Coverability Analysis Using Symbolic Model Checking

Gil Ratzaby, Shmuel Ur, and Yaron Wolfsthal

IBM Haifa Research Laboratory, Israel
{rgil,ur,wolfstal}@il.ibm.com

Abstract. In simulation based verification of hardware, as well as in software testing, one is faced with the challenge of maximizing coverage of testing while minimizing testing cost. To this end, sophisticated techniques are used to generate clever test cases, and equally sophisticated techniques are employed by engineers to determine the quality – a.k.a. coverage – attained by the tests. The latter activity is called Test Coverage Analysis.
While it is an essential component of the development process, test coverage can only be analyzed late in the design cycle when the tested entity and the test harness are both stable. To address this serious restriction, we introduce the notion of coverability, which intuitively refers to the degree to which a model can be covered when subjected to testing. We also show an implementation of coverability checking using Model Checking. The notion of coverability highlights a distinction between (1) whether a model has been covered by some test suite and (2) whether the model can ever be covered by any test suite. Coverability Analysis can be performed as soon as the hardware or software are written, before the test harness has been written.

1 Introduction

State machines are a simple and powerful modeling means used in a variety of areas, including hardware [VHDL93] and software design [Biezer90] [Marick95], protocol development and other applications [Hol91]. As a normal part of the modeling process, state machine models need to be analyzed and reasoned with regard to their function, performance, complexity and other properties. Traditionally, functional simulation has been a key vehicle to analyzing state machine models. The model is simulated against its expected real world stimuli and the simulated results are compared with the expected results. Simulation coverage analysis is normally applied to determine the thoroughness and quality of simulation. For example, the most common coverage metric used in the industry is *statement coverage* which checks that each statement has been executed at least once during simulation.

While test coverage analysis is an essential component of the design verification process, it can only be used late in the cycle when the code is stable and simulation environment is running. This is inherent limitation of test coverage analysis.

In this paper, we introduce the notion of coverability. Formally, a *coverability model* is defined by creating a *coverability goal* for every coverage goal in the coverage model of interest. The coverability goal is met if and only if a test that covers the corresponding coverage goal exists. Informally, coverability is a property of a state-machine model and

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 155–160, 2001.

refers to the degree to which the model can be tested using simulation. Reasoning about a model's coverability is a powerful analysis technique as it obviates the aforementioned limitation. Thus, a tool for determining coverability can help assess whether a given fragment of HDL code contains dead code, whether all branches of a particular control-flow statement can be taken, etc.

To implement coverability analysis, we build on symbolic model checking techniques, which provide a framework for reasoning about finite-state systems [McM93]. In recent years, Symbolic Model Checking, and formal verification in general, has been successfully used in the verification of communication protocols, as well as software and hardware systems [BBEL96]). Our work may be viewed as an extension of the recent research trend to bring together simulation-based verification and formal verification. Some recent works, for example, have focused on improving the quality of simulation by using formal verification methods to generate test sequences that ensure transition coverage [CFSM, LLU96].

To contrast coverability analysis and coverage analysis, consider the implementation of statement coverage. Here, a coverage tool typically implements statement coverage by adding a counter after every statement and initializing the counter to zero. Every time a test is simulated, some of the counters are modified. The coverage tool outputs all the counters that remain zero, as they are indicative of either dead-code or holes in the test plan. In coverability analysis, a rule for every statement would be automatically generated to check that it can be reached. These rules are executed by the model checker on the program (or instrumented program) and a warning on the existence of dead-code is created for every statement that cannot be reached.

Our approach is based on two key observations. First, as described above, a coverage model is composed of coverage goals, each of which is mappable to a corresponding coverability goal. The second observation is that a state-machine model can be instrumented with control variables and related transitions; these, on one hand, retain the original model behavior as reflected on the original state variables, and on the other hand can be used for coverability analysis of the model. The analysis is carried out by formulating special rules on the instrumented model and presenting these rules (with the instrumented model) to a symbolic model checker.

The rest of this paper is organized as follows: in Sect. 2 we define the terms used. In Sect. 3, we demonstrate a few coverability models and show their implementation. In Sect. 4, we explain how CAT – our Coverability Analysis Tool – was implemented. In Sect. 5, we discuss our experience using CAT and then we discuss our conclusions.

2 Definitions

A *coverage goal* is a binary function on test patterns. This function specifies whether some event occurred when the test pattern is simulated against the state-machine model. A coverage model is a set of coverage goals. The statement coverage model, for example, is a model containing a goal for every statement and indicates if this statement has been executed in simulation.

Every coverage model has a corresponding *coverability model*. A coverability model is defined by creating, for every coverage goal in the coverage model, a coverability goal

which is a binary function on the state-machine model. The coverability goal is true if there exists a test on the state-machine model for which the corresponding coverage goal is true.

3 Coverability Analysis via Model Checking

This section describes and exemplifies the concepts of instrumentation and generation of auxiliary rules in our implementation of coverability analysis. We focus on coverability models relating to values of variables and to dead code analysis, and we show sample results in figure 1.

3.1 Attainability of All the Values of a Variable

This coverability model checks whether all variable values in a code fragment are attainable. To this end, for each variable declaration: `TYPE:var`, where `var` is a variable of interest, a collection of auxiliary rules of the form !EF ($var = V_i$) – one for each value V_i of `var` – is created. The conjunction of these rules is a property requiring all relevant variables to take their respective values. This rule is presented to the underlying model checker, which in turn decides on the attainability of the value. When the formula passes, an example is also produced, demonstrating how the value is attained. Checking for this kind of coverability does not require instrumentation; it needs only the information about the variable declaration that enables the creation of the auxiliary rule.

3.2 Statement Coverability Analysis

This coverability model checks whether all statements can be reached. To this end, the program is instrumented separately for each statement S_i in the following way:

- Create an auxiliary variable V_i and initialize it to 0.
- Replace statement S_i with the statement "$V_i = 1$".

The model checker is then presented with the following rule: " !EF($V_i = 1$) , which indicates whether S_i could be reached.

Statement coverability analysis changes the program's behavior. It may now contain dead-locks and behave in an unpredictable way. However, the program's behavior remains the same until the first execution of the replaced statement. As reachability analysis is performed, the behavior of the program after the first execution of the statement is immaterial. In this type of instrumentation, only one statement at a time may be checked. We are currently working on checking more efficient implementation.

4 CAT – Our Coverability Analysis Tool

We have created a coverability tool called CAT (Coverability Analysis Tool) which is very simple to use. It receives two parameters: the name of the program to be tested and the coverability models to be used. CAT outputs a list of all the coverability tasks,

```
module example (A,B,clock,F);
 input A,B;
 input [0:7] clock;
 output F;
 wire W1,W2;
 reg reg1;
 reg [0:7] arr;
 integer int1,int2;
 initial
    begin
        int2 = 1;
        int1 = 0;
        arr[0:7] = 0;
        reg1 = 0;
     end
 assign W2 = ! clock[6];
 assign W1 = reg1;
 always @(clock)
 begin
20int2 = 2;
21if(clock[3]>clock[2] || W2==1)
22 for(int1=0;int1<5;int1=int1+1)
23  begin
24   int2 = int2 + 1;
25   if(int1 > int2)
26      arr[4:5] = 2;
27   if(int1 < int2)
28      arr[4:5] = 3;
29  end
30 else
31  if(W2 == 1)
32     reg1 = 1;
33end
endmodule
```

Attainability Results:

```
Wire W1:
 Value 0 is attainable.
 Value 1 cannot be attained!
Wire W2:
 Value 0 is attainable.
 Value 1 is attainable.
```

Statement Coverability Results:

```
ASSIGNMENT in line 20: ok
IF in line 21:  ok
ASSIGNMENT in line 24: ok
IF in line 25:  ok
ASSIGNMENT in line 26:
   -- can never execute!
IF in line 27:  ok
ASSIGNMENT in line 28: ok
IF in line 31:  ok
ASSIGNMENT in line 32:
   -- can never execute!
```

Fig. 1. Verilog program with analysis

indicating whether each task is coverable or not. CAT works in the following way: for every coverability goal, CAT instruments the original program with the needed auxiliary statements, and creates a corresponding temporal rule. The rule is then checked by using a model checker on the instrumented program and the result of the run is reported. For example, if we want to find whether a line can be reached, we add an instrumentation that marks this line so it can be referred to by the rule. The model checker then checks the attainability of the marked line and CAT extracts and reports the answer.

CAT uses RuleBase, a symbolic Model Checker developed by the IBM Haifa Research Lab ([BBEL96]), as its underlying engine. RuleBase can analyze models formulated in several hardware description languages, including VHDL and Verilog. The basis for CAT is RuleBases's Verilog parser, Koala. CAT parses the input Verilog design, extracts the information needed in the current coverability goal, and constructs the auxiliary CTL rules on demand. CAT then transforms the design so that it will include the relevant auxiliary statements, and presents the instrumented program to RuleBase.

CAT supports default free-behavior environments, as well as user-defined environments. The user may choose between the two modes of environment modeling – default

or user-defined. For example, in the application of CAT for dead-code analysis, default free-behavior environments are used. If a statement cannot be covered with free inputs, it can never be reached under any circumstances.

5 Experience

The coverability report for the sample program of Fig. 1 was generated in less than one minute. Figure 1 shows a Verilog program and Fig. 2 shows the issued report that indicates which variable values are not attainable and which statements cannot be reached. Working on benchmark PCI local BUS ([AZ97]), statement-reachability rules for a 1500-line Verilog program were evaluated at the rate of one rule per minute. To test CAT's industrial applicability, we used CAT to analyze the coverability of some customer code. It took two hours to complete a dead-code analysis report for a 1200-line Verilog module of a relatively simple control structure. The rules looking for dead code (e.g., "line 312 is never reached") execute quickly since each rule induces a relatively small cone-of-influence, which is amenable to the many reductions supported by RuleBase, our underlying model checker. Working on another Verilog file of comparable size (1300 lines) but with a significantly more complex control structure, a report was created at a rate of ten statement coverability rules per hour. This appears to be excessive for the developers, and we therefore are currently evaluating some of the optimization techniques.

6 Discussion

In this paper we introduced the concept of coverability analysis and described how a number of coverability metrics, corresponding to some commonly-used coverage metrics, can be implemented via Symbolic Model Checking. The same ideas can be used to implement many other coverability metrics (e.g., define-use, mutation, and loop [Marick95][Kaner95]).

The presented technique integrates ideas from traditional simulation and formal verification. It is somewhat similar to ideas seen in fault grading [KPKR94]. In fact, the technique derives its strength from its use of the exhaustiveness of formal verification to improve the planning of simulation.

A possible critique of the coverability approach might be that if a state machine is small enough to measure its coverability, it would also be small enough to be formally verified. However, we believe the merit of the coverability approach does in fact complement – and extend – the capabilities of formal verification due to the following reasons:

1. Today's model checkers are complicated tools that require the users to write complex rules in some form of temporal logic. The application suggested in this paper is fully automatic and enables naive users to take advantage of the power of model checkers with a "one button" interface for coverability analysis. We believe that adding this capability to existing model checkers will increase their utilization and applicability in the hardware design community.

2. Our approach can be used in the debugging stage (as soon as code is being written), before formal verification or simulation are used.
3. Formal verification does not guarantee that the design is correct. Even units that were formally verified need to be tested via simulation in which coverability can help.

References

[AZ97] Adnan Aziz, Example of Hardware Verification Using VIS, The benchmark PCI local BUS, `http://www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/`.

[BBEL96] I. Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an Industry-Oriented Formal Verification Tool", Proc. DAC'96, pp. 655–660.

[Beizer90] Boris Beizer, Software Testing Technique, 2/e. New York: Van Nostrand Reinhold, 1990.

[CGP99] E.M. Clarke, O. Grumberg. D.A. Peled. Model Checking, MIT Press, 1999.

[CFSM] D. Geist, M. Farkas, A. Landver, Y. Licthenstein, S. Ur and Y Wolfsthal, Coverage Directed Test Generation using Symbolic Techniques, FMCAD 96: Int. Conf. on Formal Methods in Computer-Aided Design, November 1996.

[Hol91] G. J. Holtzman, Design and Validation of Computer Protocols, Prentice Hall, 1991.

[Kaner95] C. Kaner, Software Negligence & Testing Coverage, Software QA Quarterly, Vol 2, #2, pp 18, 1995.

[KN96] M. Kantrowitz, L. M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor", Proc. DAC'96.

[KPKR94] S. Kajihara, I. Pomerantz, K. Kinoshita and S. M. Reddy, "Cost Effective Generation of Minimal Test Sets for Stack-At Faults in Combinatorial logic Circuits", 30th ACM/IEEE DAC, pp. 102-106, 1993.

[LLU96] D. Levin, D. Lorentz and S. Ur, "A Methodology for Processor Implementation Verification", FMCAD 96: Int. Conf. on Formal Methods in Computer-Aided Design, November 1996.

[Marick95] B. Marick, The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing, Prentice-Hall, 1995.

[McM93] K. L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[Mil90] Raymond E. Miller, Protocol Verification: The first ten years, the next ten years; some personal observations, in Protocol specification, Testing, and Verification X, 1990.

[RB] RuleBase User Manual V1.0, IBM Haifa Research Laboratory, 1996.

[TCE] I. Beer, M. Dvir, B. Kozitsa. Y. Lichtenstein, S. Mach, W.J. Nee, E. Rappaport. Q. Schmierer, Y. Zandman, VHDL TEST COVERAGE in BDLS/AUSSIM Environment, IBM HRL Technical Report 88.342, December 1993.

[VHDL93] D. L. Perry, VHDL Second Edition, McGraw-Hill Series on Computer Engineering, 1993.

[Weyuker94] E. Weyuker, T. Goradia and A. Singh, Automatically Generating Test Data from a Boolean Specification, IEEE Transaction on Software Engineering, Vol 20, No 5 May 1994.