An Efficient Assertion Checker for Combinational Properties

Gagan Hasteer  
Dept. of Computer Science  
University of Illinois  
Urbana, IL-61801

Anmol Mathur  
Silicon Graphics  
2011 N. Shoreline Blvd.  
Mountain View, CA 94043-1389

Prithviraj Banerjee  
Electrical and Comp. Engineering  
Northwestern University  
Evanston, IL-60208

Abstract

Formally verifying properties of signals in a circuit has several applications in an equivalence checking based formal verification flow. In a hierarchical design, functionality is divided across blocks. This necessitates the use of constraints on input signals of a block to avoid false negatives. Validating such input constraints requires assertion checking at the outputs of modules generating the constrained signals. In this paper, we present an efficient assertion checker for combinational properties which avoids the BDD explosion problem by finding an optimal intermediate correlation free frontier. It has been successfully used in an industrial setting to uncover a number of bugs.

1 Introduction

As the functional complexity of chips has grown, it has become impractical to reason about their correctness manually. Traditional techniques for verification of a design against its specification such as simulation or emulation-based methods can leave corner cases untested, and can be prohibitively time consuming. This has resulted in a rapid increase in interest in techniques for formally proving the correctness of designs before committing them to silicon [1]. In a typical design flow, a design is decomposed into several functional blocks, each of which is further divided into smaller blocks for ease of design. The specification of a module is written in a hardware description language (such as Verilog) at the register transfer level (RTL). This specification is then used by the designers to design a circuit that implements the functionality specified in the RTL. Equivalence checking is used in such a flow to compare the specification against its implementation.

Since functionality is partitioned across blocks, and equivalence checking is done on these blocks independently, it becomes necessary to specify properties of signals that cross block boundaries to the equivalence checking tool. For example, Figure 1 shows a scenario where block A decodes some signals and block B uses these decoded signals. When block B is formally verified, the equivalence checker needs to be informed that the decoded signals are one-hot since the behavior of block B in conditions where these signals are not one-hot is irrelevant and may be different in the RTL specification and the implemented circuit. This may result in a false negative. In order to avoid such false negatives, constraints need to be specified on input signals of a block before performing equivalence checking. However, if the constraints imposed at the inputs of a block are not independently verified to be true at the outputs of the blocks that generate them, there is a potential hole in the formal verification process. The problem of verifying constraints used in equivalence checking is referred to as the constraint validation problem.

![Figure 1: Interaction between modules and distribution of functionality across modules necessitates constraints.](image-url)

We address the problem of formally verifying functional properties (assertions) on sets of signals in a circuit. This tool has several applications in the formal verification flow using equivalence checking. Constraints on the inputs of a block are translated into assertions on
the outputs of the blocks that generate these signals and then our assertion checker can be used to formally verify that these output assertions are true. A constraint is valid if and only if the corresponding assertion is true.

Recently, many researchers have focused their attention on efficient **combinational equivalence checking** [2, 3, 4, 5]. In some sense checking for combinational equivalence is a special case of assertion checking, where the assertions state the equality of the corresponding output and latch pairs of the two circuits. Binary Decision Diagram (BDD) [6] based approaches have been used widely for equivalence checking. These techniques suffer from the memory explosion problem for BDDs. Powerful input ordering heuristics have been developed [4] but these methods are not robust. Structural similarity of the two circuits being compared has been exploited to reduce the complexity of the problem in [5]. The authors point out the problem of false negatives but do not provide a concrete procedure to handle it. In [2] the author handles the BDD explosion problem by choosing a set of intermediate signals as the support of the BDD. However, his heuristics for selecting a good basis do not guarantee that the equivalence of the two signals will be established. The heuristic has an expensive adjustment phase to avoid false negatives.

In this paper we describe an efficient assertion checker (PropCheck) that has been successfully used to verify assertions on a variety of industrial circuits at Silicon Graphics (SGI). It is being used in conjunction with a formal verification environment to check constraints at module interfaces. Our main contributions are novel techniques to avoid the BDD explosion problem. These include using intermediate signals for BDD support and modeling assertions efficiently. PropCheck uses a depth first search based **frontier finding** heuristic. It finds a set of intermediate signals as a basis for building the BDD. The false negative problem is avoided altogether by ensuring that the vertices chosen in the frontier are independent. The intermediate frontier guarantees to prove or disprove an assertion if none of the frontier signals are stuck (constant 1 or 0).

The rest of this paper is organized as follows: In §2, we formulate the assertion checking problem and introduce some concepts. §3 describes the assertion checking algorithm in detail. §4 discusses the performance of the assertion checker on various applications which warrant its use. We conclude the paper with a discussion on future work in §5.

## 2 Problem Formulation

An **assertion** on a set of signals (or nodes) of a circuit is a relation which should be satisfied by these nodes in a correct implementation of the circuit. The nodes on which the relation is defined are called **assertion nodes**.

The relation can be expressed in terms of a boolean function defined on the assertion nodes. Henceforth, we refer to an assertion and its boolean function interchangeably whenever the interpretation is clear by context. **Assertion Checking** is the process of formally verifying the truth or falsity of an assertion.

A circuit $C$ can be represented as a directed graph $G = \langle V, E \rangle$. Each gate of $C$ is a vertex in the graph $G$. A directed edge $(v_1, v_2)$ exists between $v_1, v_2 \in V$ if the output of gate $v_1$ fans into the input of the gate $v_2$. Note that the direction of the edge is opposite to the signal flow. We denote the fanin and fanout cones of a gate $v$ by $FI(v)$ and $FO(v)$ respectively.

Consider a depth first search (DFS) of a directed graph $G$ starting from a source vertex $s$. Common ancestors of two vertices $u$ and $v$ are all the vertices in the intersection of the paths $s \sim u$ and $s \sim v$ in the DFS. **Least Common Ancestor (LCA)** $w$ is a common ancestor such that the paths $w \sim u$ and $w \sim v$ are disjoint. Intuitively, its the common ancestor which is closest to the two nodes $u$ and $v$.

## 3 Algorithm

We prove an assertion by constructing a BDD for an equivalent boolean expression. The circuit description is read in as a directed graph as explained in §2. For each assertion an equivalent boolean expression is generated. A BDD is built for this expression using primary inputs of the circuit as support variables of the BDD. If the function is a tautology the BDD evaluates to constant 1 and the assertion is true. Otherwise an input assignment which violates the assertion is reported.

There are some efficiency issues and practical considerations which guided the design of PropCheck. Some BDD based applications are notorious for aborting due to memory explosion. For making assertion checking memory and time efficient we concentrated on the following refinements to the basic algorithm.

- **Refinement 1:** Using intermediate signals as BDD support.
- **Refinement 2:** Efficient modeling of assertions.

### 3.1 Refinement 1: Using Intermediate Signals

A look at Figure 2 shows that the truth of the assertion (\textit{COMPL B C}) can be proved by just evaluating $Ckt1$. The large logic embedded in $Ckt2$ which is the fanin cone of signal $A$ need not be considered. Thus, if $A$ is used as the support of the BDD for this assertion, unproductive work can be saved and the BDD memory explosion problem avoided. Intuitively, the set of intermediate signals that can be used to prove or disprove
an assertion forms a frontier. By finding such a frontier we can restrict the construction of the BDD for the assertion to the intermediate signals in the frontier. This allows us to avoid constructing BDDs for signals which do not need to be expanded for verifying the assertion (e.g. signal A in the above example). This can result in significant savings in BDD memory usage as well as run time of the algorithm.

Formally, a correlation free frontier (CFF) for an assertion is a set of nodes which is needed as support of the BDD to prove or disprove the assertion. The nodes in the CFF are called frontier nodes. \( F = S_1, S_2, ..., S_n \) is a CFF for an assertion \( G(A_1, A_2, ..., A_k) \) if and only if

1. Each frontier node belongs to the fanin cone of some assertion node.
2. Each assertion node can be defined as a function of frontier nodes (or a subset thereof).
3. The fanin cones of the frontier nodes are disjoint.

Figure 3 illustrates the above properties of a CFF. In general there can be many CFFs for proving or disproving an assertion. Consider an assertion \( G(A_1, A_2, ..., A_k) \) and its CFF \( F = \{ S_1, S_2, ..., S_n \} \). Cost of a frontier with respect to an assertion is the number of nodes in the circuit which are in the fanout cone of at least one frontier node and in the fanin cone of at least one assertion node. Thus, cost of a frontier is equal to the number of gates or internal nodes which need to be evaluated for constructing the BDD for the assertion. The most expensive CFF for any assertion is the set of primary inputs. We are interested in finding the cheapest CFF. It has the following advantages:

1. It minimizes the cost of building the BDD.
2. In a circuit with sequential loops we may find a CFF before the loop, enabling us to prove a property that would not be provable if we went all the way to primary inputs.
3. Proving a property at a frontier that has a path with \( k \) flip-flops from a frontier signal to an assertion signal, implies that the property is proven only for all time \( k \) cycles after the start of the system in an arbitrary state. Consequently, proving a property at an earlier frontier also proves a stronger property.

Figure 3: An example of CFFs for a circuit.

**Theorem 1:** Truth or falsity of an assertion defined on a subset of signals of a circuit can always be verified by using any of its CFFs as the support for its BDD provided the frontier nodes are not constant 0 or 1

**Proof:** Omitted due to lack of space. Refer [7].

### 3.2 Finding the Cheapest CFF

Figure 4 gives an overview of our algorithm for finding the cheapest CFF for an assertion. It exploits the fact that false negatives can be caused only due to reconvergent fanouts (or due to a stuck node). The node (signal) from which these reconvergent fanouts originate will have at least one cross edge incident on it in graph \( G \). The algorithm maintains a field (frontier flag) with each vertex of the graph. This field represents whether a vertex is a candidate for the CFF or not. Initially, the frontier flags of all the vertices are switched on. In the first pass (DFS1), the algorithm detects all the cross edges and disables the frontier flags of all signals which occur on either of the two reconvergent paths for each cross edge. In the second pass (DFS2), as soon as a vertex whose frontier flag is on is seen, it is selected for the CFF. Intuitively, such a node does not lie on any reconvergent paths. So this node has the potential to assimilate the effect that its fanin cone has on the assertion. This makes it an ideal candidate for the CFF. Further, in DFS2 we do not walk over any vertex whose frontier flag is on. This gives us the cheapest CFF.

All latches are correlated by the common clock signal. If the cheapest CFF is located beyond the first level of latches as seen from the assertion nodes, then the CFF would be pessimistically chosen beyond all the latches in the fanin cones of the assertion nodes. Similarly the constant signals may introduce some false correlations. A preprocessing phase traces the clock tree and marks its leaves. Constant nodes are also located by a breadth first constant propagation. These nodes are ignored while finding the cheapest CFF. By treating latches as buffers and ignoring the clocks, timing information is lost. Hence a false assertion may be verified as true as shown in Figure 5. A CFF is said to be safe...
Theorem 2: Algorithm Find_Frontier() finds the cheapest CFF for the given assertion

Proof: Omitted due to lack of space. Refer [7] □

PropCheck detects sequential loops and always finds a frontier in front of the loop if such a frontier exists. In general, a property which depends on a sequential loop cannot be proved by combinational techniques. Full blown model checking is required to analyze the state machine behavior and verify such properties. For such assertions a loop free CFF does not exist. PropCheck uses a combination of techniques to handle sequential behavior and give useful information to the designer. It unravels a sequential loop once and finds a frontier at the tail of the loop. It then tries to prove the assertion failing which it prints out the loop and suggests the designer to pick one of the vertices of the loop as a variable for BDD support. This is done using the VAR directive in the assertion file. If PropCheck still fails to prove the assertion, it gives the designer an assignment to the frontier variables which violate the assertion. Readers are referred to [7] for a detailed description of the techniques used to handle sequential behavior.

3.3 Refinement 2: Modeling of Assertions

For some types of assertions, converting them into an equivalent boolean expression is costly. For example: \( \text{ONE} \rightarrow \text{HOT} \ A_1 \ A_2 \ldots \ A_n \) has a boolean expression \( \vee_{i=1}^{n} (A_1 A_2 \ldots A_i A_{i+1}) \) which has length \( O(n^2) \). After building the BDDs for \( A_1, A_2, \ldots, A_n \), \( O(n^2) \) BDD operations have to be performed to obtain the BDD for this boolean expression, even though the size of the final BDD is \( O(n) \). Almost-k-of-n assertions can be modeled...
Table 1: Types of assertions handled by PropCheck

\begin{tabular}{|c|c|c|}
\hline
Assertion type & Boolean expression & Atmost k-out-of-n model \\
\hline
EXCLUSIVE A B C & $ABC + ABC + ABC + ABC$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
ONE HOT A B C & $\bar{A}B\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C}$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
COMPL A B & $\bar{A}B + \bar{A}B$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
ONE,COLD A B C & $\bar{A}BC + \bar{A}BC + AB\bar{C}$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
IMPLIES A B & $\bar{A} + B$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
EQUIV A B & $AB + \bar{A}\bar{B}$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
BOOLEAN F(A,B,C) & $F(A,B,C)$ & \begin{align*} & \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,A,B,C) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,A,B) \\
& \text{AT MOST}(0,A,B,C) \cdot \text{AT MOST}(1,\bar{A},b,c) \\
& \text{AT MOST}(0,A,B) \cdot \text{AT MOST}(1,\bar{A},b) \\
& - \end{align*} \\
\hline
\end{tabular}

(A MOST(c, A1, A2..., Am) represents the template atmost-c-of-n with signals A1, A2..., Am as arguments.)

Figure 7: BDD templates for atmost k-of-n constraints (k=0,1)

using a compact BDD template with $O(kn)$ nodes [8]. Figure 7 shows examples of atmost-0-of-n and atmost-1-of-n assertion templates for $n = 5$. As shown in Table 1, most of the common assertion classes can be expressed as a combination of these two templates. Only $O(n)$ BDD operations (ite calls) are required for building such templates. The use of these templates allows us to avoid the generation of large boolean expressions thus avoiding possible blowups in the intermediate BDD size. BDDs for the assertion nodes are composed with the corresponding templates in accordance with Table 1. If the BDD thus obtained is constant 1 then the assertion is true otherwise it is false.

4 Results

PropCheck is being used to verify the input/output interface of the blocks of a microprocessor being developed in SG1. It is also being used to verify the assertions generated by other tools. In Table 2 we report the results obtained from running PropCheck on some test cases for input/output constraint validation problem. Column 1 gives the block on which the assertion checking was carried out and the number of assertions which were verified. Columns 2 and 3 show the improvements obtained from successive refinements described in §3. For each of these, the total time required to run PropCheck (in seconds) and the maximum number of BDD nodes required over the entire application is reported. The last column reports the number of assertions identified as true or false, and the number of bugs found in the implementation of the block (T/F/Bgs).

9 out of the 11 circuits could not be verified using the naive algorithm. The reason is that most circuits had loops in the fanin cone of the assertion nodes. Hence the BDD for the assertions could not be built using primary inputs as the support variables. Refinement 1 is able to find a CFF before encountering a sequential loop for all the test cases. Thus, we were able to verify the constraints for all the blocks successfully using Refinement 1. Using templates instead of boolean expressions in Refinement 2 lead to lesser BDD memory usage as shown by examples Ckt1 and Ckt2. Our algorithm follows a simple DFS ordering heuristic for the support variables of the BDD. This is inherent in the computation of a CFF in DFS2.

Some very interesting bugs were found using PropCheck for constraint validation. As explained in §1 the input constraints of a module define the valid input space for checking the equivalence between its RTL specification and transistor netlist implementation. Most of the bugs were found for cases where this space was over-constrained. As a result the equivalence checker verified the block for only a subset of the valid input space and the blocks were falsely assumed to be verified and promoted in design (Ckt1, Ckt8, and Ckt9). Using PropCheck the constraints which were over-restrictive were proved false and reported to the designers. The designs were corrected and passed equivalence checking under the correct relaxed constraints. In case of Ckt4, although an over-specified constraint was found, the block’s implementation passed equivalence checking under the new relaxed constraint. Ckt7 had two constraints, both of which were sequential in nature and required the state machine analysis of the circuit. These constraints were not proven by PropCheck although...
Table 2: Performance of PropCheck on Input/Output Constraint Verification.

<table>
<thead>
<tr>
<th>Block</th>
<th>Refinement 1</th>
<th>Refinement 2</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Size</td>
<td>Time</td>
</tr>
<tr>
<td>Ckt1 (13)</td>
<td>17.3</td>
<td>660</td>
<td>16.9</td>
</tr>
<tr>
<td>Ckt2 (5)</td>
<td>59.5</td>
<td>40712</td>
<td>56.8</td>
</tr>
<tr>
<td>Ckt3 (3)</td>
<td>25.1</td>
<td>881</td>
<td>24.7</td>
</tr>
<tr>
<td>Ckt4 (1)</td>
<td>1.6</td>
<td>194</td>
<td>1.3</td>
</tr>
<tr>
<td>Ckt5 (3)</td>
<td>8.6</td>
<td>186</td>
<td>8.6</td>
</tr>
<tr>
<td>Ckt6 (1)</td>
<td>3.1</td>
<td>131</td>
<td>2.9</td>
</tr>
<tr>
<td>Ckt7 (2)</td>
<td>30.3</td>
<td>307</td>
<td>29.4</td>
</tr>
<tr>
<td>Ckt8 (1)</td>
<td>1.3</td>
<td>51</td>
<td>1.2</td>
</tr>
<tr>
<td>Ckt9 (1)</td>
<td>1.3</td>
<td>51</td>
<td>1.2</td>
</tr>
<tr>
<td>Ckt10 (1)</td>
<td>0.7</td>
<td>37</td>
<td>0.7</td>
</tr>
<tr>
<td>Ckt11 (2)</td>
<td>6.5</td>
<td>285</td>
<td>6.12</td>
</tr>
</tbody>
</table>

some information was given about the sequential loop on which they depended.

Ckt1, which produces the control signals for a data-path circuit (DataPath1) deserves special attention. DataPath1 had passed equivalence checking. It assumed 13 constraints on its inputs describing relations between the left and right bit and byte shifts. For example

ONE_HOT shift1[3:1] shiftr[3:1] shift0

says that exactly one of the shift signals is always on. Six of these constraints were proven false when verified by PropCheck on Ckt1 including the one above. A bug was discovered in the decoding of the byte shift signals in Ckt1. When these constraints were corrected the RTL description of DataPath1 was again checked for equivalence against its transistor netlist. Under the relaxed constraints DataPath1 failed equivalence check. A bug was found in DataPath1's netlist due to a wrong connection. This was a corner case and had escaped the equivalence checker due to an over-specified constraint.

Although the RTL is extensively verified using simulation of instruction sequences, most of the above bugs would not have been located by these simulations because they were bugs in the implementation rather than the specification. They might have been located later in the design cycle when the transistor netlist for the entire microprocessor would be ready and simulations carried out at the netlist level. Thus, PropCheck has proved useful in catching bugs earlier in the design process.

5 Conclusion

The modular design of big chips and other practical constraints leave some gaps in formally verifying an implementation against its specification. These gaps have to be filled by application specific tools like PropCheck. PropCheck has been successfully used in SGI to uncover a number of bugs. It handles the BDD explosion problem by using intermediate variables as BDD support.

If a circuit has sequential loops PropCheck can still verify an assertion because the CFF may not encounter the loop at all. We plan to extend the tool to handle assertions based on the sequential behavior of the circuit. It will also be worthwhile to apply the concept of intermediate CFFs to combinational equivalence checking where the BDD explosion problem is prominent.

References