

Improvements in Coverability Analysis

Gil Ratsaby, Baruch Sterin, Shmuel Ur
{rgil, baruch, ur}@il.ibm.com

IBM Haifa Research Labs, Israel

Abstract. In simulation-based verification users are faced with the challenge of maximizing test coverage while minimizing testing costs. Sophisticated techniques are used to generate clever test cases and to determine the quality attained by the tests. The latter activity, which is essential for locating areas of the design that need to have more tests, is called *test coverage analysis*.

We have previously introduced the notion of coverability, which refers to the degree to which a model can be covered when subjected to testing. We showed how a coverability analyzer enables naive users to take advantage of the power of symbolic model checking with a 'one-button' interface for coverability analysis. In this work, we present several heuristics, based on static program analysis and on simulation of counter examples, for improving the efficiency of coverability analysis by symbolic model checking. We explain each heuristic independently and suggest a way to combine them. We present an experiment that shows improvements based on using random simulation in the analysis of coverability.

1 Introduction

We introduced the notion of coverability in [6]. Informally, coverability is a property of a State-machine model that refers to the degree to which the model can be tested using simulation. 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. Thus, a tool for determining coverability can help assess whether a given fragment of HDL code contains dead-code or whether all branches of a particular control-flow statement can be taken. A coverability analyzer is a tool that enables naive users to take advantage of the power of model checkers with a 'one-button' interface for coverability analysis.

To compare coverability analysis and coverage analysis, consider the implementation of statement coverage. In statement coverage, the coverage model is composed of a goal for each statement which can be satisfied by a test whose control passes through that statement. 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. While it is often reasonable to leave dead-code in the model to ease the testing of future modifications, this

dead-code should not be considered 'uncovered' in the coverage analysis. The concept of coverability captures this point.

We implemented coverability analysis by building on symbolic model checking techniques which provide a framework for reasoning about Finite-state systems [20]. The implementation is based on two key observations. The first is that 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. On one hand, they retain the original model behavior as reflected on the original state variables and, on the other hand, they 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. Then, the symbolic model checker either verifies that the task is coverable and presents a proof in the form of a counter example, or states that the task is uncoverable and a bug is found.

Coverability analyzers are computationally intensive tools that automatically execute a large number of rules. In this paper, we show a number of techniques that reduce the number of rules that need to be run, as well as improve the efficiency of each rule as it is executed. One idea is based on the observation that a test that was generated to cover one coverability goal may cover other goals as well. There are a number of techniques for determining which other tasks are covered; some techniques are based on static analysis and some on the simulation of counter examples. If one takes into account that covering one task may cause others to be covered, the order chosen to cover the tasks is of importance. The second idea, which has applications for other domains in model checking, is that given a rule R and a program P , it is sometimes possible to create a simpler program P' on which R yields the same reply.

Unlike recent coverage work for the Finite State-machine (FSM) [citeFSMCover](#), where coverage is calculated for formal rules on the FSM and is measured on the FSM (e.g., percentage of the FSM states covered), our measurement is on the program itself. Our tool's feedback is expressed in a language that is natural to the developer, with clear action items based on the tool's outputs.

The rest of this paper is organized as follows: in Section 2, we define the common terms used throughout the paper. In Section 3, we describe the context in which our work was created. In Section 4, we show how coverability can be implemented for several coverability models. In Section 5, the main section, we show a number of algorithms and a general methodology used to improve the efficiency of coverability analysis. In Section 6, we explain how CAT — our Coverability Analysis Tool — was implemented. In Section 7, we discuss our experience using CAT and present our conclusions in Section 8.

2 Definitions

An FSM is an abstract model of a system where, at any given time, the outputs are a function of the inputs and values of the state variables at that time. The relationship between the inputs of the FSM, present state (represented as a vector of state values),

and next state is described by a transition relation. Hereafter, we use the terms *FSM*, *State-machine model*, and *program* interchangeably.

A *symbolic model checker* [20] is an algorithm that takes a State-machine model and a set of temporal logic properties (rules) as input. It computes the truth or falsity of the property by traversal of the State-machine. Whenever the property is not satisfied by the model, the model checker produces a counter-example, namely, a sample execution sequence of the model where the property is violated.

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 goals. The statement coverage model, for example, is a model that contains a goal for every statement and indicates whether 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.

A *basic block* is a sequence of non-branching statements. Basic block A *dominates* basic block B ($Dom(A, B)$) if, whenever the control goes through basic block B, it *must* go through A. Basic block A *predominates* basic block B ($Pre(A, B)$) if, before the control gets to B, it *must* go through A. Basic block A is reachable from basic block B ($Reach(B, A)$), if there is a path from B to A. Calculating dominance, predominance, and reachability is fast. For more information on basic blocks, dominating blocks, and their uses in compilation, see[1].

In the next section, we describe how to efficiently implement coverability analysis in a model checker, such that the implementation is invisible to the user.

3 Background

State-machines are simple yet powerful modeling techniques used in a variety of areas, including hardware [29] and software design [5] [19], protocol development, and other applications citeHol91. As a normal part of the modeling process, State-machine models need to be analyzed with regard to their function, performance, complexity, and other properties.

Formal verification techniques, especially symbolic model checking, are a natural means for reasoning about State-machines. However, most tools available today require specialized training, not available to all designers. Further, due to the state explosion problem, the tools are limited to relatively small designs or design components. As a result, functional simulation is the most commonly used technique for verification. In simulation-based verification, the model is simulated against its expected real world stimuli and the simulated results are compared with the expected results. Simulation, by its very nature, cannot and does not guarantee complete exhaustive coverage of all execution sequences. Furthermore, some parts of the model code, such as error handling, may be inherently hard to cover. Simulation and simulation coverage analysis also suffers from several significant limitations:

- Analysis can only start after ‘system integration’, when the simulation environment can receive stimuli. This occurs fairly late in the development process.
- Simulation requires a process of test pattern generation (i.e., test cases or vectors), which is often costly, as it involves the creation of a suitable test-bench.
- Simulation Coverage Analysis is, by definition, an analysis of the test suite, rather than of the model under investigation. Therefore, it is essentially limited in its ability to provide deep insight into the model.

In recent years, symbolic model checking and formal verification have been successfully used in the verification of communication protocols, as well as software and hardware systems [3]. 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 [9, 18]. Some researchers [22] have used formal methods to specify simulation targets at the micro-architecture level, while others have used formal verification as part of the simulation process [10]. Indeed, with the ever-increasing complexity and diversity of systems under development, we view the paradigm of integrating formal methods with simulation as a powerful and practical approach.

4 Coverability Analysis via Model Checking

This section uses two simple examples to describe how coverability can be implemented. We focus on two types of coverability models: one that relates to the values of variables, which only requires auxiliary rules, and one that relates to dead-code analysis, which requires code instrumentation (the insertion of some additional code) in addition to auxiliary rules.

4.1 Attainability of all the Values of a Variable

This coverability model checks whether all variable values, in binary or enumerated types, in a code fragment are attainable. For each variable declaration, we automatically create a collection of auxiliary rules of the form $\neg \text{EF} (var = V_i)$ — one for each value V_i of var — which specifies that V_i cannot be attained by var .

The conjunction of these rules is a property that requires all possible values of var to be attainable by var . This rule is presented to the underlying model checker, which, in turn, decides on the attainability of these values. If the formula passes, an example is also produced, which demonstrates how the value is attained. Checking for this kind of coverability does not require code instrumentation; the tool only needs the information about the variable declaration that enables the auxiliary rule to be created.

4.2 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 manner:

- Create an auxiliary variable V_i and initialize it to 0.
- Replace statement S_i with $\{V_i = 1; S_i\}$.

The model checker is then presented with the following rule: $\neg \text{EF}(V_i = 1)$, which indicates whether S_i can be reached.

5 Improvements in Coverability

5.1 Optimizing Coverability Analysis by Reducing the Rules to be Checked

When performing coverability analysis, there are a large number of temporal rules that have to be checked. In our previous implementation [6], we created a list of these rules and presented each of them, in turn, to the model checker. Since running a rule in a symbolic model checker is a time consuming task, we looked for ways to reduce the number of rules that have to be checked, based on static analysis and on simulation of counter examples.

Using Inflation: The following simple optimization is natural; start by creating a list of coverability tasks. Randomly choose one of the coverability tasks from the list, create a corresponding rule, and present it to the model checker. The model checker yields a counter example if the task, as expected, is reachable. If the task is reachable, it can be removed from the list. Then, the counter example can be examined to see if it contains evidence that can be used to remove additional tasks. If it contains such evidence, the examination is rewarded since running a rule is more time-consuming than examining the counter example, and the total execution time is reduced.

Currently, symbolic model checking performs many optimizations to improve efficiency. One of these optimizations, called *cone of influence*, retains those variables referred to in the rule, as well as any variable that affect them. After a rule is run, the counter example only contains values for the variables of the reduced model. This causes the examination of the counter example to yield very few additional tasks.

To overcome this problem, we use *inflation*. Inflation is a method for adding variables to a trace created from the counter example, so that it contains legal values for all the variables in the design. This is accomplished by using simulation and randomly selecting values for any variables not contained in the trace and not deterministically set by the program. For example, the value of the model inputs not contained in the trace, and of random or non-deterministic operations, is selected at random. The inflator can be used to create such a trace by itself or to expand a counter example. A random choice is consistent with the counter example since all variables in the counter examples were chosen using the cone of influence reduction. Therefore, variables outside that cone cannot influence them. Such an inflator is part of the RuleBase package [3][26].

We can improve the simple idea of observing the counter example by using the inflator. The counter example is inflated, using a set of variables mentioned in the coverability tasks. All the coverability tasks that are covered by the inflated trace are removed from the list. This reduces the number of rules and the execution time, since running the inflator is faster than checking a rule.

In our implementation, we have two types of rules: rules that check if some auxiliary Boolean variables reached the value of 1 (related to statement coverability and multi-condition coverability) and rules that check that each model variable received all of its possible values (a rule for each variable-value combination). We instrument the program only at basic blocks where we have not yet ascertained if the blocks are reachable (a decreasing number). After receiving a counter example, we use the inflator to expand the trace with all the auxiliary variables and all the variables for which some value has not yet been achieved. For each auxiliary variable, we check whether it attained the value 1. For each variable, we check whether it reached new values. After we remove all the coverability tasks that have been attained through inflation, we run the next rule. Before running a rule, we instrument the program at all the locations that have not yet been reached. As this list shrinks, the execution speed increases and the inflator is used on a decreasing number of variables. Although many auxiliary variables are added in the instrumentation, they hardly affect the speed of execution, since all but one are removed by the cone of influence heuristics. The algorithm, expanded with some steps from the next subsection, is described in the left side of Figure 2.

The inflator chooses random values for the model inputs and for non-deterministic operations. Therefore, it is possible that repeated runs of the inflator would yield different inflated traces for the same counter example that contains additional coverability tasks. A possible strategy for deciding how to run the inflator is to re-run the inflator and, after each run, to randomly (e.g., at 0.5 probability) decide whether to continue re-running the inflator or move to the next rule. A more intelligent strategy would be to re-run the inflator as long as the total time of the re-runs is less than the average time it took to run the model checker. This way we have the chance of covering many coverability tasks in the time it would take to model-check one coverability task. In addition, we can modify the inflator to receive a list of variables and the values we would like to cover, and to bias its random choice toward achieving the supplied values.

There are a number of additional ways in which the inflator can be used to improve performance. The first, which is simple to implement, but very useful, is to create a long random trace for the program and examine it to remove the tasks seen in it. In this case, we use only the random simulation capabilities of the inflator. The advantage is that after a few seconds of execution, many of the tasks are eliminated. The number of the tasks eliminated is larger than initially expected due to the highly parallel nature of the HDL code. The second way is to extend every trace created for a counter example by a number of cycles. The idea is that there may be other tasks ‘hiding’ next to the one we found and we want to increase the chances of finding them. This is based on the coupling effect [23].

Static Reduction of the Number of Rules: Static analysis can show the relationship between coverage tasks. For example, dominating block analysis shows which additional blocks are always executed if a block is executed. This information may be presented in a list in which each entry contains a block and the block that are always executed whenever the former block is executed. This list may be used to find a small set of blocks that dominate the entire program. Therefore, instead of directly covering the entire program, we only have to cover this small set of blocks.

Techniques for solving the Set Cover problem [12] can then be used to find a subset of tasks, which if covered, imply that all the tasks are covered.

For example, a list for the program on the left side of Figure 1 will appear as follows:

$$\begin{aligned} & \{ A, \{A, B, F\} \} \\ & \{ B, \{A, B, F\} \} \\ & \{ C, \{A, B, C, E, F\} \} \\ & \{ D, \{A, B, D, F\} \} \\ & \{ E, \{A, B, C, E, F\} \} \\ & \{ F, \{A, B, F\} \} \\ & \{ G, \{A, B, C, E, F, G\} \} \end{aligned}$$

From the list we can see that if we created a task that covers C we can remove $A, B, E,$ and F from the list of blocks to cover. It can be seen that D and G are necessary and sufficient to cover everything. If this analysis was not used, it is possible we would have tried to cover a random block in the beginning and would have ended up using three tests. For example, the tests used could have tested for D passing through A, B, D, F and then for E passing through A, B, C, E, B, D, F and finally for G passing through A, B, C, E, G, B, D, F .

A number of papers have been published on the similar problem of modeling test compression as the Set Cover Problem [14]. In general, the problem of finding a minimal set cover is NP-Complete [12] Finding a good solution turns out to be easy.

If this technique is used, even without inflation, it is possible to use coverability only on a subset of the tasks and thus reduce the performance requirements.

5.2 Faster Execution of Rules

Given a program A and a question (rule) Q on program A , we create an auxiliary program A' such that the question Q on A' yields the same reply as the question Q on A . If A' is simpler than A , Q may execute faster on A' .

Other than the guarantee that Q will have the same reply on A and A' , we make no other claims on A' . For example, it is possible that A' will not terminate and that a trace on A' is not expandable to a trace on A . A similar idea was used in [18], where A' was constructed for specific rules that have to do with tuning tests on pipelines.

We created an auxiliary program based on this idea to improve statement coverability analysis. When checking if basic block B is reachable in program A , program A' is created by removing all basic blocks X from A such that either X is in the set $Pre(B, X)$ or X is in $!Reach(X, B)$. Then, we can check if B is reachable in A' . In many cases, A' is much simpler than A and we expect the rule to run faster. Furthermore, this heuristic is orthogonal to the more common cone of influence heuristics and provides different and disjoint improvements.

We can remove all X , such that X is in the set $Pre(B, X)$ (including B itself) because we know before the control reaches X it will go through B . Since we only care if the control can reach B , we know that the control can reach B in A' if and only if it can reach B in A . If there is no way to get to B (X in $!Reach(X, B)$) from X , then we do not care what is done in X . Since the control will never reach B after it reaches

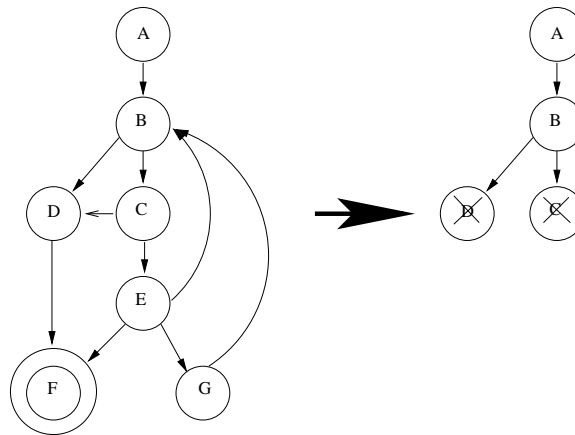


Fig. 1. Creating an Auxiliary Program

X , X can be removed. When a block is removed, its entire content can be removed. If a block that was not removed points to a removed block in the control flow diagram, a stub (empty block) needs to be maintained for consistency.

An example of creating an auxiliary program can be seen in Figure [reffig:ShortProgram](#). The original program on the left is composed of seven basic blocks, each containing any number of statements. The execution of the program can be complex and terminates once the control reaches block F . If the question is ‘Is basic block C reachable?’, we can reduce the program to a much simpler program. Basic blocks E and G can be eliminated because, in order to reach them, the control has to pass through C first; stated differently, they belong to $Pre(C, X)$. Blocks D and F may be removed since if the control reaches them it will not reach block C . The content of block C can be removed, because we only care whether it was reached, and, it can become an empty block. We have to keep an empty block for block D to maintain a legal program, since it is pointed to by some part of the program. The auxiliary program does nothing and probably terminates abnormally; however, the reply to the rule ‘block C can be reached’ is the same for the original and the auxiliary program. The rule will almost certainly run faster on the auxiliary program because it is much simpler.

Similar optimizations may be used when we want to check if variable y can attain the value y_0 . Assume there are n locations in which the assignment happens (i.e., it can happen in these specific locations and cannot happen in any other location). In essence, we want to check if we can reach one of a set of basic blocks. When checking if a group B of basic blocks is reachable in program A , program A' is created by removing all basic blocks X from A such that either X is in $Pre(b, X)$ (b in B) or X is in $!Reach(X, b)$ for all b in B . Then we can check if B is reachable in A' .

Similar reasoning to that of the first application shows the correctness of this reduction.

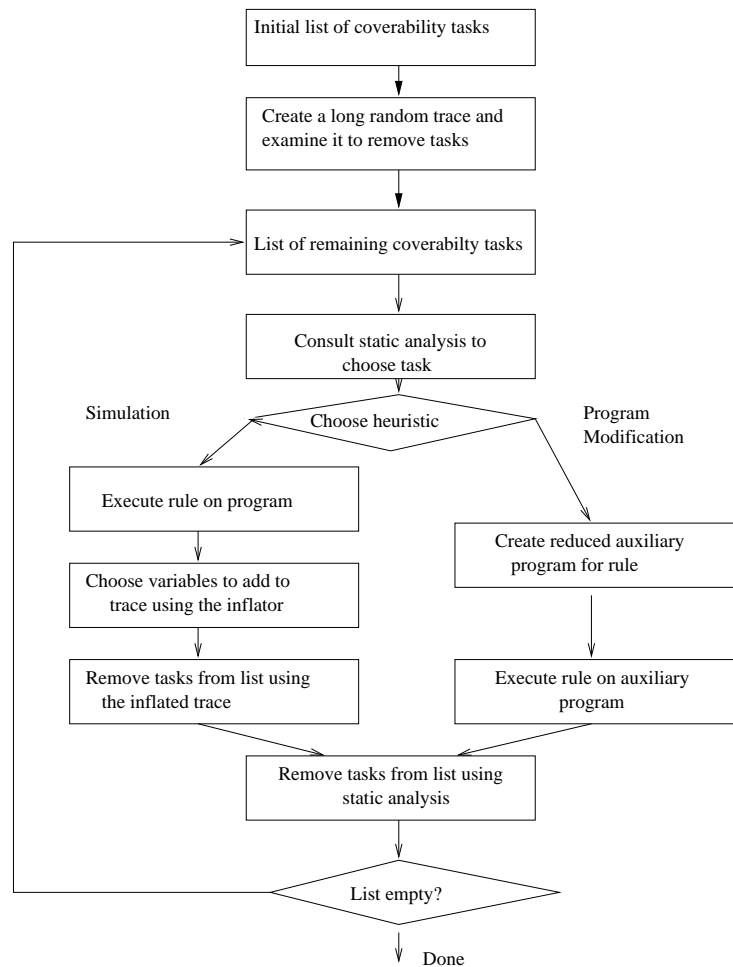


Fig. 2. Combining Heuristics

The power of this idea is not limited to rules that originate in coverability. We found a number of other general cases in which such reduction can be applied; however, these cases are beyond the scope of this paper.

5.3 Combining Heuristics

We presented heuristics based on three distinct ideas. First, we explained how simulation may be used. Initially, random simulation can be used to remove many tasks. Then, given a counter example, we can try to expand it to find similar tasks that are coverable. Next, we show that for each coverability task, we can statically evaluate other tasks, such that if that task is covered, these tasks will also be covered. We showed that we can choose a subset of the tasks such that if they are coverable, all the tasks are cover-

able. In addition, we presented a method for more rapidly calculating whether a task is coverable. All of these heuristics can be combined into a single heuristic. First, a list of tasks is calculated. Then, random simulation is used to remove many of the tasks. Next, we reach a loop that we follow until all tasks are evaluated. In the loop we choose a task according to static analysis. Next, we either execute it faster using the third heuristic or use the inflator to inflate the trace. The two options are not compatible since, if we modify the program, we cannot correctly evaluate the trace for the other tasks. The combined algorithm is shown in Figure 2.

6 CAT — Coverability Analysis Tool

We improved our Coverability Analysis Tool (CAT), which is very simple to use, with some of the optimizations presented here. CAT 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, indicating whether each task is coverable. 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 using a model checker on the instrumented program and the result of the run is reported. For example, if CAT wants to find whether a line can be reached, it adds an instrumentation that marks this line so that it can be referred to by the rule. CAT activates the model checker that checks the attainability of the marked line and extracts and reports the answer.

6.1 Using CAT

To use CAT, the user invokes CAT with the source file of the Verilog design and the name of the design's top module. CAT parses the design, extracts any information needed to create formal rules, and instruments the program to its needs. CAT has two modes of operation: GUI mode and command-line mode. The command-line mode is the simplest, where the user activates CAT and waits for the results. By default, CAT performs all three types of coverability analyses, and creates unique-name report files with the results. CAT also has a graphic interface, which allows the user to choose the kind of coverability analysis to be performed and the parts of the design on which to perform them.

The user may also choose to create an environment for the design, view the details of the design details based on the CAT parser, and activate a heuristic run of the coverability analysis. The left side of Figure 3 shows the CAT reachability screen, which presents the statements that are potentially reachable. The user can change the list of statements to be checked using the Custom option, and may choose between running a regular full check, or a heuristic check — which can run faster.

Figure 4 presents the results of running the heuristic reachability analysis and shows which of the statements are reachable and which are not. It also reports the running time, the number of full runs needed, and the number of rules found using simulation. It reports that two lines, line 21 and line 30, are not coverable. Line 21 is not coverable since *int1* and *int2* are updated at every cycle and are never equal to each other. Line

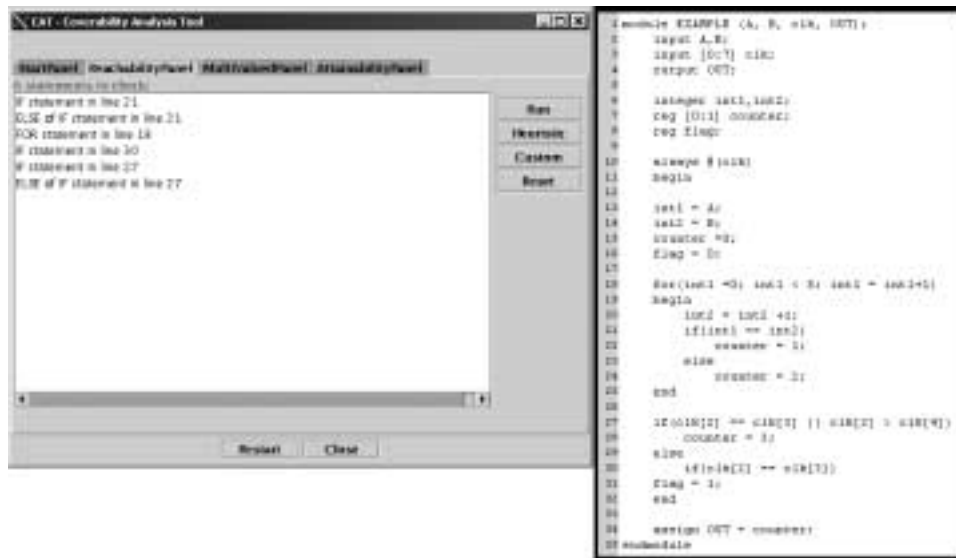


Fig. 3. Source Code and Reachability Analysis Screen

30 demonstrates a typical error; writing *or* in line 27 instead of *and* causes line 30 to be unreachable.

Figure 5 shows the attainability analysis report. For each variable the user may check which values are attainable and which are not.

6.2 Supporting Software

CAT uses RuleBase, a symbolic model checker developed by the IBM Haifa Research Labs ([3]), as its underlying engine. RuleBase can analyze models formulated in several hardware description languages, including VHDL and Verilog. The basis for CAT is Koala, the RuleBase Verilog parser. CAT parses the input Verilog design, extracts the information needed in the current coverability goal, and constructs the auxiliary SUGAR citeSUGAR rules on demand. CAT then transforms the design so that it includes the relevant auxiliary statements and presents the instrumented program to RuleBase.

6.3 Environment Modeling

CAT supports default non-deterministic behavior environments, as well as user-defined environments. The user can choose between the two modes of environment modeling — default or user-defined. For example, default non-deterministic environments are used in the application of CAT for dead-code analysis. If a statement cannot be covered with free inputs, it cannot be reached under any circumstances.



Fig. 4. Results of Reachability with Simulation on Counter Examples

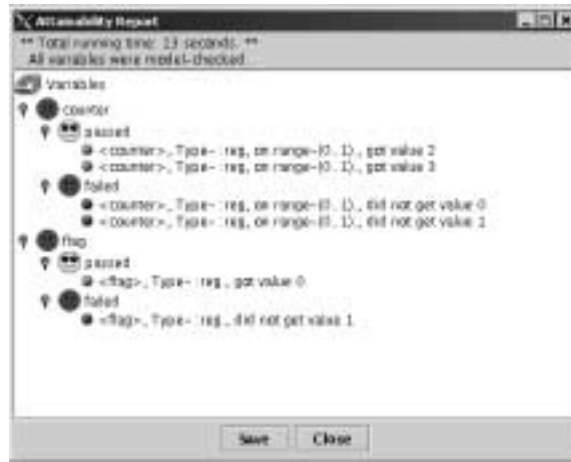


Fig. 5. Results of Attainability Analysis

7 Experiments

We ran coverability analysis on several designs, ranging from trivial examples to large complex designs consisting of thousands of lines of Verilog code. All the tests were run with a non-deterministic environment, covering all possible behaviors of the inputs.

Designs were collected from several sources: [2], internal IBM, and a customer supplied design. All the tests were executed on a 375 MHz PowerPC 604e machine with 1GB of RAM. Table 1 provides a summary of our results. Each row contains parameters that describe a design and the experimental results for that design. The design is described using its name, number of lines, number of reachability tasks, and number of flip flops. For each design, we measured the run time without any heuristics. We measured the total run time when the simulator is used to inflate the counter examples which are then inspected to remove tasks. In the next column, we show the number of

statements that were removed using that inspection. Next, we present the total time if random simulation is used to remove statements before the FSM is model checked. The last column details the number of tasks removed when simulation is used first.

As seen in Table 1, coverability analysis on the smaller designs is usually very fast; CAT was able to perform a full analysis in a matter of minutes. Medium-size designs were processed in less than an hour, which suggests that CAT can be used early in the development process of such modules to efficiently find some initial design problems. The largest designs we tried (such as `ac97_ctl`), were too much of a challenge for the model checker, and made it impossible to process even the simplest queries.

All designs benefited from running the simulation first and then inflating the tests. As designs become larger, the use of heuristics to improve the performance of coverability analysis becomes a must; without them, the analysis is either too slow or does not terminate. For example, for `DLX`, running the simulation made it possible to complete the coverability analysis. After simulation, only four tasks remained which were model checked. The model checker alone could not handle the entire design.

Simulation works so well is because of the parallel nature of HDL code, where many statements execute in parallel, leading even the shortest simulation to find many reachable statements. Inflation of counter traces works well, even after simulation was used to remove the tasks, because of the coupling effect.

Design	Lines	Statements	FF	Regular	Heuristic	Statements Removed	Simulation First	Tasks Removed
Example	35	6	5	34sec	21 sec	3	15 sec	4
MEM	50	5	54	26sec	13 sec	2	0.5 sec	5
VEND	100	14	16	69 sec	41 sec	6	0.5 sec	14
PG_FIFO	1200	74	816	42 Min	18 Min	57	20 Min	43
PCI(norm)	1700	264	646	62 Min	52 Min	103		102
DLX	1700	57	2274	N/A	N/A	N/A	181 sec	54
Prime Arbiter	1365	62	173	517 Min	15 Min	57	11 Min	55

Table 1. Performance with and without heuristics

8 Conclusions

In [6], we introduced the concept of coverability analysis and described how a number of coverability metrics, which correspond to some commonly-used coverage metrics, can be implemented via symbolic model checking. The same ideas can be used to implement many additional coverability metrics (e.g., define-use, mutation, and loop [19][15]).

Tools that measure coverability have a strong appeal; they are very simple to use, the reports generated are easy to understand without any training, and the tool can be used as soon as the code is written — when verification by simulation is not yet an

option. Further, the reports have the additional benefit that every bug reported is a real bug with no false alarms. However, the naive implementation proved slow.

We have therefore started to work on heuristics that improve the implementation of coverability analysis. We recognize a number of avenues that may be pursued in order to improve performance:

1. Observing the counter examples. We can put all the coverability tasks into a pool of tasks. Whenever a counter example is created to demonstrate that a task is coverable, we can check if there are additional tasks, not yet covered, that are covered by that counter example. If we find such tasks we remove them from the pool.
2. Using random simulation. Simulation is fast and can be used to generate traces of possible execution of the program. However, unlike functional coverage, simulation cannot be efficiently directed to cover specific tasks. Before we begin to look for specific tasks, we can run simulation on the program a few times and remove all the observed tasks. In addition, simulation may be used to expand any counter example to contain all the values, referred to in the remaining coverability tasks, to improve the likelihood of detecting and removing these tasks.
3. Using static analysis. For each task we can identify a list (which may be empty) of additional tasks, such that, if this task is covered, all the additional tasks are covered. Using algorithms for set cover [7], we can select a subset of the tasks and try to cover only these tasks, thereby reducing the number of the tasks that need to be covered.
4. Using program transformations. The rules used for coverability analysis, especially statement coverability analysis lend themselves to optimizations based on modifying the program. We can create an auxiliary program that is simpler, in which running the rule on the auxiliary program yields the same results.

We have already implemented item one and two of the above heuristics. The results are very promising and it seems that coverability can be measured on modules that are at the size limit used for model checking, or that have slightly surpassed it. One reason is that many of the tasks are eliminated using fast simulation. Another is that many of the reductions used in model checking engines apply to the simple rules used in coverability analysis.

In the future, we plan to implement and test the remaining heuristics, see how to efficiently combine all the heuristics, and most importantly see how many bugs we can find in real, designer written code.

References

1. A.V. Aho, R. Sethi, J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. Adnan Aziz. Example of Hardware Verification Using VIS, The benchmark PCI Local BUS, URL <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/>.
3. I. Beer, S. Ben-David, C. Eisner, A. Landver. RuleBase: an Industry-Oriented Formal Verification Tool. Proc. DAC'96, pp. 655–660.
4. I. Beer, M. Yoeli, S. Ben-David, D. Geist and R. Gewirtzman. Methodology and System for Practical Formal Verification of Reactive Systems. CAV94, LNCS818, pp 182-193.

5. Boris Beizer. *Software Testing Technique*. New York: Van Nostrand Reinhold, second edition, 1990.
6. G Ratzaby, S. Ur and Y. Wolfsthal. *Coverability Analysis Using Symbolic Model Checking*. CHARME2001, September, 2001.
7. E. Buchink and S. Ur. *Compacting Regression Suites On-The-Fly*. Joint Asia Pacific Software Engineering Conference and International Computer Science Conference, Hong Kong, December, 1997.
8. E.M. Clarke, O. Grumberg. D.A. Peled. *Model Checking*, MIT Press, 1999.
9. D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur and Y Wolfsthal. *Coverage-directed test generation using symbolic techniques*. In Proc. Int. Conf. Formal methods in Computer-Aided Design, pages 143158, 1996.
10. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. *FoCs - Automatic Generation of Simulation Checkers from Formal Specifications*. In Proc. 12th International Conference on Computer Aided Verification (CAV), 2000.
11. Y. Hoskote, T. Kam , P. Ho and X. Zhao. *Coverage Estimation for Symbolic Model Checking*. DAC'99, pp300-305, June 1999.
12. M.R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
13. G. J. Holtzman. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
14. D.S. Hochbaum. *An Optimal Test Compression Procedure for Combinational Circuits*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 15:10, 1294-1299, 1996.
15. C. Kaner. *Software Negligence & Testing Coverage*. Software QA Quarterly, Vol 2, #2, pp 18, 1995.
16. 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.
17. 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.
18. 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.
19. B. Marick. *The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1995.
20. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
21. Raymond E. Miller. *Protocol Verification: The first ten years, the next ten years; some personal observations*. In Protocol specification, Testing, and Verification X, 1990.
22. Y.V. Hoskote, D. Moundanos and J.A. Abraham. *Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors*. IEEE International Conference on Computer Design (ICCD '95), October 1995.
23. A. J. Offutt. *Investigation of the software testing coupling effect*. ACM Transactions on Software Engineering Methodology, 1(1):3-18, January 1992.
24. F. Orava. *Formal Semantics of SDL Specifications*. In Protocol Specification, Testing, and Verification VIII, 1988.
25. *RuleBase User Manual V1.0*, IBM Haifa Research Laboratory, 1996.
26. I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. *RuleBase: Model Checking at IBM*. Proc. 9th International Conference on Computer Aided Verification (CAV), 1997.
27. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. *The Temporal Logic Sugar*. Proc. 13th International Conference on Computer Aided Verification (CAV), 2001.

28. 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.
29. D. L. Perry. VHDL Second Edition. McGraw-Hill Series on Computer Engineering, 1993.
30. Telecordia Software Visualization and Analysis Tool. URL <http://xsuds.argreenhouse.com/>.
31. 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.