti/atgt/.

```
  fi;
  ...
  if
    :: P_n -> printf("Covered: caseP_n");
  fi;
```

The search for common coverage can reduce dramatically the number of tests to generate.

5 Experimental Results

The main goal of the experiments we conduct, is to provide evidence that our method is effective. In particular, we tackle the fundamental question about the quality of the tests our method generates, i.e. whether they are suitable to test programs developed from ASM specifications. Indeed, our test sequences, as always in specification based testing, are generated regardless the real code of the implementation, but the final goal of testing is discovering faults in code and producing quality software.

Coverage	# test pred.	# generated	# useful	Test name	Test length
Basic Rule	18	4	3	BR_R7_1 BR_NEG_R7_1 BR_R3 BR_R4	1 32 (useless) 910 1034
Complete Rule	9	0	0		
Rule update	11	2	2	UR_R4_2 UR_R2_2	not feasible 1000
Full predicate	38	0	0		
Rule guard	9	0	0		

Table 1. Test generation for SIS

First, we generate a complete test suite for the Safety Injection System (SIS) specification [5], for which the Java code is available. The complete description of SIS and its ASM specification can be found in [8]. Table 1 reports our results in test generation.

In the first and second column, respectively, Table 1 reports the coverage and the number of test predicates to achieve it. We start generating a test sequence for each test predicate, but since the tool has the capability to detect if a test predicate is covered by a test sequence already generated (as explained in Remark 1), the generation of one test for each test predicate is not necessary and we avoid to generate test sequences for test predicates already covered. This fact explains why the number of generated tests shown in the third column is less than the number of test predicates in the second column. Moreover, if a test

sequence t_1 covers a set of test predicates S_1 and another test sequence t_2 covers a set of test predicates S_2, and S_1 is a subset of S_2, then we consider t_1 useless and we do not include it in the test suite. For this reason, the number of useful tests shown in the forth column is less than the number of the generated tests. The fifth column reports the name of the tests and in the sixth column their length as number of states. Note that we discover that one test predicate is not feasible, i.e., as explained in Section 3, there is no test sequence that can cover it. In the end, the test suite for the basic rule coverage is compound of 3 test sequences, while the complete test suite to achieve every coverage is compound of 4 tests.

Second, we apply the test suite to the Java code for SIS. We modify the SIS code in such a way that the code reads the monitored values from a specific file instead of from the real environment. Then, we evaluate the code coverage using an automatic coverage tool, JCover[5], that is capable of computing the branch and statement coverage of a given Java program. Results are reported in Figure 3. Branch and statement coverage are achieved already by the test suite

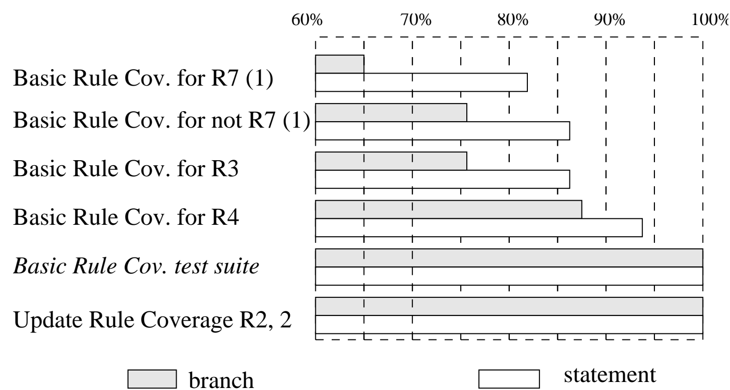

Fig. 3. Code coverage results

for the basic rule coverage. This experiment suggests that a programmer should generally achieve the statement and branch coverage by running code with test suites for the basic rule coverage. Intuitively, each branch in the code corresponds to some rule guard in the specification and, therefore, testing specification guards as true and false, corresponds to testing branches in the code. Moreover, we claim that other coverage criteria (for example Update Coverage Criteria) can guarantee a better code coverage than branch coverage, and that statement and branch coverage are coarser than our criteria. In the future, we plan to

[5] JCover^{TM} is a Java Code Coverage Analyzer developed by Man Machine Systems (www.mmsindia.com)

evaluate the code coverage with more power tools than JCover (for example, LDBA Testbed©).

To judge further the effectiveness of the generated test suite, we compare it with several randomly generated tests. In the generation of the random tests, we make monitored variables randomly change, taking into account only environmental constraints on monitored variables, i.e. their interval and the delta between their values in two consecutive states. Results are reported in the Figure 4.

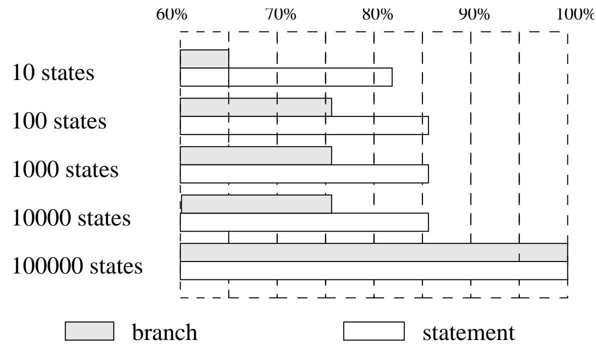

Fig. 4. Random generation coverage results

In random generation, only one test with length 10^5 states can achieve the statement and branch coverage, while in specification based testing, three tests (BR_R7_1, BR_R3, BR_R4) with a total length of about 2000 states are enough. This observation suggests that random test generation might not be able to provide the same coverage as the specification based testing, and that, in case that both achieve the same coverage, our method can lead to test sequences shorter than the random tests. A thorough comparison between specification based and random testing is still subject of many research papers and it is clearly out of the scope of this work.

As final experiment, we generate a test suite for the specification of a UPnP door controller, briefly OpenDoor, as presented in [9]. In that specification a controlled variable `Counter` is initially equal to 0 and is incremented by the rule `Close` till it reaches a `max` value (representing the number of CD slots) and then it restarts from 0. The part of the rule `Close` that causes some problem in tests generation, is

```
If open and counter = max
then          open := false
              counter :=0
```

In [9] the proposed algorithm *never discovers the reachable hyperstate where the value of counter equals max* [9], i.e. it never discovers when the guard `open`

`and counter = max` becomes true. This problem is called "non discovery problem" in that report. Note that discovering whether a state is reachable or not is undecidable, and finding a trace to cover a particular state is also undecidable, so there's no algorithm that can definitively solve the non discovery problem. However, our method is able to find a test sequence for the basic rule coverage of `Close` even for great values of `max`. Setting `max` = 100000, Spin takes 1 minute and 23 seconds[6] to generate the counter example to cover the critical guard of `Close`.

6 Related Work and Future Directions

Test generation using model checkers has been proposed in several papers. In [6] the authors present a method to generate test sequences using Spin. Their approach exploits the counter example generation of Spin, but they do not introduce coverage criteria, they require the designer to translate his/her specification in PROMELA and to insert testing goals by hand, suitably modifying the original specification. In [2] the model checker SMV is used 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 (this criterion is called *mutation adequacy*). In [7] tests are generated using model checkers (both SMV and Spin) from SCR specifications to achieve a coverage similar to the branch coverage for programs, or to cover particular system requirements.

To the best of our knowledge, the only method for generating test suites from ASM specifications is that recently developed by the Microsoft group in Redmond [9]. In order to find a test suite, they extract a finite state machine from ASM specifications and then use test generation techniques for FSMs. The problem of reducing ASMs to FSMs is undecidable. To assess the quality of the testing activity, they still need to define some coverage criteria.

The possibility to automatically encode ASM specifications in PROMELA, the automatic generation of trap properties from coverage criteria, together with the Spin automatic computation of test sequences from PROMELA specifications, allow us to develop a method to generate test sequences from ASMs in a completely automatic way. In the future, we plan to introduce some abstraction techniques that allow the extraction of models with finite domains from models with infinite domains such that some behaviors are preserved, and that reduce the likelihood of the state explosion problem (as in [3]). We are also currently working to improve our tool and to support tests generation from ASML specifications [1].

References

1. AsmL. `http://research.microsoft.com/fse/asml/`.

[6] Spin running on a PC Pentium III with 866 Mhz and 256 Mb RAM.

2. Paul Ammann, Paul Black, and William Majursk. Using model checking to generate tests from specifications. In *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, December 1998.
3. Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. In *Automated Software Engineering*, volume 6, pages 37–68. Kluwer Academic, jan 1999.
4. Giuseppe Del Castillo and Kirsten Winter. Model checking support for the ASM high-level language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 331–346, 2000.
5. P.-J. Courtois and David L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD, 1993.
6. André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *TACAS'97, Lecture Notes in Computer Science 1217*, pages 384–398. Springer, 1997.
7. 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*, sep 1999.
8. Angelo Gargantini and Elvinia Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7(11), 2001.
9. Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Conformance testing with abstract state machines. Technical Report MSR-TR-2001-97, Microsoft Reasearch, October 2001.
10. R. Hierons and J. Derrick. Special issue on specification-based testing. *Software testing, verification & reliability (STVR)*, 10(4):201–262, December 2000.
11. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
12. A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, 1999.
13. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. Springer, May 1992.
14. J. Schmid. Compiling abstract state machines to C++. *Journal of Universal Computer Science*, 7(11), 2001.
15. J. Schmid. *Introduction to AsmGofer*, 2001.
16. Gregory Tassey. The economic impacts inadequateinfrastructure software testing. Technical Report Planning Report 02-3, National Institute ofStandards and Technology, May 2002.
17. Elaine J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, 1979.
18. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit text coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.