Smart Simulation
Using Collaborative Formal and Simulation Engines

Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula,
Robert Damiano, Valeria Bertacco, Jerry Taylor, Jiang Long

{pho, shiple, kevinh, kukula, robertd, valeria, jerryt, long}@synopsys.com

Advanced Technology Group, Synopsys Inc.

Abstract

We present *Ketchum*, a tool that was developed to improve the productivity of simulation-based functional verification by providing two capabilities: (1) *automatic test generation* and (2) *unreachability analysis*. Given a set of "interesting" signals in the design under test (*DUT*), automatic test generation creates input stimuli that drive the DUT through as many different combinations (called *coverage states*) of these signals as possible to thoroughly exercise the DUT. Unreachability analysis identifies as many unreachable coverage states as possible.

Ketchum differs from the previous published results for several reasons. First, Ketchum provides 10x higher capacity than previous published results. The higher capacity is achieved by carefully orchestrating simulation and multiple formal methods including *symbolic simulation*, *SAT-based BMC*, *symbolic fixpoint computation* and *automatic abstraction*. Second, Ketchum performs not only automatic test generation but also unreachability analysis, which enables the test generation effort to be focused on coverage states that are not unreachable. Third, the backbone of Ketchum is an off-the-shelf commercial simulator. It enables Ketchum to reach deep states of the design quickly and supports simulation monitors through the standard API of the simulator during test generation.

We applied Ketchum to several industrial designs, including the picoJava microprocessor from SUN and the DW8051 microcontroller from Synopsys and obtained very promising results. The experiments show that Ketchum can (1) handle design blocks containing more than 4500 latches and 170K gates, (2) reach up to 6x more coverage states than random simulation and (3) identify a majority of the unreachable coverage states.

1. Introduction

Functional verification checks if the functionality of the hardware design meets the specification. A typical method for "bullet-proofing" the functionality of the design is random simulation [11]. Random simulation can leverage today's fast simulators and computer farms by using some sophisticated test harness. Verification engineers need to build a model of the environment in which the DUT operates. During random simulation, biased pseudo random generators drive the primary inputs of the environment model with random values and the environment model then drives the primary inputs of DUT with random but legal stimuli. The behavior of the DUT can be checked by simulation monitors during the simulation. We call the model that consists of the DUT and the environment model the *model under test (MUT)*.

To measure the quality of the verification effort, verification engineers apply *coverage metrics* to estimate how thoroughly the input stimuli have exercised the design. Coverage metrics directly based on the source code of the RTL design, such as line coverage, are too weak because they do not take the concurrency of hardware designs into consideration. Consider two finite state machines (FSMs) that control a buffer (see Figure 1).

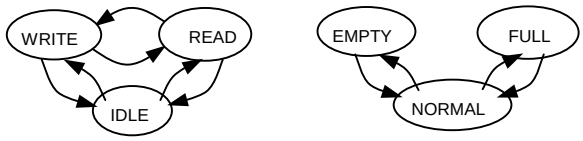

Figure 1. Two FSMs that control a buffer

The first FSM represents the operation that is being performed on the buffer, the second FSM the status of the buffer. It's possible to achieve 100% line coverage on the corresponding RTL description by

exercising only 3 cross-states: idle/empty, write/normal, read/full. However, interesting corner cases such as read/empty (reading when the buffer is empty) and write/full (writing when the buffer is full) could be left unvisited.

State coverage, on the other hand, can distinguish different concurrent events. Given a set of interesting signals, which we call *coverage signals*, the state coverage measures how many different combinations of the values of the coverage signals have been reached during simulation. We call each combination a *coverage state*. For the above example, if we select the set of signals that encodes the states of the two FSMs as our coverage signals, 100% state coverage indicates that all nine cross states have been visited.

Coverage signals are usually selected from signals that constitute pipeline control, interacting FSMs, decoder outputs, status registers or other key control signals. By forcing the design into many different coverage states, the probability of detecting errors is increased. For the above example, a simulation monitor may detect that some valid data in the buffer is mistakenly over-written during a write to a full buffer. Therefore, verification engineers may want to construct input stimuli to maximize the state coverage; that is, to reach as many coverage states as possible. This task is known as *coverage-driven test generation*. Today it can be achieved by manually writing test sequences, or tweaking the biasing in random simulation. However, this approach is very expensive in terms of engineering resources.

Alternatively, formal methods like symbolic fixpoint computation and symbolic simulation can exhaustively search for an input sequence to hit a coverage state. Recently many formal and semi-formal (mixture of formal and simulation techniques) methods [1][7] have been proposed for this application. However, existing results in the literature suffer from a serious capacity limit, as they were only demonstrated on design blocks with less than 500 latches.

Our contribution is the development of a practical solution, Ketchum, to automate the test generation process for verifying real-world designs. Given (1) a synthesizable MUT and (2) a set of less than 64 coverage signals, Ketchum automatically (1) generates test sequences to reach as many coverage states as possible and (2) identifies as many unreachable coverage states as possible. The generated high-coverage tests can be replayed during regression tests. Simulation monitors written in arbitrary languages communicating to the standard API of the commercial simulator can be used during test generation for catching bugs.

The target capacity of Ketchum is to handle synthesizable MUT in the range of 100K gates and 5K latches. This range encompasses design blocks of integral functionality and many IP blocks, to which random simulation may be applied today.

Ketchum's unreachability analysis employs *symbolic fixpoint computation* [12] and robust *automatic abstraction techniques* to prove many coverage states unreachable. Its automatic test generation utilizes a combination of simulation and formal methods like *symbolic simulation* [1][4] and *SAT-based bounded model checking* (SAT-based BMC)[2], to generate input sequences to achieve high state coverage.

More specifically, random simulation and formal methods are interleaved to perform a deep and wide exploration of the state space. Starting from a given initial state, random simulation quickly hits several new coverage states but will eventually stops reaching new coverage states so quickly. Then one of the formal methods is used in an exhaustive search for a new coverage state. This search is narrowed to just those states not yet proven unreachable. The process then repeats, leveraging the ability of simulation to search deep, and the ability of formal methods to search wide.

We applied Ketchum to several industrial designs, including the picoJava microprocessor from SUN and the DW8051 microcontroller from Synopsys and obtained very promising results. The experiments show that Ketchum can (1) handle design blocks containing more than 4500 latches and 170K gates, (2) reach up to 6x more coverage states than random simulation and (3) identify a majority of the unreachable coverage states.

The remainder of the paper proceeds as follows. Section 2 describes our method for generating input sequences to reach coverage states, and Section 3 our algorithm for proving coverage states unreachable. Section 4 presents the experimental results. Having defined the problem and explained our approach in detail, Section 5 describes related work, and finally Section 6 concludes the paper.

2. Test Generation

We present a test generation algorithm that combines (1) random simulation, (2) symbolic simulation and (3) SAT-based bounded model checking to generate stimuli to reach coverage states. We will first provide a brief overview of these three search techniques.

A *state* (resp. *coverage state*) of a design is a valuation of all signals (resp. *coverage signals*) of the design. Given a starting state, random simulation

automatically generates a trace of the MUT from the starting state by driving the primary inputs of the MUT with random values. During the random simulation, we observe the values of the coverage signals. If the values of the coverage signals indicate that a new coverage state has been reached, we store the new coverage state and mark it as reached. The advantage of random simulation is that it is extremely fast and it can generate traces that reach very deep states in the state space. The disadvantage is that it only searches along a single trace at a time. We perform random simulation using a commercial simulator. Note that off-the-shelf simulators usually perform much better in generating a single very long trace than a lot of short traces where the simulator needs to be injected with the same starting state over and over again. We use random simulation as our long range search engine to reach deep states.

Symbolic simulation drives each primary input of the MUT with a new symbolic variable at each simulation step [4]. It computes the symbolic formula for each signal according to the logic in its fanin. Notice that the DUT will be driven by legal symbolic formulas because of the environment model in the MUT. The symbolic formulas are stored as Binary Decision Diagrams (BDDs) [3]. Given a starting state, the *i*-th step of symbolic simulation can reach any new coverage state that is *i* steps away from the starting state. We store the unclassified coverage states as a BDD, called *unclassified* BDD. During symbolic simulation, we check if a new coverage state has been reached by substituting the coverage signals in the unclassified BDD with the symbolic formulas of the coverage signals. If the result of the operation is not the empty set, a new coverage state has been reached by symbolic simulation. In that case, we update the unclassified BDD and generate a trace to be used in simulation, as discussed later. Otherwise we continue performing additional steps of symbolic simulation until a new coverage state is reached or some memory or time limit is reached.

Table 1. Comparison of Search Engines

Engine	Effective Search Range	Strength	Limitation
Random simulation	Long	Deep states	Single trace
Symbolic simulation	Medium	Designs with fewer inputs	Time, memory, length of trace
SAT-based BMC	Short	Short hit traces	Time, length of trace

Two important observations about symbolic simulation can be noted. First, the number of symbolic variables that have been used during simulation has more impact on the complexity of symbolic simulation than the number of latches in the MUT. The number of symbolic variables used in the symbolic simulation is the number of primary inputs times the number of simulation steps. Second, when the size of the BDD that represents the simulation values gets unwieldy, we can easily under-approximate the symbolic simulation values by setting some symbolic variables to Boolean constants [1][6]. As a result of these two observations, we can efficiently perform symbolic simulation on designs with thousands of latches and hundreds of primary inputs for 10 to 50 steps. Because of our first observation, symbolic simulation generally is not adequate to generate traces to reach coverage states that are more than a few tens of steps away from the initial state. Thus, we use symbolic simulation for middle-range exhaustive (or semi-exhaustive with underapproximation) search.

SAT-based BMC was introduced in [2]. Given a particular starting state, we can use a SAT solver to search for a trace up to a certain length *i* to reach a new coverage state. The targeted coverage states can be restricted to coverage states that are not yet reached or proven unreachable by the unreachability engine. Within a certain time limit, a *complete* SAT solver [14] will: (1) find a trace to reach a new coverage state, or (2) prove that no new coverage states can be reached by traces up to length *i*, or (3) return no conclusions. If the outcome is (2) or (3), we apply the SAT solver to search for traces of length *i+1*, until some memory or time limit is reached. For designs with thousands of latches and hundreds of inputs, SAT-based BMC usually requires a lot less memory and is a lot faster than symbolic simulation for finding traces of length less than 10. Consequently, we use SAT-based BMC as our short-range exhaustive search engine.

Figure 2. Test generation algorithm

Knowing the advantages and disadvantages of each search engine (summarized in Table 1), we orchestrate the search engines to hide the disadvantages and exploit the advantages of individual engines. We want to perform a deep and wide search by randomly simulating to a deep state

and, starting from that state, a short or middle range wide (exhaustive) search using SAT-BMC or symbolic simulation.

Figure 2 illustrates the test generation algorithm. The rectangular box represents the entire state space of the MUT and the stars represent the coverage states in the state space. The algorithm starts off by playing an initialization sequence provided by the user to reach the initial state of the MUT (a). Then, we perform random simulation (the generated trace is represented as the zig-zag line) and quickly reach easy-to-reach coverage states.

After most of the easy-to-reach coverage states have been reached, the rate of reaching new coverage states falls below a heuristic threshold (b). At this moment, we kick off our SAT-based BMC with the current state of random simulation as the starting state. If SAT-based BMC does not reach a coverage state within the search range, we kick off symbolic simulation after that. The search space of an exhaustive search engine is represented as nested circles in Figure 2. When, the exhaustive search engine finds a new coverage state, it generates a sequence of input assignments that is then replayed by the simulator (represented by the gray arrows). If both exhaustive search engines run out of time or memory limits without finding a new coverage state, we will return control to the random simulator. In either case, we start random simulation again and repeat this process until a desirable coverage number is reached. The final output of Ketchum will be a single long trace that traverses all the reached states, which is stored in a separate file for use during regression testing of the MUT. Since all the traces returned by the formal engines are replayed through the simulator, the simulation monitors will function correctly just as in ordinary random simulation.

We noticed that after an exhaustive engine reaches a new coverage state, the subsequent random simulation run could often reach a bunch of new coverage states quickly. We suspect that the reason is that among the set of coverage signals, some coverage signals are relatively easy to transition from one value to another and some coverage signals are relatively hard to transition. As a result, after an exhaustive engine manages to reach a new combination of the hard-to-transition coverage signals, random simulation will bump into different combinations of the easy-to-transition coverage signals, which combined with the new combination of the hard-to-reach signals become new coverage states.

3. Unreachability

Proving coverage states unreachable is a major aspect of coverage-driven test generation. Knowing the number of unreachable coverage states provides a more accurate measure of the quality of a simulation test bench. Moreover, knowing specifically which states are unreachable allows the reachability engine of Ketchum to focus on a much smaller set of states.

The goal of the unreachability engine of Ketchum is to provide fast and robust results without necessarily trying to detect all of the unreachable states. Simply put, we sacrifice exactness in favor of capacity and robustness. To this end, we have designed a straightforward algorithm that is conservative: it can prove states unreachable, but cannot prove states reachable. The basic idea of the algorithm is one that has been employed by many others before: perform exact analysis on a pruned model of the MUT. However, we introduce novel techniques in the way we select the latches and the combinational logic to include in this pruned model.

Specifically, we first select a small set of latches that includes the coverage signals; the remaining latches are treated as primary inputs. Next, a novel cutting procedure is used to reduce the fanin of the chosen latches, to further simplify the analysis. Then, exact reachability analysis is performed on the pruned model using BDDs. The computed set of reachable states is projected onto the coverage signals. Coverage states in the complement of this projection are provably unreachable, since the pruned model is an abstraction of the original. In the following we detail the latch selection and logic cutting procedures of the algorithm.

The latch selection process starts with the coverage signals. We then add latches incrementally in a breadth-first search (BFS) of the latch dependency graph of the MUT until we reach a user-specified limit on the number of latches. If all the latches of the last BFS level visited cannot be included within the limit, then an arbitrary subset from that level is used.

After a subset of latches has been selected, we employ a cutting algorithm to reduce the number of variables in the support of the transition functions. Experience has shown that even if only 50 or so latches are being analyzed, if the transitive fanin of the latch subset depends on many hundreds of primary inputs and non-selected latches, then later BDD computations will be intractable.

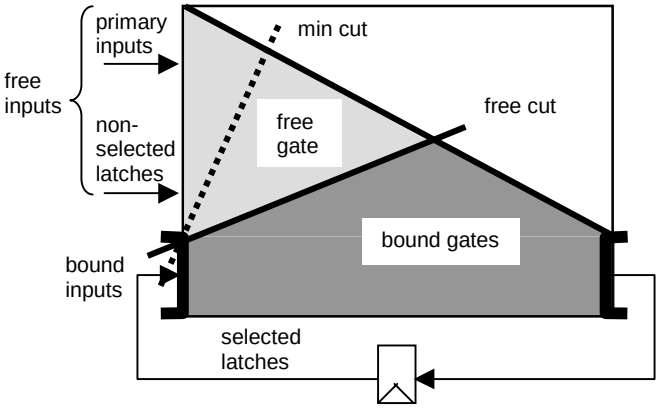

Figure 3. Definition of min-cut for cutting procedure

To explain the algorithm we define a few terms. Consider the transitive fanin of the selected latches (see Figure 3). The *bound inputs* are the outputs of the selected latches. The *free inputs* are the primary inputs and the outputs of the non-selected latches. A gate in the transitive fanin of the selected latches is *bound* if its transitive fanin contains a bound input, otherwise it is *free*. Now consider the free gates that directly feed bound gates; we call the collection of such signals the *free cut*. The signals on the free cut may be correlated, but they do not depend on the bound inputs. Hence, it would appear to be a good tradeoff to replace all the signals on the free cut with primary inputs, thus removing all the free gates from the BDD computations. But, we can do much better, for it is not the number of gates that we want to reduce, but the number of signals in the support of the transition functions.

To this end, we extract a directed network from the signal dependency graph (the graph whose set of nodes is the set of signals and set of directed edges is the fanin relation). The sources of this network are the free inputs, and the sinks are the signals on the free cut. We then compute a minimum cut of this network and replace the signals on the min-cut with primary inputs. The min-cut has the advantage of not only reducing the number of variables in the BDD computations, but also of maintaining more correlation relative to the free cut.

Experience shows that using the min-cut almost always produces the same set of unreachable coverage states as compared to no cutting at all. Furthermore, there are cases where the reachability fixed point computation only completes when using the min-cut. For instance, in the case of IU, one of the experimental results reported below, the cache miss signal originally had 1102 variables in its support, and reachability could not complete. The min-cut procedure reduces the support to 60 variables, and enables reachability to complete with a result that is superior to that of just leaving the cache miss signal out of the analysis entirely.

It is clear that the unreachability algorithm computes a lower bound on the number of unreachable coverage states of a MUT. In other words, every coverage state that was identified as unreachable is really unreachable, but an unreachable coverage state may be missed due to automatic abstraction.

4. Experimental Results

We implemented a prototype of Ketchum in C. We used a commercial Verilog simulator (VCS) as our random simulator and we developed in house all the other engines. Our symbolic simulator and unreachability engines rely on the CUDD package [18] for BDD computations. The SAT engine is an implementation of the GRASP algorithm [14].

We applied our prototype of Ketchum to some real-world designs. Table 2 summarizes the experimental results on the Integer Unit, Data Cache Unit and Stack Management Unit of the Sun picoJava microprocessor [16], the entire Synopsys DesignWare DW8051 microcontroller, and a commercial bus controller design “Bus”. Each experiment was run on a Sun Solaris server of 4 processors and 4GB memory.

The first 3 columns show the characteristics of the five designs. The 2nd column shows the number of latches in each design. The 3rd column shows the number of coverage signals. Note that the number of coverage states is 2 to the number of coverage signals. In order to choose the coverage signals, we located the control FSMs in the source code of the designs. Then we chose the latches that encode these control FSMs as the coverage signals. For example, for the case of DCU, we identified the “cache-miss” FSM (6 latches), the “cache-fill” FSM (5 latches), the “write-back” FSM (8 latches) and the “zero-out” FSM (6 latches). Because all these FSMs are 1-hot encoded, there are at most 6x5x8x6=1440 reachable coverage states. Among those coverage states, Ketchum identified that only 111 are potentially reachable and it was actually able to generate tests that reach all of the 111 coverage states. Note that Ketchum was not told in advance about the FSM encoding.

The 4th column shows the unreachability results: the number of coverage states that were NOT identified by Ketchum as unreachable (the number of potentially reachable coverage states) and the CPU time taken by the unreachability analysis. We can see that Ketchum can effectively identify the majority of the coverage states as unreachable. The number of

latches that we include in the abstract model is 50 across all designs.

In the next 3 columns we compare two automatic test generation approaches, random simulation and Ketchum, in terms of the effectiveness of reaching coverage states. The 5th column shows the number of coverage states reached by random simulation and the time taken by random simulation (24 hours). The 6th column shows the corresponding results for Ketchum. The 7th column shows the percentage improvement in terms of the number of coverage states that Ketchum reached over random simulation. We can see that Ketchum uniformly outperforms random simulation by a large margin. The exception is the DCU unit of picoJava, on which random simulation reached almost as many coverage states as Ketchum but cost 700x more CPU time.

Table 2. Experimental Results

DUT	Ltch	Cov sig.	Cov state aftr Kchm unreach	Reach cover states Rndm	Reach cover states Kchm	Imp (%)
IU	4558	17	230 *445sec*	9 *24hr*	40 *24hr*	344
8051	784	11	896 *299sec*	317 *24hr*	597 *24hr*	88
DCU	385	25	111 *259sec*	109 *24hr*	111 ***2min***	2
SMU	217	16	132 *1423sec*	104 *24hr*	132 ***45min***	30
Bus	155	16	342 *60sec*	44 *24hr*	342 ***75min***	677

During the experiments, we provided minimum environment models for the designs. We set some exception signals (like the test signals that control the scan chain) to constants and assigned very small probabilities for asserting some reset signals during random simulation. The same setting and biasing were applied to both random simulation and Ketchum. So we believe that it is a fair comparison.

Since Ketchum needs to analyze the MUT that consists of both the DUT and the environment model for test generation, the size of the DUT that Ketchum can effectively handle decreases as the size of the environment model increases. In our experience, since the environment model only needs to model the interface behavior of the DUT, a reasonable environment model of the DUT is usually much simpler than the DUT itself, especially if the DUT is an integral functional block of the design.

5. Related Work

Unreachability Analysis

Symbolic fixpoint computation [12] is the common basis for unreachability analysis. Different methods that over-approximate the reachable state space have been proposed in [1][5][8][13]. However, these methods were designed for approximating reachable "states", not reachable "coverage states", so they do not utilize the fact that the logic that is closer to the coverage signals has more impact on the behavior of the coverage signals than the logic that is farther away. The algorithms presented in [15] perform unreachability analysis on the pruned model where all latches except the coverage signals of the original DUT are treated as primary inputs. However, this may leave too many primary inputs to perform symbolic fixpoint computation. In [9], a heuristic is introduced to reduce the number of primary inputs. Our algorithm, on the other hand, finds the optimal cut to minimize the number of primary inputs.

Test Generation

Algorithms that perform some form of over-approximated symbolic image computation to guide simulation for reaching coverage goals have been proposed in [1][8][17]. However, these solutions have been ineffective so far in attacking designs with thousands of latches, on which even the first image computation may not terminate. In [1] under-approximated symbolic simulation was introduced to mitigate BDD blowup. But according to our experimental results, the under-approximation technique presented in the paper is too drastic in the sense that it often misses coverage states.

Our test generation algorithm is similar to the SIVA system described in [7] in the sense that both methods interleave simulation and formal engines to reach coverage goals. However, the following are the major differences. First, SIVA uses ATPG and symbolic image computation while Ketchum uses symbolic simulation and SAT-based BMC. We believe that using symbolic image computation for test generation is not practical. On the simulation side, SIVA computes a search tree rather than a linear trace, which prevents it from taking advantage of the speed and the deep states offered by commercial simulators. In addition, with the commercial simulator, Ketchum can utilize arbitrary simulation checkers. In terms of coverage goals, SIVA was designed for maximizing toggle coverage rather than state coverage. But in practice, toggle coverage goals in the DUT are often relatively easy to reach, so the edge of SIVA over random simulation is slim. At last, it was reported that SIVA was only tested on examples with around 400 latches whereas we

successfully applied Ketchum on real-world designs with 10x the number of latches.

6. Conclusion and Future Work

We have presented Ketchum, a tool that automates the problem of coverage-driven test generation. This novel technology combines multiple formal verification techniques and random simulation to classify most of the unreachable coverage states and reach up to 6x more coverage states than random simulation alone. At the same time, it can handle designs of more than 4500 latches, an order of magnitude more complex than published formal or semi-formal verification results.

Through the use of a robust abstraction algorithm, Ketchum is able to quickly prove most coverage states as unreachable. Central to this algorithm is a cutting procedure that reduces the variable support of the transition functions of the abstract model, enabling unreachability analysis to complete on big designs. On the reachability side, Ketchum employs a novel interleaving of simulation and formal verification techniques that exploits hard-to-reach coverage states. The combination of these approaches greatly improves the mobility of the search in the state space, thus leading to better coverage results.

There are multiple directions to extend and improve Ketchum. For coverage metrics, we want to extend Ketchum to handle transition coverage. For unreachability, foremost is an enhanced algorithm for selecting a subset of latches that takes more factors into account beyond BFS level, such as the number of fanins from, and fanouts to, the coverage signals. For test generation, we want to improve the underapproximation capabilities of the formal techniques, as well as integrate additional engines, such as sequential ATPG.

7. References

[1] J. Bergmann and M. Horowitz. Improving coverage analysis and test generation for large designs. In Proceedings of ICCAD, 1999.

[1] V. Bertacco, M. Damiani and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In Proceedings of DAC, pp. 391-396, 1999.

[2] A. Biere, A. Cimatti, E. Clarke, M. Fujita and Y. Zhu. Symbolic model checking using SAT procedures. In Proceedings of DAC, 1999

[3] R.E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35(8), pp 677-691, 1986.

[4] R.E. Bryant. Symbolic simulation--techniques and applications. In DAC, pp. 517-521, 1990.

[5] H. Cho, G. Hatchel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. IEEE TCAD, 15(12), pp. 1451-1464, 1996.

[6] D.L. Dill. Embedded tutorial: formal verification meets simulation. In Proceedings of ICCAD, 1999.

[7] M.K. Ganai, A. Aziz and A. Kuehlmann. Enhancing simulation with BDDs and ATPG. In Proceedings of DAC, pp. 385-390, 1999.

[8] S.G. Govindaraju, D.L. Dill, A.J. Hu, and M.A. Horowitz. Approximate reachability with BDDs using overlapping projections. In Proceedings of DAC, pp. 451-455, 1998.

[9] P.-H. Ho, A. Isles and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In ICCAD, 1998.

[10] R.C. Ho and M. Horowitz. Validation coverage analysis for complex digital designs. In ICCAD, 1996.

[11] M. Kantrowitz and L. Noack. I'm Done Simulating: Now what? verification coverage analysis and correctness checking of the DEC chip 21164 Alpha microprocessor. In DAC, pp. 325-330, 1996.

[12] K.L. McMillan. Symbolic model checking. Kluwer Academic Publishers, 1994.

[13] I.-H. Moon, J. Kukula, T. Shiple and F. Somenzi. Least fixpoint approximation for reachability analysis. In Proceedings of ICCAD, pp. 41-44, 1999.

[14] J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. In IEEE Transaction on Computers, pp. 506-521, May 1999.

[15] D. Moundanos, J.A. Abraham and Y.V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. In IEEE Transactions on Computers, January 1998.

[16] Sun Microsystems. PicoJava technology. http://www.sun.com/microelectronics/communitysource/picojava.

[17] C.H. Yang and D.L. Dill. Validation with guided search of the state space. In DAC, pp. 599-604, 1998.

[18] F. Somenzi. CUDD: CU Decision Diagram Package. ftp://vlsi.colorado.edu/pub/.