Hybrid Verification Using Saturated Simulation*

Adnan Aziz
Dept. of Elec. and Comp. Engineering
Univ. of Texas at Austin

Jim Kukula Tom Shiple
Advanced Tech. Group
Synopsys, Inc.

Abstract — We develop a verification paradigm called *saturated simulation*, that is applicable to designs which can be decomposed into a set of interacting controllers. The core procedure is a symbolic algorithm that explores the space of controller interactions; heuristics for making this traversal efficient are described. Experiments demonstrate that our procedure explores substantially more of the controller interactions, and is more efficient than conventional symbolic reachability analysis.

1 Introduction

In this paper we are concerned with the problem of *design verification*; specifically, the problem of *invariant checking* over gate-level designs. Traditionally, designs have been verified by extensive simulation. A model is built (in software or hardware), and monitors may be added to check for bad behavior. Large numbers of test inputs are applied to this model. While offering the benefits of simplicity and scalability, simulation offers no guarantees of correctness; for large designs, the fraction of the design space which can be covered in this methodology is vanishingly small. Indeed, there are many examples of designs that passed extensive simulation, but were still found to contain bugs [5]. This has led to the proposal of "formal methods" for design verification.

The computational complexity of invariant checking on netlists is very high. In practice, many designs are well structured, and this can be exploited to devise heuristic procedures that perform well on specific classes of designs; in particular, BDDs have been used to successfully verify a large number of complex designs [7]. The primary limitation of BDD based approaches to FV is that is that for many designs, the resulting BDDs grow very large [7].

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; this is attested to by the complete lack of penetration of theorem-provers in the hardware verification community. Indeed, "falsification" is a more accurate description of the endeavor commonly called "verification".

Faced with the twin dilemmas of diminished coverage through simulation and the inability of symbolic methods to formally verify large designs, it is natural to ask how best to use symbolic methods to find bugs in designs. In this paper we provide an answer to this problem. It is to be stressed that the approach is not complete, i.e., not guaranteed to find a counterexample, if the design fails to satisfy its properties. However, our procedures are sound — all reported violations of the invariant are true bugs. Our goal in this work is to find more and different bugs more efficiently and cheaply than simulation or formal verification used in isolation.

1.1 Our Approach

Many designs consist of a set of interacting controllers. A common source of bugs in such designs is that the designer fails to take into account "corner-cases", which occur when two or more controllers are simultaneously in "exception-handling" states. An example of this is when a transmitter unit has a buffer overflow, and a receiver unit simultaneously times out. It is difficult for random simulation to find such corner conditions, since exceptions are rare. Similarly, it is very tedious for designers to craft tests that bring the design to such states. For these reasons, corner-cases are the single largest source of bugs in designs [8].

Our contribution is a procedure that performs a partial traversal of the state space while covering controller interactions. At each step, symbolic techniques are used to compute the full set of states reachable in one step from the current set. We then restrict our attention to a subset of the complete set of visited states; the subset is chosen heuristically to minimize the size of the symbolic representation while guaranteeing that for every pair of controllers, all states that the pair could be in remain in the subset. In this way we explore as many of the controller interactions as possible (which increases the likelihood of finding bugs), while controlling the BDD size explosion that takes place with a complete analysis.

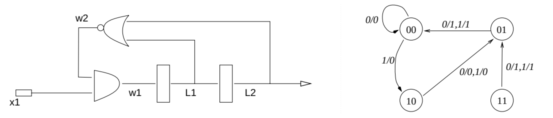
Previous Work

We have been influenced by a number of related works. The basic notion of saturated simulation was put forth by Yuan et al. [15]. However, this work is not directly applicable to designs which consist of a number of interacting controllers; they decompose the entire design into a monolithic controller and a monolithic datapath. For individual processors, such an approach works well. However, when dealing with a number of distributed components, the entire control space is very large, and their techniques fail.

The work which is perhaps most closely related to ours is that of Ravi et al. [14]. Their approach attempts to pick "high-density" subsets of the state sets encountered during reachability analysis. These sets are chosen so as to minimize the BDD representation, while keeping as many states as possible. In contrast, in our approach a subset is chosen in which all distinct control-state pairs are preserved; minimization is performed subject to this requirement. We believe that ignoring the high-level control structure of the design when doing subsetting, and using simply the number of states as a measure of the approximation, is a significant limitation of their approach.

Literature distributed by 0-In [10] at DAC 1997 also influenced us. Specifically, they underlined the importance of exploring control-pair interactions. Of course, we can only speculate about their algorithms, since no description of the underlying procedures is given.

Other less closely related works include those of Ashar



(a) Netlist **(b)** Finite State Machine

Figure 1: Examples: Netlist and FSM

and Malik [2], and McGeer et al. [12], who stress the importance of managing BDD size explosion, and the research of Ho et al. [8] and Hoskote et al. [9], who create test vectors using ideas from FV. Note that our approach is "liberal", i.e., does note guarantee that all bugs are found; "conservative" approaches include that of Cho et al. [6]. Details of these approaches are given in [1].

2 Background

In order to analytically reason about hardware, we first need to develop mathematical models for digital systems. We illustrate these models through the use of examples; refer to [7] for details.

Netlists and FSMs

Hardware can be modeled at the *structural* level using *netlists*, or at the *behavioral* level using *finite state machines* (FSMs). A netlist consists of an interconnected set of primary inputs, gates, and latches, as illustrated in Figure 1(a). Each gate has an associated Boolean function. A finite state machine can be represented by an edge-labeled directed graph, where the vertices correspond to *states*, and the labels are *input-output* pairs; an example is presented in Figure 1(b).

For a given netlist η, there is a natural way of deriving a finite state machine from it; states are evaluations to the set of latch variables, and the next-state/output functions are derived by composing the gate functions. This is illustrated in Figure 1.

Invariant Verification

A common verification problem for hardware designs is to determine if every state reachable from a designated set of initial states lies within a specified set of "good states" (referred to as the *invariant*). This problem is variously known as *invariant verification*, or *assertion checking*. Though conceptually simple, invariant checking can be used to verify all "safety" properties, making it a very general verification methodology[7].

Invariant checking can be performed by computing all states reachable from the initial states and determining that they all lie in the invariant. This can be achieved by traversing the STG of the design — successors of a state can be generated by iterating over all possible inputs. This procedure has a very high complexity; a design with n latches and m inputs may have as many as 2^n reachable states, and for each state, there are 2^m inputs to iterate over.

2.1 BDDs and Symbolic Invariant Checking

The Reduced Ordered Binary Decision Diagram [4] (BDD) is a graph based data structure that can compactly represent a large class of useful Boolean functions. Operations such as tautology checking, negation, conjunction, and existential quantification can be efficiently performed on BDDs.

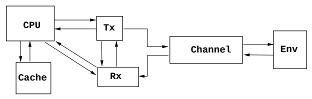


Figure 2: A design consisting of multiple functional units.

A set of states can be identified with a BDD, and the *Image* and *PreImage* operations can be performed directly on BDDs [13]. This is the basis for *symbolic model checking*; the primary limitation of this approach is the BDD size explosion problem.

3 Saturated Simulation

3.1 Introduction

Many designs can be separated into a set of interacting control units, as illustrated in Figure 2; furthermore, the designer is aware of this partition. As described in Section 1, a common source of bugs is when two controllers simultaneously enter exception handling states; the designer overlooks this possibility, and consequently the design may behave erratically. Since it is rare that controllers enter exception states, such bugs are hard to find with random simulation; indeed, a large effort is expended by designers trying manually to design test vectors that bring the design into precisely such states.

Note that for a design consisting of n controllers, each with m states, the total number of control-pair states is $O(m^2 \cdot n^2)$; this is a polynomial bound, as contrasted with the total number of states which is m^n. One way of visiting all reachable control-state pairs is by sequential ATPG. Of course this problem is as hard as computing the set of reachable states, and sequential ATPG is still in its infancy. In this section, we describe an approach we refer to as "control-pair saturated simulation", which heuristically attempts to symbolically explore as many pairs of control states as is computationally feasible.

In part, the feasibility of this approach follows from the fact that the construction of the BDD for the next-state logic can be restricted to the current set of states. This suggests that it may be possible to perform a "partial" reachability analysis, in which all distinct control pair states are preserved at each step.

3.2 Formalization

Let D be a design consisting of interacting control units $C_1, C_2, \ldots, C_n$. Let the set of state variables associated with controller i of the design be X_i. In this way the set of states corresponding to controller i is 2^{X_i}, and the global set of states is 2^X where $X = \bigcup_i X_i$.

Definition 1 Let A be a set of states. A subset $\hat{A}$ of A is *control-pair saturated* with respect to A if

$$(\forall i)(\forall j)\,(\forall \alpha_i \in 2^{X_i})(\forall \alpha_j \in 2^{X_j})$$
$$\Big[(\alpha_i, \alpha_j) \in \exists Y_{ij}(A) \;\rightarrow\; (\alpha_i, \alpha_j) \in \exists Y_{ij}(\hat{A}) \Big]$$

where $Y_{ij} = X - (X_i \cup X_j)$.

Intuitively, $\hat{A}$ is a control-pair saturated subset of A if every pair of control states occurring in A occurs in $\hat{A}$. Thus control-pair saturated subsets of A preserve all the control-pair states present in A. Heuristically, a minimal control-pair saturated subset of A is a good representative set — it

```
 /* $A$ --- BDD for set over variables $V$.  */
 /* $V' \subset V$ --- variables being cprojected.    */
BDD_t function BDD_cproject($A$, $V'$) {

    $v$ := topVar(A);
    if ($v \not\in V'$) {
        return $v \cdot BDD\_cproject(A_v) \ + \ \bar{v} \cdot BDD\_cproject(A_{\bar{v}})$;
    }

    $T := \exists(V' - \{v\})A_v$;
    if BDD_Equal($T$, $BDD\_One$) {
        return $v \cdot BDD\_cproject(A_v, V')$;
    }
    else if BDD_Equal($T$, $BDD\_Zero$) {
        return $\bar{v} \cdot BDD\_cproject(A_{\bar{v}}, V')$;
    }
    else {
         /* In cproject_min $\overline{T}$ is replaced by $1$ */
        return $v \cdot BDD\_cproject(A_v, V')$
                   $+\bar{v} \cdot \overline{T} \cdot BDD\_cproject(A_{\bar{v}}, V')$;
    }
}
```

Figure 3: Pseudocode for `cproject` operator.

includes all the distinct control-pair configurations in A, and is as small as possible (in the sense of cardinality). An example illustrating this definition is given in [1].

We now address the problem of computing control-saturated subsets of A. Let f be a Boolean function on variables X. Lin et al. [11] presented an efficient algorithm (referred to as the `cproject` operator) that takes a BDD for f and a subset $X' \subset X$ of the variables, and returns a BDD for a function f^* which has the property that

1. for any assignment v to the variables in $X - X'$, so that $f(v) = 1$, there is exactly one valuation v' which agrees with the valuation v over the variables in $X - X'$ so that $f^*(v') = 1$, and furthermore

2. for all u, $f^*(u) = 1 \Rightarrow f(u) = 1$.

Pseudo-code for the `cproject` operator is given in Figure 3; an example of the application of `cproject` is given in [1]. Since sets can be represented by their characteristic functions, we will freely apply the `cproject` operator to sets. Observe that $\bigcup_{i,j}$ `cproject`$(A, (X_i \cup X_j))$ is a control-pair saturated subset of A (though not necessarily minimal, becaue of the union).

In Figure 4 we sketch a simple symbolic procedure for invariant verification. Reachable states are iteratively computed; at each step, a control-pair saturated subset ($satImg$) of the image of the current set of states set is computed using the *BDD_cproject* operator. The union of this set is taken with the current set of states (A) and is used as the next set of states. Much work has been done on improving the efficiency of the *Img* operator [7], and we can directly use it.

This procedure may reach a fixed point without visiting all reachable states because at each step only a subset of states is being explored. However, if a state not in the invariant is reached, this is guaranteed to indicate a bug.

```
 /* $A$ --- initialized to BDD for init states.  */
 /* $C$ --- list of state vars in each controller.    */
 /* $G$ --- BDD for invariant.    */
BDD_t function Cntrl_Pair_Sat_Sim($A$, $C$, $G$) {

    if (BDD_Intersects($A$, BDD_not($G$))  /* Invariant fails!!  */
        assert FAIL;

    ImgA := BDD_Img($A$);
    satImg := BDD_Zero();
    for_each($i, j$) {
        satImg := BDD_Or(satImg, BDD_cproject( ImgA, $\bigcup_{k \neq i,j} C[k]$ ) );
    }

    $R$ := BDD_Or( $A$, $satImg$ );
    if (BDD_Equal($R$, $A$))
        return $R$;

    return Cntrl_Pair_Sat_Sim($R$, $C$, $G$);
}
```

Figure 4: Control-pair saturated simulation.

3.3 Extensions to saturated simulation

Biasing

The implementation of the cproject operator in Fig 3 is "biased towards 1", i.e., when presented with a choice for a projection variable, it sets it to 1. This biasing can result in dropping portions of the state space that may be significant (e.g., branch on zero). We overcome this by computing the union of two subsets; one biased towards 0, and one biased towards 1. While no longer minimal, the resulting set has cardinality no more than twice of the original set computed by `cproject`. More generally, we can define a `cproject_random` operator, wherein at each level of the recursion the bias for each variable is selected at random. The results of `cproject_random` can be added to the 0/1 biased subsets to obtain a rich, yet sparse subset.

BDD Minimization

Consider the expression for the final case of the *BDD_cproject* function listed in Figure 3:

$$v \cdot BDD\_cproject(A_v) \ + \ \bar{v} \cdot \overline{T} \cdot BDD\_cproject(A_{\bar{v}})$$

Replacing $\overline{T}$ by 1 results in a function which computes a superset of the result of *BDD_cproject*. Intuitively, since the expression is simplified, we reason that the BDD should also be simpler. We refer to the resulting operator as `BDD_cproject_min`; heuristically, it should give a larger set (in the sense of cardinality) with a smaller BDD.

Control-pair Edge Saturated Simulation

An extension to obtain enhanced coverage is to perform a partial reachability analysis and at each step pick a subset of the image that preserves all "controller transitions" to the image from the current set [8, 9]. Yuan et al [15] show how to compute minimal control-edge saturated sets by augmenting the design: for every control latch x_C, add a new latch x_S which "shadows" x_C, that is, the next state of x_S is the present state of x_C; hence, control-edge pair saturated simulation can be reduced to control-state pair saturated simulation.

4 Experimental Results – Saturated Simulation

We coded the routines described in the previous section as part of the VIS program [7]. We report experimental results on the viper microprocessor, which is distributed with the VIS system [3]. Viper is approximately 4000 gate-equivalents. It contains 252 latches; from these, we identified 6 control units, containing between 4 and 12 latches, for a total of 42 control latches. The variable ordering was derived by building the BDD for the decomposed transition relation and performing reordering; dynamic reordering was enable throughout.

In the previous section, we described several extensions of the basic saturated simulation procedure laid out in Figure 4. Some of these were independent (such as the use of control-pair edges, and the use of `cproject_min`); consequently, there are an exponential number of possible experiments. We performed a representative set, detailed results are given in [1]; we summarize them below. In particular, we experimented with the following —

Comp-Reach: Complete BDD based reachability analysis.

Sat-Reach: Saturated reachability analysis as in Figure 4.

0-Bias: *Sat-Reach*, with 0-biased *cproject* (§ 3.3).

Rnd-Bias: *Sat-Reach*, with random *cproject* (§ 3.3).

Min-Sat: *0-Bias*, with *BDD_cproject_min* instead of *BDD_cproject* (§ 3.3).

Min-Rnd-Sat: *Sat-Reach*, with randomly biased *BDD_cproject_min*.

All experiments were performed on a DEC AlphaServer 2100A 5/250 with 1Gb of memory.

Summary of results

1. **Comp-Reach** spaced-out after four iterations of reachability, after having visited 224,326 control pair states (CSPs).

2. **Sat-Reach** reached a fixed point in 64 iterations (144.6 seconds), having found 353,949 CSPs; peak BDD size was 5,512 nodes.

3. **0-Bias** reached a fixed point in 67 iterations (1269.5 seconds), having found 633,603 CSPs; peak BDD size was 38,033 nodes.

4. **Rnd-Bias** spaced-out in 15 iterations (2565.4 seconds), having found 1,152,080 CSPs; peak BDD size was 2,755,343 nodes.

5. **Min-Sat** reached a fixed point in 68 iterations ($4,951.8$ seconds), having found 1,063,720 CSPs; peak BDD size was 138,253 nodes.

6. **Min-Rnd-Bias** spaced-out in 15 iterations ($9,423.8$ seconds), having found 1,649,070 CSPs; peak BDD size was 3,308,410 nodes.

Comp-Reach exhibited unstable behavior — the first four iterations took 3.2 seconds, and the largest BDD encountered was $4,019$ nodes; the example spaced out on the fifth iteration.

All variants of saturated reachability were able to go through substantially more iterations. The 0-biased cproject routine found 80% more control pair states than the 1-biased version; we conjecture this is because there is more interesting behavior which can be reached from states where the datapath registers contain 0's.

Among the nonminimized routines, the random biased cproject discovered the most control state pairs, a total of 1.15×10^6, which is five times as many as complete analysis, and two/three times better than 0/1-biased analysis.

Somewhat to our surprise, the `BDD_cproject_min` operator usually gave larger BDDs than `BDD_cproject`. However, there was benefit in terms of exploring more control pair states: **Min-Rnd-Sat** found 1.65×10^6 control pair states.

For comparison we ran cycle simulation with random inputs for 100,000 seconds; we could not compute the number of control-state pairs, but the total number of states found was an order of magnitude less that the number of control-state pairs reached in any of the above experiments.

5 Conclusion

In summary, our primary contribution in this paper is a symbolic procedure for exploring the set of control pair states. The various heuristics developed in the course of this paper result in substantial improvement over complete BDD-based reachability and cycle simulation. The resulting procedure is robust with respect to time and memory, and requires only that the user specify the different controllers in the design.

We believe that it is fair to say that existing symbolic verification techniques, though complete, are limited. Coping with BDD blowup requires considerable expertise; we suggest more research should be performed on exploring how to best use computational resources.

In the future we plan to continue exploring ways in which to combine formal and informal verification; one possibility is adaptive simulation, in which random search is combined with backtrack. We are also looking at applications in software verification.

References

[1] `www.ece.utexas.edu/~adnan/publications/AKS-DAC-98.ps`.

[2] P. Ashar and S. Malik. Fast Functional Simulation Using Branching Programs. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1995.

[3] UC Berkeley. `www.cad.eecs.berkeley.edu/~vis`.

[4] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[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*, pages 132–136, March 1994.

[6] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A Structural Approach to State Space Decomposition for Approximate Reachability Analysis. In *Proc. Intl. Conf. on Computer Design*, October 1994.

[7] R. K. Brayton et al. VIS: A system for Verification and Synthesis. In *Proc. of the Computer Aided Verification Conf.*, July 1996.

[8] 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.

[9] 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.

[10] 0-in Inc. `www.0in.com`.

[11] B. Lin and R. Newton. Implicit Manipulation of Equivalence Classes Using Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer Design*, Cambridge, MA, October 1991.

[12] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1995.

[13] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] K. Ravi and F. Somenzi. High Density Reachability Analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, Santa Clara, CA, November 1995.

[15] 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.