

# Enhancing Simulation with BDDs and ATPG

Malay K. Ganai   Adnan Aziz  
Electrical and Computer Engineering  
The University of Texas at Austin  
malay,adnan@ece.utexas.edu

Andreas Kuehlmann  
IBM Thomas J. Watson Research Center  
Yorktown Heights, NY, USA  
kuehl@watson.ibm.com

## Abstract

We introduce **SI**mulation **V**erification with **A**ugmentation (**SIVA**), a tool for checking safety properties on digital hardware designs. **SIVA** integrates simulation with symbolic techniques for vector generation. Specifically, the core algorithm uses a combination of ATPG and BDDs to generate input vectors which cover behavior not excited by simulation. Experimental results demonstrate considerable improvement in state space coverage compared with either simulation or formal verification in isolation.

**Keywords:** Formal verification, ATPG, simulation, BDDs, coverage.

## 1 Introduction

In this paper we address the problem of verifying safety properties of digital hardware designs. Conventionally, such designs are verified using extensive simulation. A model of the design is built in software, to which small FSMs called monitors are added. These monitors check for failures of the user-specified safety properties. Large numbers of input sequences, called tests, are applied to this model; these tests are generated by (possibly biased) random test pattern generators, or by hand. If for a given test the safety property is violated, the corresponding monitor enters a “violation” state, flagging the failure.

Simulation is simple, and scales well in the sense that the time taken to simulate is proportional to the design size. However, simulation offers no guarantee of correctness; more disturbingly, for large designs, the fraction of the design space which can be covered in this methodology is vanishingly small [5].

This state of affairs has led to the proposal of “formal methods” for design verification; the adjective formal refers to the unambiguous specification of the system and the properties being checked, together with the validation step, which *systematically* explores all possible ways in which the system could fail the properties. The computational complexity of formal verification is extremely high. Heuris-

tic procedures have been suggested that perform well on specific classes of designs. One approach which has been used to successfully formally verify a large number of complex designs is “symbolic model checking”. The basic idea underlying symbolic model checking is the use of BDDs to implicitly represent set of states, next state functions, and perform basic FSM manipulations [18].

The primary limitation of BDD-based approaches to invariant checking is that for many designs, the BDDs constructed in the course of verification grow extremely large, resulting in space-outs or severe performance degradation due to paging. Symbolic approaches are limited to designs containing of the order of a few hundred state holding elements; this is not even at the level of an individual designer subsystem.<sup>1</sup>

In the course of our discussions with real-world hardware verifiers we have been struck by the fact that practicing verifiers are less concerned with providing formal proofs of correctness for their designs than they are with finding bugs in them as early as possible. Indeed, it has been remarked out that “falsification” is a more accurate description of the endeavor commonly called “verification”.

Faced with the twin dilemmas of diminished coverage through simulation and the computational infeasibility of complete systematic verification, it is natural to ask how best to use systematic methods in conjunction with simulation to find bugs in designs. In this paper we provide such an approach, which is based on augmenting simulation with two symbolic techniques, namely combinational ATPG and BDDs. We stress that the approach is not complete, i.e., not guaranteed to find a counter example, if the design fails to satisfy its properties. However, our procedures are sound — all reported violations of the safety properties are true bugs. We stress that our goal in this work is to find more and different bugs more efficiently and cheaply than either simulation or systematic verification used in isolation.

### 1.1 Previous Work

In this section we survey work in verification which is directly and peripherally related to combining simulation and formal verification.

---

<sup>1</sup>Certain designs containing thousands of latches have been verified. Typically, they are extremely simple consisting, for example, of iterated arrays of processors [1].

### 1.1.1 BDD-Based Approaches

The approach taken by Ravi et al [22] is to perform symbolic reachability analysis with subsetting. Whenever the BDD for the visited state set grows beyond a threshold limit, a subset of the set is taken in a manner, which heuristically preserves a large fraction of the original set while reducing BDD size. We are philosophically opposed to this, as it does not differentiate between states at all. Yuan et al [3] attempted to overcome this limitation by taking subsets which preserve all the distinct control states in the subset. Elsewhere, BDDs have been used for “target enlargement” – successive pre-images are computed from the set of “fail” states until the BDD for this set grows large; this is used as the target of simulation [27].

We have experimented with all these approaches. Our basic criticism of them is that they do not scale well. Image or Pre-Image computation is required by both; we found that this in itself to blow up, even when dealing with state sets whose BDDs are small. We tried pulling subsetting into the distributed image computation [11], we routinely found that we got empty images. Furthermore, BDDs are a very poor representation of state sets when there is limited node sharing as is commonly the case with the approach of Yuan et al [3]. In view of this, we elected not to pursue BDD-based FSM analysis.

### 1.1.2 Test synthesis

Several papers have recently emerged which address the problem of generating test sequences which excite a large fraction of the control space: [14, 20, 10]. Specifically, these sequences are derived for an abstract control model, and then “expanded” to the entire design. Clearly, the expansion is a nontrivial process. The papers [14, 10] gloss over this step. Moundanos et al [15] use a sequential ATPG tool to generate paths from state to state in the global machine. Our argument against these approaches is that it is not at all clear what they buy; the problem of performing expansion is as high as that of formally verifying the entire design. It may be possible to manually spell out the expansion step, but then the procedure is no longer automatic.

### 1.1.3 Coverage estimation

This research is aimed at evaluating the effectiveness of an existing set of tests. The authors of [13, 15] operate on designs which can be split into control and datapath. An abstract representation of the controller is built, and it is determined which edges in the STG of the controller are excited by the verification test suite applied to the design. In this way, the designer may discover that certain important functions of the controller have not been excited during simulation. The primary drawback of these approaches is that since the inputs to the controller are abstracted, the uncovered edges may be unreachable in the complete design. A related paper is that of Devadas et al [7], wherein estimation of test suite coverage is derived for high-level designs using a generalization of the  $\{0, 1, D, \bar{D}, X\}$  calculus used for fault simulation. They make the important point that bugs need to be both excited as well as made visible.

### 1.1.4 Conservative approximations

Several authors have elected to work with sacrificing FV in the “other” direction, i.e., they never report false positives [2, 17]. A simple view of these papers is that they ap-

proximate the design and build an over-estimate of the set of reachable states. Consequently, if the tools reports that the design satisfies the specified invariants, it really does so. However, there is the problem of false negatives: checking that the error trace reported holds in the true design is non-trivial. Furthermore, the design has to be “deabstracted” when the bug is false; by the time the procedure converges, the resulting model could be easily as complex as the original design, and the entire process will be far more time consuming. We have not seen compelling experimental evidence in favor of this approach.

### 1.1.5 Sequential ATPG

There are commonalities between our work and sequential ATPG in that we try to justify nets. However, much of the work in sequential ATPG is orthogonal to our effort. For example, reduction in the amount of “scan logic” is a major theme in [26]; there is no analog of this in functional verification. Similarly, much effort in sequential ATPG is devoted to dealing with the lack of a reset signal and the possibilities of X values [9]. Sequential ATPG approaches based on BDDs have been proposed by Cho et al [6]; these suffer from the same limitations of BDD-based model checking mentioned previously.

## 1.2 The Case for a Simulation Based Approach

In this work, we have decided to focus on large designs, at least of the order of individual designer subsystems (50K gates, 2K latches). Based on our experiences with BDD-based model checking (both exact and approximate) we felt it was inappropriate for such an application. Two techniques which have been successfully applied to analyzing large designs are cycle-simulation and combinational verification. The best combination verification tools tightly integrate random simulation, combinational ATPG, and BDDs to achieve robustness [16, 21]. The sharing of information between subroutines allow them to achieve much greater efficiency than any of them in isolation. Even though we are addressing a problem which is in some way quite different from combinational equivalence checking (there is no concept of “correspondences between netlists” in our domain), it is our premise that “dove-tailing” between different sequential-search strategies is imperative. Furthermore, we will show that combinational ATPG and BDDs fit very nicely with simulation-based state space search. Based on this we have put together an invariant verification tool, we call SIVA.

A high-level description of our main procedure is as follows: Designs are specified as a netlist of gates and latches<sup>2</sup>. We start by applying a fixed number (say  $N$ ) of randomly generated input vectors to the initial state; this gives us a set of states that can be reached in one step from the initial state. For each node in the netlist, we keep its signature, i.e., an  $N$  bit sequence, where the  $k$ -th bit corresponds to the value taken by the node on the  $k$ -th input vector. We determine which nodes have a constant 0 signature, and which nodes have a constant 1 signature.

We then try to construct input vectors under which nodes with constant 0 signature get set to 1, and nodes with constant 1 signature get set to 0. Vector generation is performed by a combination of SAT-based ATPG, and BDD building. When the ATPG/BDDs show no vector exists, we go on to

<sup>2</sup>For simplicity, we assume a single initial state.

the next node. To keep the procedure robust, backtrack limits are set on the ATPG routine, and node limits are set on BDD size; if these are reached, we abort the vector generation for the given node, and proceed to the next one. After completing analysis from the initial state, the procedure is called recursively on the states reached.

The above description is extremely simple. For example, we found it useful to restrict attention to a (user-specified) set of nodes. Also, the number of states grows so large that some prioritization/pruning was needed. These extensions along with detailed descriptions of the simulator and solver are given in Section 3.

### 1.3 Paper Organization

The rest of this paper is structured as follows: Section 2 reviews relevant background material. In Section 3 we present detailed accounts of our approach. Experimental results are given in Section 4. We summarize our contributions, and suggest future work in Section 5.

## 2 Background

### 2.1 RTL descriptions and Indicator variables

The basic unit in the RTL description of a design is a *module*; this consists of a set of input variables, a set of output variables, a set of declarations (consisting of registers, combinational variables, and module instantiations), and the module body. The latter can be viewed as consisting of a series of conditional assignments to combinational variables. Later, we will see that it is useful to “guide” the simulation to unexplored behaviors by adding Boolean-valued “indicator” variables.

Specifically, we will “annotate” designs in the following manner: for every module in the description, for every control block<sup>3</sup>, add a fresh Boolean-valued register variable  $B_i$  to the module. Initialize the variable to 0, and set its value to 1 if at the current state, the control block is executed. Essentially, these new variables act as indicators, denoting if the corresponding block is being exercised; we will use these to “guide” our search.

We illustrate the application of indicator variables by an example, as shown in Figure 1. Here we added two indicator variables  $B_0$  and  $B_1$  that indicate whether particular conditions were evaluated to be true. More generally, we can add indicator variables corresponding to whether a control FSM is in a particular state, or the values of two data registers are equal. In general, the indicator variables can be used for the usual notions of RTL/FSM coverage including structural (e.g., line coverage) or functional (e.g., FSM edges) [8].

### 2.2 Invariant Verification

A common hardware verification paradigm is that a designers specifies a set of “good states” and then a verifier ensures that the set of reachable states lies in this set.

This problem is variously known as *invariant verification*, or *assertion checking*. Though conceptually simple, invariant checking can be used to verify all “safety” properties when used in conjunction with monitors. These are FSMs which are added to the designs; they observe the design and enter an error state when the corresponding safety property fails and “raise a flag”.

<sup>3</sup>sequence of statements guarded by boolean predicate, controlling their execution

```

module main(l);
  module foo A(l, x, y, z, w );
  module foo B(l, z, w, x, y);
endmodule
module foo(l, a, b, c, d );
  input l, a, b;
  output c, d;
  reg p, q, pc;
  reg B1, B2; /* Indicator variables */
  initial begin
    B1 = 0; B2 = 0;
    p = 0; q = 0; pc = 0;
  end
  assign c = p|q;
  assign d = p&q;
  always @(posedge clk) begin
  case(pc)
    0:
      if ((l) && (a == b)) { /* 1st indicator */
        B1 = 1; /* Condition has been taken */
        p = a; q = 1; pc = 1; }
    1:
      if ((~l) && ~(a == b)) { /* 2nd indicator */
        B2 = 1; /* Condition has been taken */
        p = b; q = a; pc = 0; }
  endcase
  end
endmodule

```

Figure 1: An example of a design instrumented with indicator variables

```

if ( a+b == c && PS == 'st0 )
begin
  PS = 'st1;
  out = 'FLOW;
end

```

Figure 2: RTL code fragment

One approach to formal invariant checking is to compute all states reachable from the initial state, and checking that they all lie in the invariant. This method, like all formal verification approaches to invariant checking, suffers from very high computational complexity.

## 3 Augmenting Simulation with ATPG and BDDs

In this section, we give a detailed account of the basic algorithm underlying SIVA.

SIVA reads designs specified in the .blif format which is a textual representation of a netlist of gates and latches. Internally, complex gates are broken down into two-input ANDs exploiting the local structural equivalence. States are represented using bit vectors; sets of states are represented by hashing states into a hash table.

The motivation for the search strategy used by SIVA can be seen by considering the RTL code fragment shown in Figure 2. If  $a, b$  and  $c$  are 16-bit inputs, the probability of a single random input when  $PS == 'st0$  is extremely low - of the order of  $2^{-16}$ . It is precisely in order to generate

inputs enabling these transitions that we use a “solver”. The solver is based on combinational ATPG and building BDDs for combinational logic.

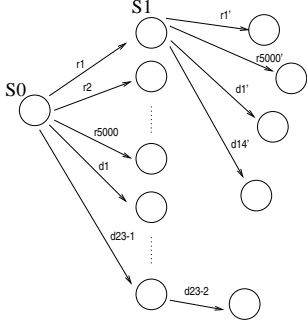


Figure 3: FSM view of directed simulation.

When the RTL is compiled to a netlist of gates and latches, a gate  $G$  will exist which evaluates to the expression  $(a+b == c \ \&\& \ PS == 'st0)$ . Performing say 5000 random simulations at state where  $PS == 'st0$  will most likely result in the signature of  $G$  remaining at constant 0, which leads to an invocation to the solver which will return an input vector for which  $(a+b == c)$ . Thus, search proceeds as shown in Figure 3. At state  $S_0$  we perform  $N$  random simulation steps, followed by calls to the solver; vectors generated by the solver are immediately “fed back”, i.e., simulated at state  $S_0$ .

The basic procedure **Ver-Sim** underlying SIVA is presented in Figure 4. It follows the approach given above with some enhancement. Rather than performing signature analysis on all network nodes, the user can specify a list of nodes  $T$  which he deems as being important. We chose this flexibility as it saved time for signature analysis otherwise, the number of calls to the solver grew very large. Typically,  $T$  would consist of inputs to latches, and indicator variables corresponding to conditions/line covered/control transition etc. as described in Section 2.1.

The procedure **Random\_Simulate\_With\_Sig\_Generation** performs simulation by evaluating the netlist nodes in topological order starting from PI’s and latches. Simulation is performed word-wise, allowing 32 input vectors to be processed in one pass.<sup>4</sup> As in [16], all gates are two-input ANDs with bubbles on edges to denote inversion. Thus, simulation is extremely fast.

Node signatures are stored as bit vectors, where the  $k$ -th bit corresponds to the value of the node on the input  $k$ -vector. Signature analysis is done by hashing the nodes based on the signature.

States are also represented using bit vectors. Visited states are stored in a hash table; along with the state, we store a pointer to the predecessor state and also to the input vector which yielded it. In this way, for any state visited, we can trace a path back to the initial state. The overhead is acceptable.<sup>5</sup>

<sup>4</sup>We also experimented with BDD-based simulation which is very fast on small examples. For large designs the BDDs take up very large amount of memory, unless the partition threshold is small, at which point much of the lookup advantage of BDDs vanishes. Additionally, there is no facility for word-level simulation on BDDs.

<sup>5</sup>Such a scheme makes it possible to considerably reduce memory used for storing states. The input vector is typically much narrower than the state vector; thus we can perform compaction on states [25]. Since we store the input vectors we will be able to recover the actual state by applying the corresponding input sequence to the initial state. We are currently experimenting with this approach.

```

/* The routine attempts to identify a set of input vectors */
/* which result in each node in T to take both 0 and 1 values */
/* η is the design, s0 is a state in the design, */
/* T is target node list, */
/* (T is entire set of nodes, if not user-specified), */
/* N is Number of random vectors to apply. */
Boolean function Ver_Sim(η, s0, N, T) {

    /* Simulate N random vectors at s0, */
    /* Keep signature for all nodes, */
    /* Update the reached state set S, */
    /* Remove Nodes from T which are found to be not constant */
    S := S ∪ Random_Simulate_With_Sig_Generation(η, s0, N, T)

    for_each_target_node_with_constant_sig(η, node) {
        vec = Symbolic_Solve(η, s0, node);
        if (vec ≠ ∅) {
            /* Simulate with vector vec and update sig */
            S := S ∪ Update_Signatures(η, s0, vec);
            /* Remove node from T */
            T := Prune_List(T, node);
        }
    }

    Mark_Done(s0);
    t := Select_State(S);
    if (t ≠ ∅) /* no remaining state */
        Ver_Sim(η, t, N, T);
}

```

Figure 4: Pseudocode for SIVA.

```

/* Create input vector resulting in non-constant */
/* signature for node. For simplicity, assume that */
/* the existing signature is constant 0 */
Input_t function Symbolic_Solve(η, s, node) {

    faninConeNodeList := Build_Fanin_Cone(node);
    faninConeClauseList := Build_Clauses(faninConeNodeList);
    Add_Clause(faninConeClauseList, (node == 1));

    for(i=0; i < MAX_TRY; i++) {
        backTrackLimit := B1 * (C1)i;
        /* B1, C1 are run time constants */
        bddSizeLimit := B2 * (C2)i;
        /* B2, C2 are run time constants */
        (result, vector) := Sat_Solve(faninConeClauseList,
                                   backTrackLimit);
        if (result == SATISFIABLE)
            return vector;
        elseif (result == NOT_SATISFIABLE)
            return ∅;
        else { /* result == BACKTRACK_EXCEEDED */

            (result, BDD) = Build_BDD(node, bddSizeLimit)
            if (result == BDD_Is_Not_Zero(BDD))
                return BDD_Get_Minterm(BDD);
            elseif (result == BDD_Is_Zero(BDD))
                return ∅;
            else /* result == LIMIT_EXCEEDED */
                continue;
        }
    } /* end of for */
}

```

Figure 5: Pseudocode for ATPG/BDD-based Symbolic Solver.

We now invoke a deterministic procedure, denoted by **Symbolic\_Solve**, to target each node in the list  $T$  whose signature is a constant. **Symbolic\_Solve** finds a vector under which the node take a value opposite to the constant. (We found it advantageous to also consider nodes pairwise – if two nodes have same signature we attempt to differentiate them using **Symbolic\_Solve**. The extension to node pairs is simple; for brevity we have left it out of this paper.)

The outline of **Symbolic\_Solve** is described in Figure 5. It is a combination of SAT-based Combinatorial ATPG [24], and BDD building. For the SAT solver we used the PODEM heuristic [12] to pick input variables to branch on. For robustness, we imposed backtrack limits on SAT-solver.

As we allow only two-input AND gates, each gate corresponds to three clauses (CNF expressions). The routine **Build\_Fanin\_Cone** returns *faninConeNodeList* which is set of nodes in the fanin cone of the node. The procedure **Build\_Clauses** returns *faninConeClauseList* which is a list of clauses corresponding to each node in *faninConeNodeList*. If **Sat\_Solve** comes up with a satisfying assignment, we return that as the witness. On the other hand if it exceeds the backtrack limit, we invoke a BDD-based approach. We build BDD using ITE operator and also we keep dynamic variable reordering enabled [23]. To prevent BDD explosion, we posed a limit on BDD size. If we find the BDD has a satisfying assignment, we pick and return one minterm. For robustness, we alternate between **Sat\_Solve** and **Bdd\_Build** increasing the backtrack limit/BDD size threshold upto max values for each. After invoking the solver, if it returns a vector, we perform simulation, **Update\_Signatures** using the witness *vec* generated. We update the states  $S$  and also prune the target list  $T$ .

To increase the coverage of the target node, the user may select to provide a list of “light-houses”<sup>6</sup> for selected target nodes (which in turn could be other indicator variables). The solver will try to generate input vectors for the light-houses at every state till the desired target is reached. The list of light-houses is not pruned until the desired target is reached. We found that the overhead of not pruning the list of light-houses is much smaller than the overhead of not pruning the target list itself.

After calling **Symbolic\_Solve** on each target nodes, we mark the state as being “done” (this state is not selected again) and **Select\_State** selects a new state. The procedure **SIVA** is called recursively on the states reached. The routine **Select\_State** can be implemented in DFS or BFS order of state traversal. In our approach, we implemented BFS order. In other words, say we start from the initial state  $s_0$ . We invoke **SIVA** with  $s_0$  to obtain a set of states reachable in one step from  $s_0$ ; call this  $S_1$ . Now we select each state from  $S_1$  and invoke **SIVA**. Let  $S_2$  denote the states reached from  $S_1$ . This process is continued till we have no state unmarked or we reach all the targets nodes.

The state set  $S_i$  can grow to be very large. On inspection, we found that there many registers that correspond to datapath. One way to pruning the state set is to hash states only on the (user-specified) control latches. We implemented this, and found it reduced the sets considerably (Yuan et al [3] uses the BDD *cproject* operation to achieve the same effect on state sets represented as BDDs). One point worth mentioning is that the indicators are not part of the design and they don’t control execution of any code blocks. Hence these variables can be safely kept away from

<sup>6</sup>The “light-house” play a role more than just a guide-post as suggested by Yang et al [27]. At every state, it “illuminates” or helps to generate interesting states that can lead to target eventually.

the list of control latches.

## 4 Experiments and Results

There is a paucity of meaningful examples available to researchers in the design verification area. To test our procedure, we constructed a large design by connecting several smaller units (**amp**, **shifter**, **ethernet**) derived from the VIS benchmark suite [28]. To this, we added 18 indicator variables which were derived by inspection. The resulting design has 431 latches and equivalent to 24951 two-input AND gates. Our target set consisted of 30 nodes including both control input and indicator variables. We performed  $N = 1000$  word-level random simulations at each state. To identify the benefits of augmenting simulation with a solver, we compare target coverage numbers achieved by random simulation to those achieved by **SIVA**. We used 1000 seconds for each set of experiments.

In the first set of experiments we did not prune states. For pure random simulation method, we could apply 66000 word-level input vectors in the given time limit. We found only 5 targets could be covered. On the other hand, with **SIVA** we could cover 20 targets. Invocation to **Symbolic\_Solver** took only 1% of the total time. We found most of the time, **Select\_State** picked un-interesting states. This was expected given that we did not prune states. To experiment with the benefits of pruning, we identified 14 control latches. We ran our experiment for 1000 seconds, hashing states only on control latches. We found again that only 5 targets could be covered with the random simulation. On the other hand, with **SIVA** we could cover 29 targets. The time taken by **Symbolic\_Solver** was only 0.7% of the total time. The slight decrease in the time taken (as compared to 1% without pruning) was due to the fact that the solver was called less often unsuccessfully.

We inspected the design to better understand the target node that eluded **SIVA**. Based on our intuition, we added 4 light-houses which we felt could help **SIVA** bridge the gap to the lone unreached target node. We then ran **SIVA** with the added information. It turned out that we could cover all the 30 targets. The time used by the **Symbolic\_Solver** was 4.6% of the total time. The increase was intuitive given the fact that it has to generate tests for light-houses at each state. In other words, we should use light-houses prudently as and only when required.

We tried to compare **SIVA** to conventional BDD-based formal verification, by using VIS system [4]. However, VIS could not proceed past reading in the design; when we attempted to perform BDD variable ordering, it spaced out. Closer inspection indicates that simply reading the design into VIS results in a 30Mb process (mainly due to inefficient network node data structure); in contrast **SIVA** uses only 2Mb to read in the design; the entire run fit in 5Mb.

All our experiments were performed on a PentiumII-266 with 64 MBytes of main memory running Redhat Linux 4.0.

## 5 Conclusion

In summary, we have developed a stand alone tool **SIVA** that strives to capture the best of both simulation and symbolic verification. Simulation is used to quickly sample the design behavior; symbolic methods are used to enable the transitions to rare cases, and to guide the simulation. The basic components – ATPG, BDD, simulation, indicator variables – are well known; our more significant contribution is the tight coupling of these approaches. It is our thesis that no

single technique can achieve what can be achieved by tightly integrating diverse approaches.

SIVA development is continuing on the following lines:

1. In the short term, we will apply SIVA to a number of ongoing class projects which are concerned with hardware verification. This experience should help us tune performance, and add features (both at the user interface and functional level). In the longer run, we plan to collaborate with design houses, perhaps through summer assignment.
2. Currently, the indicator variables are created manually. Ideally, this should be automated. The problem we have faced is that there are far too many candidates for indicator variables – control latches, HDL lines, FSM edges etc. We view automatic generation of indicator variables to be the most significant research area in this project.
3. The SAT-based ATPG engine in SIVA was developed internally. We are experimenting with other SAT solvers which may be more efficient.
4. The backtrack limits and BDD-size thresholds are set statically. We plan to make these dynamic depending on the target node. That is if a target node repeatedly fails to be justified (due to exceeding the limit), the solver should try harder on it.
5. We would like to automatically perform state pruning and prioritization. In the current set up, control latches have to be defined by the user; we would like to automate this. Additionally, we plan to experiment with state prioritization – when selecting a new state to control simulating, it may be beneficial to see how “rare” the state is.

While we have been critical of BDD-based model checking in this paper, we feel it still has an important role to play in design verification. More importantly, it is a core routine in general compositional verification paradigm that have been proposed [19]. Hence, continued research pushing the frontiers of BDD-based model checking is well founded.

## References

- [1] Felice Balarin and A. L. Sangiovanni-Vincentelli. An Iterative Approach to Language Containment. In *Proc. of the Computer Aided Verification Conf.*, June 1993.
- [2] R. Kurshan. Formal Verification in a Commercial Setting. In *Proc. of the Design Automation Conf.*, June 1997.
- [3] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On Combining Formal and Informal Verification. In *Proc. of the Computer Aided Verification Conf.*, July 1997.
- [4] R. K. Brayton et al. VIS: A System for Verification and Synthesis. In *Proc. of the Computer Aided Verification Conf.*, July 1996.
- [5] B. Chen, M. Yamazaki, and M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proc. European Conf. on Design Automation*, March 1994.
- [6] H. Cho, G. Hatchel, E. Macii, M. Poncino, and F. Somenzi. A State Space Decomposition Algorithm for Approximate FSM Traversal Based on Circuit Structural Analysis. Technical report, ECE/VLSI, Univ. of Colorado at Boulder, 1993.
- [7] S. Devadas, A. Ghosh, and K. Keutzer. An Observability-Based Code Coverage Metric for Functional Simulation. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1996.
- [8] David L. Dill. Embedded Tutorial: What's between Simulation and Formal Verification? In *Proc. of the Design Automation Conf.*, San Francisco, CA, June 1998.
- [9] A. El-Maleh, T. Marchok, J. Rajski, and W. Maly. Behavior and Testability Preservation Under the Retiming Transformation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, May 1997.
- [10] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage Directed Test Generation Using Formal Verification. In *Proc. of the Formal Methods in CAD Conf.*, November 1996.
- [11] Daniel Geist and Ilan Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1994.
- [12] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, 1981.
- [13] R. Ho and M. Horowitz. Validation Coverage Analysis for Complex Digital Designs. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1996.
- [14] Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David L. Dill. Architectural Validation for Processors. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.
- [15] Y. Hoskote, D. Moundanos, and J. Abraham. Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors. In *Proc. Intl. Conf. on Computer Design*, Austin, TX, October 1995.
- [16] Andreas Kuehlmann and Florian Krohm. Equivalence Checking Using Cuts and Heaps. In *Proc. of the Design Automation Conf.*, June 1997.
- [17] W. Lee, A. Pardo, G. D. Hachtel, J. Jang, A. Pardo, and F. Somenzi. Tearing Based Automatic Abstraction for CTL Model Checking. In *Proc. Intl. Conf. on Computer-Aided Design*, 1996.
- [18] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [19] K. L. McMillan. Verification of an Implementation of Tomaso's Algorithm by Compositional Model Checking. In *Proc. of the Computer Aided Verification Conf.*, Vancouver, BC, Canada, June 1998.
- [20] D. Moundanos, J. Abraham, and Y. Hoskote. A Unified Framework for Design Validation and Manufacturing Test. In *Proc. Intl. Test Conf.*, 1996.
- [21] J. Burch and V. Singhal. Tight Integration of Combinational Verification Methods. In *Proc. Intl. Conf. on Computer-Aided Design*, 1998.
- [22] K. Ravi and F. Somenzi. High Density Reachability Analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, Santa Clara, CA, November 1995.
- [23] R. Rudell. Dynamic Variable Ordering for Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1993.
- [24] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combination Test Generation using Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, September 1996.
- [25] U. Stern and D. L. Dill. Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In *Proc. of the Computer Aided Verification Conf.*, June 1998.
- [26] D. Xiang, S. Venkataraman, W. K. Fuchs, and J. H. Patel. Partial Scan Design Based on Circuit State Information. In *Proc. of the Design Automation Conf.*, Las Vegas, NV, June 1996.
- [27] C. H. Yang and D. L. Dill. Validation with Guided Search of the State Space. In *Proc. of the Design Automation Conf.*, June 1998.
- [28] UC Berkeley. [www.cad.eecs.berkeley.edu/~vis](http://www.cad.eecs.berkeley.edu/~vis).