Journal on Satisfiability, Boolean Modeling and Computation 5 (2008) 111–132

# A Probabilistic and Approximated Approach to Circuit-Based Formal Verification

**Sergio Nocco** sergio.nocco@polito.it
**Stefano Quer** stefano.quer@polito.it
*Dipartimento di Automatica e Informatica,*
*Politecnico di Torino,*
*Torino, Italy*

## Abstract

In both the hardware and the software domains, non-canonical circuit-based state set representations have recently been the subject of intensive investigations. One of the limiting factors of these representations has been the difficulty to control their size during key operations. For example, existentially and universally quantifying a variable implies doubling the circuit size in the worst case.

In this paper, we present a probabilistic approach to keep under control the size of circuit-based representations when manipulating them. Every time a formula is becoming too cumbersome, we estimate it instead of building the exact result. The nature of the estimate, i.e., under- or over-approximation, depends on the problem that is being computed. The key idea of this process is to boost the expressiveness of the formula, delivering a dense representation, i.e., a formula compact in size but more expressive for the given verification problem.

Experimental results show decisive reductions in terms of circuit size, and an increase in terms of density, i.e., the ratio between the cardinality of the on-set of a formula and the size of its representation. We applied the strategy to Bounded Model Checkingm, and circuit-based backward Unbounded Model Checking. We present experimental results from applying the approach to hard-to-solve verification instances. We observed speedups of more than one order of magnitude in some cases.

Keywords: *and-inverter graph, Boolean satisfiability, SAT-solvers, model checking, density, controllability*

*Submitted February 2007; revised March 2008; published June 2008*

## 1. Introduction

Binary Decision Diagrams (BDDs) [5] have long been the key instrument for representing and manipulating Boolean functions. Being a canonical structure, BDDs suffer from the so called space-explosion problem, i.e., they often become too complex to be efficiently used.

Ravi et al. [13] addressed this problem proposing the "high-density" paradigm. Given a BDD representing a set of states, density was defined as the ratio between the number of minterms in the represented formula and the size of the corresponding BDD in terms of nodes. The main idea of their procedure was to increase the density of the state set representation, in order to obtain a better characterization of the state space. Their algorithm performed a breadth-first visit of the state space [6] as long as the BDDs used in image computation were small. When they became too large, a dense subset was retained. This set was generated trying to maintain the highest possible number of minterms (i.e., states) represented with the smaller possible number of BDD nodes. Such a subset was used to proceed in the traversal. Once a dead-end was reached, i.e., a stage where no new states were added, the image of the entire set was computed to recover from previously lost states.

More recently, SAT-based verification techniques have been shown to be more robust and scalable than symbolic model checking methods based on BDDs. SAT tools usually work on a Conjunctive Normal Form (CNF) representation. It, in turn, is usually derived from non-canonical circuit-based representations such as And–Inverter Graphs (AIGs) [10], Boolean Expression Diagrams (BEDs) [15], and Reduced Boolean Circuits (RBCs) [1]. AIGs, for instance, can be adopted as underlying structure in many verification methods. In Bounded Model Checking (BMC), the formula under check is often represented through AIGs, and then stored into CNF just before the satisfiability step. Similarly, pre-image computations, involving the existential quantification of input and state variables, can be performed on an AIG representation [8].

Albeit AIGs (as well as BEDs and RBCs) are a non-canonical structure, their size is still a key problem in many common procedures. For example, existentially and universally quantifying a variable implies doubling the circuit size in the worst case. Analogously, in the BMC procedure the size of the formula grows linearly with the bound of the problem.

In this paper, we present a probabilistic approach to keep under control the size of circuit-based state set representations. The core idea is to apply the high-density paradigm on AIG-based state set representations. We define the *representativity* of an AIG as the ratio between the number of minterms represented and the size of the AIG in terms of gates. Whenever an AIG is too large, we try to increase its representativity. To do that, we use a set of heuristic measures to decide how to simplify the AIG. While BDD heuristics are essentially based on the BDD structure and its canonicity, we resort to probability and testability measures. We first compute the probability to have any node equal to 1, or, similarly, we compute the controllability value of any node to 1. These measures indicate the difficulty of setting a line to a certain value. Then we cut-out all the nodes having a small probability (or high controllability), as these nodes are considered to have a small influence on the on-set of the characteristic function of our state set representation. When cutting-out part of the AIG representation, we have to decide whether to produce a subset or superset of the original state set. This choice depends on the given verification problem.

We apply our approximation technique to BMC and to the circuit-based property verification technique presented in [8]. Both sub-setting and super-setting may have a larger variety of applications. For example, super-setting, i.e., producing over-approximations of the exact representation, can be applied within abstraction-and-refinement techniques. In this field, it can possibly deliver abstract models on which to prove the properties, while the original (concrete) design is used to refute them. Moreover, super-setting can also be applied in interpolant-based verification to produce a tighter estimate of the so-called $k$-adequate over-approximate images.

Notice that in our backward verification procedure, as in the original paper [13], we eventually have to recover from previously lost states. This problem is alleviated by the observation that a complete traversal is not always required for property verification. This mixed approach has two advantages:

1. It typically reaches more states with smaller memory resources, since it is not constrained by a fixed sequence of states as in the breadth-first search.
2. Even when the number of states are comparable, it can reach farther away from the initial states, thereby producing a more uniform sampling of the reachable space.

Notice that the two proposed methods, the probabilistic BMC and unbounded backward verification, do not produce false results as they always end-up with an exact search. The increased efficiency steams from mixing under-, over- and exact-searches in an accurate way. While the second strategy is also complete, i.e., it is able to prove true properties, the first one shares with standard BMC its debugging characteristic, i.e., it can only disprove false properties.

To sum up, this paper makes the following contributions:

- The introduction of probability or controllability measures to estimate the influence of the gates of a circuit on the output of the circuit itself. These measures are computed with a complexity which is linear in the size (number of AIG gates) of the design.
- A procedure to compute both a subset and a superset of a given circuit, using the previous measures. The complexity of such a procedure is again linear with respect to the size of the circuit.
- The application of such an approximation routine to Bounded and Unbounded Model Checking of standard and industrial benchmarks.

With respect to Ravi et al. [13], who applied high-density to BDD-based reachability analysis, we focus on AIG-based representations, we compute both under- and over-estimates, and we concentrate on BMC and backward verification.

Experimental results prove the ability of our strategy to increase the density of circuit-based state set representations, and to produce a subsequent performance gain. Moreover, they show how under- and over-approximations may help to solve hard verification problems from BMC and backward unbounded verification.

The paper is organized as follows. Section 2 introduces some preliminary information on our notation, AIGs, and formal verification. Section 3 describes the theoretical framework to extract high-density representations from an AIG. Section 4 introduces practical aspects of the previously described operation. Section 5 shows how to apply the technique within circuit-based BMC and unbounded backward verification. Finally, Section 6 discusses the experiments we performed, and Section 7 concludes with a few summarizing remarks.

## 2. Background

Among the three main circuit-based representations, i.e., AIGs [10], BEDs [15], and RBCs [1], we adopt the first one because of its simplicity. AIGs can conceptually be seen as a graph structure containing operator nodes, variable nodes, and a mechanism for representing terminals. They consist of two-input AND operators only, and represent negations with markers on edges. Complex gates can always be expressed with this representation. Two level-minimization [10] is done before node creation. When an operator node with two child nodes must be created, the two level circuit formed by the node itself and its children is rewritten to a canonical two-level circuit, with as few nodes as possible. Given an AIG $f$, we indicate with $n_f$ its size in terms of number of gates.

In our notation, $B$ indicates the Boolean space. Symbols $\wedge$, $\vee$, $\neg$, $\equiv$ and $\Rightarrow$ are used for Boolean conjunction (AND), disjunction (OR), negation (NOT), exclusive-nor (XNOR) and implication, respectively. We indicate with $s$, $x$ and $y$ the sets of present state, input and next state variables, respectively. Given a set $\alpha$, we adopt the notation $|\alpha|$ to indicate its cardinality.

The sequential systems we address are usually modeled as Finite State Machines (FSMs). We indicate with $\delta(s,x)$ the set of next state functions as a function of present states and input variables. Each FSM is then described by an initial state set $\mathsf{S}$ and a Transition Relation which indicates the present-to-next state behavior of the design:

$$\mathsf{TR}(s,x,y) \quad = \quad \textstyle\bigwedge_i (y_i \equiv \delta_i(s,x)) \tag{1}$$

The positive and negative cofactors of $f(v)$ with respect to a variable $v_i$ are

$$f_{v_i} \quad = \quad f(v_1,\ldots,1,\ldots,v_n)$$

and

$$f_{\neg v_i} \quad = \quad f(v_1,\ldots,0,\ldots,v_n)$$

respectively.

Existential quantification of $f(v)$ with respect to a variable $v_i$ is defined as:

$$\exists_{v_i} f \quad = \quad f_{v_i} \vee f_{\neg v_i}$$

A function $f$ is positive unate in variable $v$ iff:

$$f_v \quad \Rightarrow \quad f_{\neg v}$$

It is negative unate in variable $v$ iff:

$$f_{\neg v} \quad \Rightarrow \quad f_v$$

An invariant property $\mathsf{P}$ can be checked by disproving the reachability of its complement, the target set of states $\mathsf{T}$ ($\mathsf{T} = \neg\mathsf{P}$), from the initial state set $\mathsf{S}$.

## 3. Probabilistic Circuit-Based Quantification

Let us suppose that a single-output Boolean circuit (i.e., an AIG) represents a function $f$. Moreover, let us suppose that we have a circuit size threshold $t$, determined by the memory and time resources available during the verification process. If the circuit size exceeds the threshold, we explore approaches to reduce the size of $f$ and thus increase its representativity, by sub-setting or super-setting its representation. This problem can be more formally posed in the following way.

**Definition 1.** *Let $f$ be a function represented by a single-output circuit with $n_f$ gates. Let $t$ be a threshold, such that $n_f > t$. We define a proper under-estimate of $f$, the function $f_s$ represented by a circuit with $n_{f_s}$ gates, such that $f_s \Rightarrow f$, $n_{f_s} < n_f$, and such that the number of minterms of $f_s$ is maximum, i.e., its on-set has the maximum cardinality.*

Over-estimates ($f^S$ of $f$) can be defined dually. The new circuit $f_s$ ($f^S$) can be obtained from $f$ by simplification, i.e., by erasing or modifying part of the original circuit. This means that we have to cut-out part of the AIG, trying to reduce (or increase, for $f^S$) the number of minterms as little as possible. As a result, we define the circuit density as follows.

**Definition 2.** *Given a circuit-based representation $f$, we define its density as:*

$$density\ (f) \quad = \quad \frac{|on - set\ (f)|}{n_f}$$

Our target is to maximize the density of the representation.

In [13], the authors proposed two main strategies to obtain dense subsets from a BDD. The first one, called heavy branch sub-setting, is based on the simple observation that a subset of a given BDD can be created by setting one cofactor of a node to the constant 0, and retaining the other. The procedure creates a subset of the given BDD by discarding (setting to the constant 0) the child at each node with the smaller number of minterms. The second strategy, the short-path sub-setting, is based on the rationale that, among all paths from the root of the BDD to the terminal constant 1, shorter paths represent more minterms. Thus, the driving idea is to keep all the nodes on the shortest paths and to erase the nodes in the longest paths.

Although the two techniques can be applied to our AIG structure, they do not deliver the desired results. For instance, it is easy to rule-out longer paths on an AIG, but on this structure there is no correlation between the length of a path and the number of minterms represented by the path. In fact, while BDDs are a canonical representation, AIGs are not, as several representations are possible for the same set of minterms.

To perform the same operation on an AIG, our algorithm proceeds in two steps. In the first one, it evaluates which parts of the circuit have to be cut-out in order to increase the circuit density. In the second one, it cuts them out to obtain a proper under- (or over-) approximation of the original representation. We analyze the two phases in the following two subsections.

## 3.1 Probabilistic High-Density Analysis on AIG

To simplify our AIG network representation, we resort to probability and testability measures. We first analyze these measures. Then, we apply them to decide which nodes to cut in order to under- or over-approximate the circuit.

### 3.1.1 Probability Measures

Given a node $n$, we denote by $P_1(n)$ the probability of $n$ to be equal to 1. Analogously, $P_0(n)$ indicates the probability of $n$ to be equal to 0. The probability formulas for AND and NOT gates are given by:

$$\begin{array}{rclcrcl} o & = & i_1 \wedge i_2 & \qquad & P_1(o) & = & P_1(i_1) \cdot P_1(i_2) \\ o & = & \neg i & \qquad & P_1(o) & = & 1 - P_1(i) \end{array}$$

Obviously, for a given node $n$, $P_0(n)$ can always be computed from $P_1(n)$ as

$$P_0(n) \quad = \quad 1 - P_1(n)$$

By assigning a probability to the AIG variables, probability values can be derived for the entire Boolean network proceeding from the primary inputs to the primary output. Unfortunately, these expressions are correct only if the Boolean network is a tree structure, as, because of signal re-convergence, gates are correlated and not independent. As we do not have specific information on the value assumed by the inputs, we arbitrarily set the probability of each variable equal to 0.5.

### 3.1.2 Controllability Measures

An alternative approach to the one outlined in Section 3.1.1 is allowed by *controllability* measures. These measures indicate the relative difficulty of setting a line to a certain value. For every node $n$ we can compute two cost functions $C_0(n)$ and $C_1(n)$, to reflect the relative difficulty of setting $n$ to 0 and 1, respectively. In this approach, knowing the values of $C_0$ and $C_1$ for each input of a gate, we can compute controllability measures for each AND and NOT gate following [7]:

$$\begin{array}{lllllll} o & = & i_1 \wedge i_2 & \quad C_0(o) & = & min\{C_0(i_1), C_0(i_2)\} + 1 \\ & & & \quad C_1(o) & = & C_1(i_1) + C_1(i_2) + 1 \\ o & = & \neg i & \quad C_0(o) & = & C_1(i) + 1 \\ & & & \quad C_1(o) & = & C_0(i) + 1 \end{array}$$

where the term “+1” is added to keep into account the logic depth of a gate. The process starts by setting the controllability values for variables equal to 0. As in the probabilistic analysis, we have less accurate controllability values in the presence of non-independent gate inputs, i.e., reconvergent fanout. In order to reduce this effect, it is possible to exploit the so called fanout-based controllability measure [2], introducing a correction term which depends on the fanout of the gate.

As a final remark, notice that, given its definition, the probability measure $P_\alpha(n)$ increases if the probability to have $n$ equal to $\alpha$ is higher. On the contrary, the controllability measure $C_\alpha(n)$ increases when the difficulty to set $n$ to $\alpha$ is higher, i.e., increases if the probability to have $n$ equal to $\alpha$ is lower. Moreover, the probability of a node to be equal to 0 or 1 is always included in the range [0, 1], whereas the controllability values increase from the inputs to the output. As a consequence, controllability has to be normalized with respect to the level of the node in the network, if we want to obtain a steady and level-independent measures. Finally, notice that a further alternative to probability measures would be using observability on the output.

### 3.1.3 Selecting the Cut-Points

In the sequel we present our strategy for selecting the cut-points by using the probability measures, although our reasoning can be applied to controllability (and observability) as well. Let us consider Figure 1. Given the output function $f$, for each internal node $n$ we compute:

- The probabilities of that node to be equal to 1 and 0, i.e., $P_1(n)$ and $P_0(n)$ respectively.
- The numbers of paths from $n$ to the output $f$ containing an even and an odd number of negations (i.e., edge inversions in the AIG structure). We denote the first set of

paths by $E$, and its cardinality by $|E|$, and the second set of paths by $O$, and its cardinality by $|O|$.

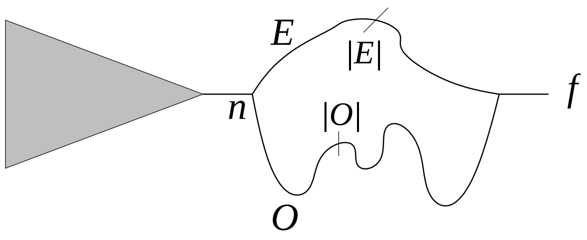

**Figure 1.** Selecting a cut-point using probability measures.

To obtain a proper subset (superset) of the function, we replace some of the nodes $n$ with the constants 0 and 1, essentially propagating a value equal to 0 (1) on the $E$ paths, and a value equal to 1 (0) on the $O$ paths. This procedure will be fully described and motivated in Section 3.2, but the intuition behind it is that each complementation toggles a subset in a superset and vice-versa[1.]. Hence, obtaining the right (under/over) estimate can be seen as a function of the constant value (0/1) used to replace $n$, and the number of inversions along the paths from $n$ to the output. In any case, this procedure produces an error in the circuit as the we propagate inconsistent 0/1 values different paths. To clarify this point, let us concentrate on sub-setting. If the node $n$ is set to 0 by the current variable assignment, we do not have any error on the $E$ paths (on which we propagate the right value 0), but we generate an error on the $O$ paths (on which we propagate the wrong value 1). The error produced on the $O$ paths is proportional to the probability of $n$ being equal to 0. As this error propagates on all $|O|$ paths towards the output, the global error on these paths can be evaluated as $|O| \cdot P_0(n)$. The same reasoning can be made when $n$ is equal to 1. In that case, we do not have any error on the $O$ paths (on which we propagate the right value), but we have an error on the $E$ path (on which we propagate the wrong value). This error is equal to $|E| \cdot P_1(n)$. As a consequence, the probability to have a wrong value on the output is equal to:

$$P_E(n) \quad = \quad |E| \cdot P_1(n) + |O| \cdot P_0(n) \tag{2}$$

Unfortunately, the value of $P_E(n)$ in the previous expression is influenced by the position of the node $n$ in the AIG. In fact, nodes closer to the output tend to have fewer paths toward the output (smaller $|E|$ and $|O|$ values). To have a probability distribution independent from the position of $n$ in the AIG, we normalize Equation (2) as follows:

$$P_E(n) \quad = \quad \frac{|E| \cdot P_1(n) + |O| \cdot P_0(n)}{D \cdot (|E| + |O|)} \tag{3}$$

where $D$ is the number of gates on the longest path from $n$ to the output, i.e., the maximum number of gates to traverse to reach the output from $n$. The denominator of the equation is larger for nodes closer to the primary inputs.

1. The reason for this is that for any subset $f_s$ and superset $f^S$ of a Boolean function $f$, such that $f_s \Rightarrow f \Rightarrow f^S$, it is true that $\neg f^S \Rightarrow \neg f \Rightarrow \neg f_s$.

Once we have the probability value for each node $n$, we perform our cuts on nodes $n$ with a small value of $P_E(n)$, in order to minimize the probability to have a wrong value on the output. For each cut:

- The nodes in the transitive fanin of the selected node $n$, which do not have any fanout outside the transitive fanin itself, may be erased.
- The injection of a constant (0 or 1) value on node $n$, and the subsequent Boolean constraint propagation (BCP) usually produces a simplification in the AIG and a reduction in terms of nodes.
- A certain number of new nodes must be generated to build the required under- or over-approximation. This is due to the propagation of different constant values (0 or 1) on different paths (see Section 3.2).

The result in terms of nodes is the sum of the three previous contributions. To obtain the desired reduction in terms of AIG nodes, we apply a technique derived from [4] and described in the following section.

### 3.2 Computing Under- or Over-Approximations of an AIG

To produce an under- (over-) approximation of an AIG, we may replace each sub-net we want to cut with:

- A constant value (0 or 1) expressing the under- or over-approximation.
- A simpler version of the sub-net we want to cut by re-synthesizing the sub-net.

In both cases, the main issue we have to solve is to understand how to obtain a strict under- (or over-) approximation of the network by cutting or modifying a part of it. In this paper we concentrate on the first strategy.

Note that replacing an AIG node with a constant 0 (1) does not necessarily produce an under- (over-) estimation of the output function. For example, in Figure 2(a) we have $g = x \wedge y$ and $f = \neg(x \wedge y)$. In this situation, setting $x = 0$ forces $g = 0$, under-approximating it, but $f = 1$, over-approximating it.

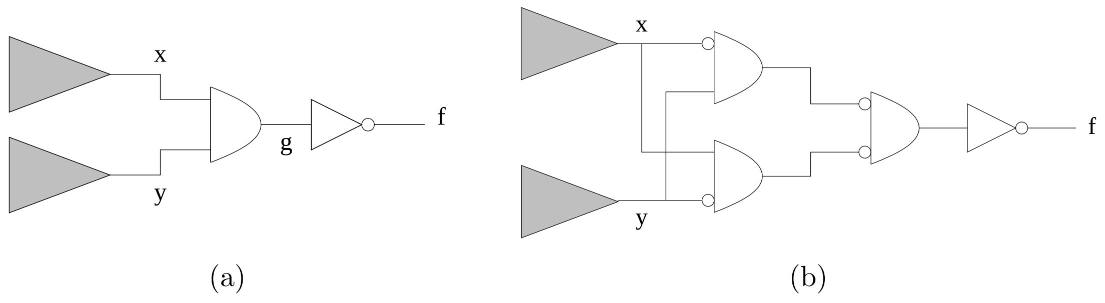

**Figure 2.** Obtaining under- (over-) approximation.

As a further example, consider Figure 2(b) where $f = \neg(x \equiv y)$. Setting $x = 0$ gives $f = y$, and setting $x = 1$ produces $f = \neg y$. These are neither an under- nor an over-estimate of the original $f = \neg(x \equiv y)$, which can always assume both the value of $y$ and $\neg y$ depending on the value of $x$.

To avoid the above problems, we remember that if $n$ is a positive (negative) unate node, $f$ can be replaced with $f_n$ ($f_{\neg n}$) to obtain an under-approximation of $f$ (the reverse is true to obtain an over-approximation). Moreover, a sufficient condition for a node $n$ to be positive (negative) unate in an AIG is that all paths from $n$ to the output $f$ contain an odd (even) number of inversions.

In general, every node $n$ will have paths with both polarities toward the output, and different paths can also share sub-paths. Let us suppose to make a cut on the gate named $g_1$, and that there is more than one path connecting gate $g_1$ to gate $g_2$ (i.e., $g_2$ is a *re-convergent gate* for $g_1$ and $g_1$ is a *re-convergent fanout* for the network). The only feasible strategy in this situation is to produce both an under- and an over-estimate for each node of the network included from $g_1$ and $g_2$. In order to produce opposite estimates (under versus over), we have to propagate different constant values on the $E$ and $O$ paths. Thus, we have to duplicate the entire circuit between gates $g_1$ and $g_2$, re-synthesizing it during BCP. Depending on how much we have to duplicate and how effective BCP is, the size of the AIG can actually increase or decrease. To avoid an increase, we estimate the number of nodes added during the procedure as a function of the node level difference between gates $g_1$ and $g_2$. As a result, we actually perform the cut only when the number of erased nodes, as computed in Section 3.1.3, is larger than the predicted number of added nodes.

As a final remark notice that contributions propagated along different paths are always combined maintaining the proper estimate. In fact, with AIGs, reconvergent paths are combined with two-input AND gates, and a conjunction of two under- (over-) approximations is always an under- (over-) approximation.

## 4. Implementation Details

In this section, following Section 3, we first present the implementation details on how to select the cutting points. Then, we show how to perform the cut.

### 4.1 Selecting the Cut-Points

Figure 3 shows the pseudo-code of the algorithm adopted to decide how to cut the AIG network.

Function ProbabilisticCut receives as parameters the AIG $f$, and a node size threshold which indicates the maximum size allowed for $f$. A depth first visit (DFSVisit, line 2) is used to compute the probability measure described in Section 3.1 for every node of the AIG. This procedure returns the probability measures array (prob$_{[]}$) and the array of the maximum distance of each node from the leaves (inputs) of the AIG (leafDist$_{[]}$). Then, the routine PFSVisit (line 3) performs a prioritized visit, using the value leafDist$_{[]}$ as priority. The purpose of the prioritized visit is to find, for every node, its maximum distance from the root (output) of the AIG (rootDist$_{[]}$) and, at the same time, the number of disjoint paths toward the AIG output (numPaths$_{[]}$). These values can be computed with a single pass on the AIG using the array leafDist$_{[]}$ as priority values for the prioritized visit. This is

```
1   ProbabilisticCut ($f$, threshold)
2       (leafDist$_{[]}$, prob$_{[]}$) = DFSVisit ($f$)
3       (rootDist$_{[]}$, numPaths$_{[]}$) = PFSVisit ($f$, leafDist$_{[]}$)
4       errorMeasure$_{[]}$ = FindError (probability$_{[]}$, rootDist$_{[]}$, numPaths$_{[]}$)
5       cutNodes$_{[]}$ = FindCutNodes (errorMeasure$_{[]}$, threshold)
6       return (cutNodes$_{[]}$)
```

**Figure 3.** Pseudo-code for the probabilistic heuristics.

because, before visiting a particular node, we are sure that all the AIG nodes in its fanout have been already visited.

After that, all these measures are combined together, by applying Equation 3, in order to obtain an estimate of the error for each node (line 4). Eventually, the algorithm selects which nodes $n$ of the AIG have to be cut (line 5). This is done by choosing, among the nodes for which $P_E(n)$ is minimum, the ones for which the net balance in terms of effectively deleted nodes is positive.

Notice that function ProbabilisticCut has a linear time-cost, as it performs two complete visits over the AIG structure. In particular, the number of paths of a node to the AIG output can be exponential in the number of gates in the AIG itself, yet it can be computed in linear time following an algorithm similar to the minterm count for BDDs. The principle is simple. The gate that generates the output has only 1 path towards the output itself. Every other gate of the network has a number of paths that is equal to the sum of all paths throughout the gates in its fanout. As a consequence, a single linear visit of the network, starting from the output and moving towards the inputs, is sufficient to compute the number of paths of all gates towards the output.

## 4.2 Computing Under- or Over-Approximations of an AIG

Once we have decided how to prune the AIG network, we still have to build the estimate of the output node. Figure 4 shows the pseudo-code to perform this step.

Function BuildSubset receives two parameters: $f$ is the AIG network to approximate; the cutNodes array, generated (as shown in Section 4.1) by routine ProbabilisticCut, indicates where to make the cuts. The function performs a single recursive post-order visit of $f$. This is guaranteed by procedure SetVisited (line 4), which marks all visited nodes, and routine IsVisited (line 2) which checks this condition, avoiding further passages for the same node. For each recursion step, procedure BuildSubset computes both an under- and an over-approximation of the Boolean function represented by the current *regular*, i.e., non-inverted, node. Such approximations are denoted, in Figure 4, with $f_s$ and $f^S$, respectively. Nevertheless, function BuildSubset always returns the right under-estimation of the node $f$. If $f$ is non-negated, then this value is really given by $f_s$; otherwise, the subset is generated as $\neg f^S$. The reason for this is that for any Boolean function $f$, given a subset and a superset of it such that $f_s \Rightarrow f \Rightarrow f^S$, it is also true that $\neg f^S \Rightarrow \neg f \Rightarrow \neg f_s$.

The critical steps of the algorithm are:

```
1   BuildSubset ($f$, cutNodes$_{[]}$)
2       if (IsVisited ($f$)) then
3           return (IsInverted ($f$) ? $\neg f^S$ : $f_s$)
4       SetVisited ($f$)
5       if (IsACutNode ($f$, cutNodes$_{[]}$)) then
6           $f_s$ = ZERO
7           $f^S$ = ONE
8           return ($f_s$)
9       if (IsVariable ($f$)) then
10          $f_s = f^S$ = Regular ($f$)
11          return ($f$)
12      // Compute regular Subset
13      $left_s$ = BuildSubset ($left$, cutNodes$_{[]}$)
14      $rigth_s$ = BuildSubset ($right$, cutNodes$_{[]}$)
15      $f_s = left_s \wedge right_s$
16      // Compute regular Superset
17      $left^S$ = IsInverted ($left$) ? $\neg left_s$ : $left^S$
18      $right^S$ = IsInverted ($right$) ? $\neg right_s$ : $right^S$
19      $f^S = left^S \wedge right^S$
20      // Return the true Under-Approximation
21      return (IsInverted ($f$) ? $\neg f^S$ : $f_s$)
```

**Figure 4.** Pseudo-code for the under-approximation construction.

- If the current node $f$ has already been visited (line 2), its under-estimate is directly returned (line 3).
- If the current node $f$ must be removed, i.e., it has to be cut (line 5), we follow the procedure detailed in Section 3.2. The subset and superset of $f$ are given by the constants 0 (line 6) and 1 (line 7), respectively. The subset is returned on line 8.
- If the current node $f$ must be kept in the final AIG, and it is a variable, then both the subset and the superset coincide with the variable itself (lines $9 - 11$).
- If the node $f$ must be kept in the result, and it does not represent a variable, its subset (superset) is computed as a conjunction of the under- (over-) estimates of its children (the children of node $f$ are indicated as $left$ and $right$). This implies a recursion on both children of the node (lines 13 and 14) and the operations on lines $15 - 19$ to evaluate the estimates of the current node given the estimates on the children.

Notice that the algorithm shown in Figure 4 replaces the selected nodes of the original AIG with 0 and 1, and then propagates one of these two values on the $E$ or $O$ paths. During this propagation, as analyzed in Section 3.2, parts of the AIG has to be duplicated, and re-synthesize during BCP.

Function BuildSuperset, which returns an over-estimate of $f$, can be obtained by BuildSubset in the following way. We have already seen that function BuildSubset computes both an under- and an over-estimate for each node $f$. To obtain function BuildSuperset it is sufficient to exchange $f_s$ and $f^S$ in lines 3 and 21, and to return $f^S$ instead of $f_s$ in lines 8. These steps guaranteed the correctness of the result, given the correctness of the under- and over-approximations computed by the original function BuildSubset.

## 5. Applications

This section shows how it is possible to use under- and over-estimates of an AIG, computed in Section 4 by functions BuildSubset, and BuildSuperset, within formal verification. We adapt BMC and backward breadth-first reachability in order to exploit the circuit-based approximations obtained through the probabilistic analysis.

### 5.1 Bounded Model Checking

In BMC, a sequence of transitions from state $s_0$ to state $s_l$ (through states $s_1, s_2, \ldots, s_{l-1}$) is expressed by unrolling $l$ times the transition relation $\mathsf{TR}$. The process usually starts with $l$ equal to 1, and it proceeds with increasing the value of $l$ until a counter-example is found or all available resources are exhausted. At each iteration, the problem is expressed in CNF form and solved with a state-of-the-art SAT-solver to prove (or disprove) the reachability between the source $\mathsf{S}$ and the target $\mathsf{T}$.

As our approximations will not guarantee to find the minimum-length path, we will express, at each step, the so called *bound l* problem:

$$(s_0 = \mathsf{S}) \wedge \left[\ \mathsf{TR}(s_0, s_1) \wedge \ldots \wedge \mathsf{TR}(s_{l-1}, s_l)\ \right] \wedge \left[\ (s_1 = \mathsf{T}) \vee \ldots \vee (s_l = \mathsf{T})\ \right] \tag{4}$$

which finds paths up to length $l$.

We modify standard BMC to use both under- and over-approximations of the combinational unrolling. Note that given a formula $f$, e.g., the combinational unrolling of Equation 4, we know that:

$$\begin{array}{ccccc} \text{UNSAT}\ (f^S) & \Rightarrow & \text{UNSAT}\ (f) & \Rightarrow & \text{UNSAT}\ (f_s) \\ \text{SAT}\ (f_s) & \Rightarrow & \text{SAT}\ (f) & \Rightarrow & \text{SAT}\ (f^S) \end{array}$$

This means that we can conservatively deduce unsatisfiability from the over-estimates of $f$, and satisfiability from its under-estimates. The idea is to reduce the size of $f$, i.e., the BMC problem, whenever possible, and to resort to the exact formulation of $f$ only when necessary. To do this, we have two main approaches:

1. Make an estimate of $\mathsf{TR}$, and use this estimate in all time-frames of Equation 4.

2. Obtain the entire unrolling of Equation 4 with the original $\mathsf{TR}$, and produce a single estimate of it.

Figure 5 shows the pseudo-code for this last possibility.

Function BMCApprox executes three different steps of BMC using, respectively, an exact (lines $4-10$), an over-approximate (lines $11-17$), and an under-approximate (lines

```
BMCApprox ($\delta$, S, T, threshold, $\alpha$)
    TR = buildTR ($\delta$)
    $l = 1$
    // Exact Section
    do {
        {size, result} = SatCheck (TR, S, T, $l$, threshold, exact)
        if (result = sat)
            return (Trace (TR, S, T))
        $l = l + 1$
    } while (size < threshold)
    // Over-Approximate Section
    while (True) {
        {size, result} = SatCheck (TR, S, T, $l$, threshold, super)
        if (result = sat)
            break
        $l = l + 1$
    }
    // Under-Approximate Section
    while (True) {
        {size, result} = SatCheck (TR, S, T, $l$, threshold, sub)
        if (result = sat)
            return (Trace (TR, S, T))
        $l = l + 1$
        threshold = threshold + $|\mathsf{TR}| \cdot \alpha$
    }
```

**Figure 5.** BMC with sub- and super-setting.

```
SatCheck (TR, S, T, $l$, threshold, dir)
    $f = (s_0 = \mathsf{S}) \wedge \left[\prod_{i=1}^{l} \mathsf{TR}\ (s_{i-1}, s_i)\right] \wedge \left[\sum_{i=1}^{l} (s_i = \mathsf{T})\right]$
    $g$ = BuildApprox (f, threshold, dir)
    result = SAT ($g$)
    return ($|f|$, result)
```

**Figure 6.** SAT on a BMC approximate instance.

$18 - 25$) combinational unrolling. To generate such an unrolling, function BMCApprox calls procedure SatCheck (Figure 6).

SatCheck first evaluates the exact unrolling $f$ following Equation 4 (line 2). Then, it calls procedure BuildApprox (line 3). This procedure returns an under-approximation of $f$ if the parameter dir is equal to the constant sub, an over-approximation of $f$ if dir is equal to the constant super, and the original AIG $f$ otherwise. When BuildApprox has to compute an estimate of $f$, it executes function ProbabilisticCut, Section 4.1, followed by BuildSubset or BuildSuperset, Section 4.2, depending on the value of the last parameter. As previously described, function ProbabilisticCut selects the nodes to get rid of, whereas functions BuildSubset and BuildSuperset produce the under- and the over-estimate, respectively. Finally, SatCheck runs the SAT solver on the generated instance (line 4).

BMCApprox contains three main loops representing the three different BMC phases. The first phase (lines $4-10$) ends when a satisfying assignment is found (and then function Trace, line 8, is called to compute a counter-example), or the combinational unrolling has a size larger than the threshold (line 10). In the latter case, we proceed with an over-estimation of the unrolling (lines $11 - 17$). This is done because, over-estimated unrolling are smaller than exact ones and their evaluation is supposed to be faster. When this second phase ends with a sat result (line 14), we break the cycle and move to the under-approximated section in order to avoid false negatives (lines $18-25$). In this last phase, at each iteration we increase the threshold used to estimate the unrolling. We do that for two reasons. First, as the combinational unrolling becomes longer, we need a larger threshold to avoid too rough estimates. Second, as we increase the threshold of a value equal to $|\mathsf{TR}| \cdot \alpha$, with $\alpha > 1$, sooner or later the threshold becomes larger than the size of the unrolling. When this happens function BuildApprox returns the exact result, instead of an under-estimation, and the procedure BMCApprox starts to evaluate exact BMC unrolling. As a result, the procedure is sound but not complete, i.e., it finds a counter-example if one exists, even though it is not guaranteed to find the minimum-length counter-example. This is proved by the following theorem.

**Theorem 1.** *Function* BMCApprox *of Figure 5 is guaranteed to find a counter-example if one exists.*

**Proof** *The key idea to obtain the final proof is to analyze the three main loops of the procedure (line $4 - 10$, $11 - 17$, and $18 - 25$), and to demonstrate that they will never end computing a wrong result.*

*First, function* BMCApprox *performs exact BMC checks (lines 6) until a trace is returned (line 8), or the threshold is reached (line 10). In the first case, the procedure ends with a correct result, an exact minimum-length counter-example, since the BMC check of line 6 is performed on an exact unrolling. Otherwise, sooner or later the threshold will be reached, as the size of the combinational unrolling increases at each iteration. In this second case, the procedure moves to the over-approximate section of lines $11 - 17$.*

*In this second section a bounded check is performed on an over-approximated unrolling. For that reason any unsatisfiable check is truly unsatisfiable, whereas any satisfiable check can be a false negative. If a satisfiable check is never encountered, the procedure loops* *forever just like a standard BMC procedure. If a satisfiable check is eventually encountered, the function proceeds to the next section to validate this check.*

*The under-approximated section, lines* $18-25$*, consists of a loop in which the threshold, used to approximate the combinational unrolling, is increased at each iteration by an amount larger than* $|\mathsf{TR}|$*. As a consequence, sooner or later, the threshold becomes larger than the size of the combinational unrolling. Hence, we stop to approximate the unrolling, and the BMC check becomes exact. If the procedure finds a SAT result before this condition happens, the result is conservative, as the counter-example has been found on an under-estimate of the problem. Otherwise, if the procedure finds a SAT result after this condition, the BMC problem analyzed is the exact problem at that length. This guarantees the correctness of the procedure.* □

## 5.2 Backward Unbounded Verification

In the unbounded backward verification strategy presented in [8] all state sets are represented and manipulated by using AIGs instead of BDDs, as usually done within standard symbolic breadth-first verification. Operations on AIGs, e.g., checking for equivalence, are performed using a SAT engine. High-density under-estimations are applied to this method as shown by the pseudo-code in Figure 7.

Given a target set of states $\mathsf{T}$, we perform backward reachability from it and we terminate as soon as no newly reached states are found (i.e., a fixed-point is reached), or the initial state set is intersected. In the latter case, we have found a complete set of paths connecting the two sets, and then a counter-example can be extracted in the forward direction, moving from $\mathsf{S}$ to $\mathsf{T}$ within this set of paths.

At each step of the algorithm, we evaluate a pre-image of $\delta$ on the set of current states ($\mathsf{From}$), line 4, yielding the ones from which they can be reached in one step ($\mathsf{To}$). The $\mathsf{Reached}$ set accumulates reached states (line 16), and it is progressively used, by function PreImg (lines 4 and 12), as a don't-care set when computing newly reachable ones. Once the $\mathsf{To}$ set is obtained, we first check its intersection with the initial state set $\mathsf{S}$ (line 6), possibly returning a counter-example (line 7) if such an intersection is non-empty. Then we evaluate the fixed-point, by running a satisfiability check on the newly reached states (line 10). When, eventually, the $\mathsf{New}$ set is empty, we have to recover from previously lost (cut) states. To do that, we compute the pre-image of the overall reachable state set $\mathsf{Reached}$ (line 12). The $\mathsf{From}$ set for the next iteration is selected by function BestOf, line 18, as the smallest set (in size) between the $\mathsf{New}$ and $\mathsf{Reached}$ sets. Furthermore, it may be under-approximated before the next iteration (lines 19 and 20) if its size exceeds the threshold.

Function BackwardVerification calls routine PreImg in lines 4 and 12. Our pre-image computation adopts quantification by substitution [15] (also called in-lining in [1]):

$$\exists_x g(x) \wedge (x \equiv f) \quad = \quad g_{x=f}$$

The applicability of this transformation in our context relies on the fact that the formulas occurring in backward reachability always have a structure that matches the previous rule. In fact, the transition relation $\mathsf{TR}$ is a conjunction of next state functions defined in terms of current state variables, as shown by Equation 1.

```
BackwardVerification ($\delta$, S, T, threshold)
    Reached = From = T
    while (True) {
        To = PreImg ($\delta$, From, Reached)
        // Evaluate intersection for failures
        if (SAT (To $\wedge$ S))
            return (Trace (TR, S, T))
        // Evaluate fixed-point
        New = To $-$ Reached
        if ($\neg$ SAT (New)) {
            // Recover lost states
            To = PreImg ($\delta$, Reached, Reached)
            New = To $-$ Reached
            if ($\neg$ SAT (New))
                return ($Pass$)
        }
        Reached = Reached $\vee$ New
        From = BestOf (New, Reached)
        if ($|$From$|$) $>$ threshold)
            From = BuildApprox (From, threshold, SUB)
    }
```

**Figure 7.** Backward verification with sub-setting.

```
PreImg ($\delta$, Curr, DC)
    // Function Composition
    Next = Compose (Curr, $\delta$)
    // Weak quantifier elimination
    forall $x_i \in (x \wedge$ Support (Next))
        if ($\neg$ SAT ($\neg$ (Next$_x \equiv$ Next$_{\neg x}$) $\wedge \neg$ DC))
            Next = BestOf (Next$_x$, Next$_{\neg x}$)
    // Strong quantifier elimination
    forall $x_i \in (x \wedge$ Support (Next))
        Next = CircuitExist (Next, $x$)
    return (Next)
```

**Figure 8.** Pre-image computation.

The pseudo-code for function PreImg is shown in Figure 8.

The pre-image computation is achieved in two passes. First of all, we compose the Curr state set with the next state functions $\delta$ (line 3). Then, we perform two iterations targeting quantification over primary input variables $x_i \in x$. During the first iteration (lines $4 - 7$), we try a *weak* existential quantification, which consists of accepting quantification if the two cofactors are equivalent (under don't care conditions). During the second iteration (lines $8 - 10$), we quantify out all the remaining variables, through the CircuitExist procedure, that includes some optimization steps as described in [8]. Notice that the double iteration of pre-image computation is essentially a quantification order heuristic. The issue of quantification order was already raised in [15] and [1], but the authors mainly adopted a random order. The idea here is to initially quantify variables which produce the smallest increase in circuit size. The quantification order is optimized by learning and adjusting the order of previous quantification phases. For instance, unate variables, which are guaranteed not to increase the circuit size, may be quantified out up-front. Furthermore, to fight the circuit size explosion, we aggressively apply node merging driven by equivalence checks, and SAT-based synthesis optimization techniques (such as re-synthesis, rewriting, etc.), following [8, 12].

To conclude our analysis of the BackwardVerification procedure, we prove the following theorem.

**Theorem 2.** *Procedure* BackwardVerification *is both sound and complete, i.e., it does eventually visit all reachable states and it does not produce false results.*

**Proof** *The proof can be reached in two steps.*

*Procedure* BackwardVerification *computes pre-images of the target set of states until a fixed-point is reached, or the initial set of states is intersected. False negatives can be obtained only when unreachable states are considered to be reachable. As the* PreImg *function computes only exact pre-images, and function* BuildApprox *is called to generate only under-approximations of the reachable state set, false negatives cannot be produced. ¡ The procedure is also complete, because it eventually visits all reachable states. In fact, when no newly reached states are found (line* 10*), it recover from previously lost states by computing pre-images of the entire reachable state set (line* 12*).* □

## 6. Experimental Results

We present results obtained with our BMC and backward verification routines described in Sections 5.1 and 5.2, respectively.

We provide data on ISCAS benchmarks and some industrial circuits from STMicroelectronics. On ISCAS designs, our properties are automatically generated invariants. We generated target state sets T with increasing Hamming distance from the initial state set S. The invariant property requires the T states to be unreachable. On industrial benchmarks, properties come with the circuits.

Our experiments ran on a Pentium IV 1700-MHz Workstation with 1 GB main memory, running Debian GNU/Linux 4.0. Our tool uses the Minisat [9] tool (version p_v1.14) as underlying SAT solver. We used a time limit of 1 hour.

Tables 1 and 2 provide our results on property verification. Table 1 compares standard BMC [3] with the strategy presented in Section 5.1. Table 2 contrasts backward unbounded verification as presented in [8] with the method introduced in Section 5.2.

In both tables, models are sorted by the original (before the cone of influence extraction) number of state variables (column #FF). Column #PIs indicates the number of primary inputs of the circuit. Properties are named as $\mathsf{P}_i$ and numbered sequentially within the same circuit to indicate increasing verification difficulty. Within this column, the number of memory elements in the cone of influence of the property (COI) is reported between parentheses. The CPU time and memory used are reported for both the exact and the approximate technique. For all the approaches, all contributions to the CPU time are included, i.e., symbolic AIG manipulations and SAT run times are both taken into account.

Table 1 compares standard exact and our approximate BMC. Given the debugging nature of BMC, all presented properties are false. The number of memory elements, after the cone of influence reduction, varies from 37 to 922. On average our algorithm spends about 30% of the time in the first (exact) section, 10% of the time in the second (over-approximation) part, and the remaining 60% in the third (under-approximation) bounded model checking loop (see Figure 5). Memory results are not impressive, as we obtained memory reduction varying from 10 to 20%. Anyway, the numbers reported do not take into account the memory used directly by the SAT solver, which is run as an external process. Time measures, however, are more interesting. In all the reported experiments, our approximate BMC approach performed better than the exact (original) BMC formulation. Furthermore, our method was able to solve even the largest instance (s38584), whereas the original technique ran out of time. Overall, we obtained an average speedup of about 2.

Table 2 provides results on backward unbounded verification, considering both true and false properties.

**Table 1.** Debugging false properties with approximate BMC on ISCAS benchmarks. Comparison between the original and the proposed algorithms. A dash (–) means data not available due to time overflow ($ovf$).

| Model | #FF | #PIs | Property (COI) | Exact Time [s] | Exact Mem [MB] | Subset Time [s] | Subset Mem [MB] | Speedup |
|---|---|---|---|---|---|---|---|---|
| s1269 | 37 | 18 | $P_1$ (37) | 14 | 15.6 | 10 | 14.1 | 1.4× |
| s1512 | 57 | 29 | $P_1$ (57) | 79 | 13.2 | 41 | 12.5 | 1.9× |
| s3271 | 116 | 26 | $P_1$ (95) | 118 | 25.4 | 65 | 19.6 | 1.8× |
| s5378-164 | 164 | 35 | $P_1$ (164) | 221 | 17.7 | 204 | 17.0 | 1.1× |
| s3384 | 183 | 43 | $P_1$ (125) | 338 | 26.6 | 245 | 24.7 | 1.4× |
| s9234 | 228 | 19 | $P_1$ (187) | 869 | 31.7 | 476 | 27.3 | 1.8× |
| | | | $P_2$ (228) | 1374 | 34.2 | 938 | 31.2 | 1.5× |
| s15850.1 | 534 | 77 | $P_1$ (534) | 1945 | 42.1 | 1037 | 38.9 | 1.8× |
| s13207.1 | 638 | 62 | $P_1$ (621) | 2693 | 59.8 | 1745 | 53.4 | 1.5× |
| s38584 | 1452 | 12 | $P_1$ (922) | $ovf$ | – | 2238 | 71.8 | $>$ 1.6× |

**Table 2.** Verifying false and true properties, with circuit-based backward unbounded verification, on some ISCAS and industrial circuits. Comparison between the original and the proposed algorithms. A dash (−) means data not available due to time overflow ($ovf$).

| Model | #FF | #PIs | Property (COI) | Property Pass/Fail | Exact Time [s] | Exact Mem [MB] | Subset Time [s] | Subset Mem [MB] | Speedup |
|---|---|---|---|---|---|---|---|---|---|
| s1423 | 74 | 17 | $P_1$ (68) | Pass | 126 | 17.6 | 55 | 16.8 | 2.3× |
| | | | $P_2$ (68) | Fail | 248 | 17.9 | 16 | 16.4 | 15.5× |
| s4863 | 104 | 49 | $P_1$ (104) | Pass | 1428 | 23.8 | 1025 | 21.1 | 1.4× |
| s3330 | 132 | 40 | $P_1$ (67) | Fail | 729 | 17.9 | 342 | 17.6 | 2.1× |
| | | | $P_2$ (131) | Pass | $ovf$ | − | 1752 | 19.8 | $>$ 2.0× |
| s5378-179 | 179 | 35 | $P_1$ (160) | Fail | 31 | 17.7 | 11 | 17.0 | 2.8× |
| s3384 | 183 | 43 | $P_1$ (125) | Fail | 1206 | 22.4 | 543 | 19.8 | 2.2× |
| | | | $P_2$ (183) | Pass | $ovf$ | − | 1356 | 21.2 | $>$ 2.6× |
| industrial$_1$ | 377 | 48 | $P_1$ (377) | Pass | $ovf$ | − | 1165 | 55.4 | $>$ 3.1× |
| industrial$_2$ | 673 | 74 | $P_1$ (545) | Pass | $ovf$ | − | 2124 | 78.5 | $>$ 1.6× |

Properties are denoted by Pass when proved to be correct, and with Fail, otherwise. Passing properties are usually harder to verify as they need a complete visit of the backward reachable state space. The cone of influence of the properties varies from 67 to 545. The advantage of sub-setting lies in keeping the overall size and memory occupation low at all stages of the traversal. The threshold is chosen heuristically, trying to achieve fast image computation and small intermediate results, while not excessively increasing the number of iterations. The proposed method was able to complete verification instances on which the original technique run out of time. On the other cases, we observe speedups up to more than 15, and memory savings in the order of $10 - 20\%$ .

Finally, Table 3 reports data on our estimation procedure. We used two different mechanisms to count the number of minterms in an AIG. The first one is based on a SAT-count, by adopting the blocking-clause strategy presented in [11]. In this case, we try to find a satisfying assignment for the formula. If such an assignment exists, we count the number of minterms represented by it, and we add a blocking clause, i.e., the negation of the assignment, to the formula itself in order to avoid running into the same assignment again. The process goes on until no more satisfying assignment exists. The second one consists of converting the AIG to a BDD and counting the number of minterms in this structure. This can be easily done by using standard routines on BDD (see [14] for further details). While our implementation of the first technique relies on a quite inefficient blocking clause generation routine, the second approach shows an evident limit as BDDs can be built only for small AIG instances. This is the motivation to present only some small state sets experiments in Table 3. For each set we present the initial size and minterm count (columns Exact), and the ones obtained after the estimate (columns Subset). We concentrate on under-estimations, since such state sets have been obtained while running the experiments in Table 2, where only sub-setting was used. Column $D_s/D_E$ shows the ratio between the density of the under-estimated ($D_s$) and the original set ($D_E$). Results are encouraging as

**Table 3.** Results from the under-estimation procedure presented in Sections 3 and 4.

| Model | Property | Exact |  | Subset |  | $D_s/D_E$ |
|---|---|---|---|---|---|---|
|  |  | \|AIG\| [Nodes] | #Minterms | \|AIG\| [Nodes] | #Minterms |  |
| s1423 | $P_1$ | 224 | 3.40e+21 | 32 | 3.39e+21 | 14.2 |
|  |  | 366 | 7.05e+21 | 86 | 6.71e+21 | 4.1 |
|  | $P_2$ | 276 | 4.18e+21 | 114 | 4.16e+21 | 2.4 |
| s3330 | $P_1$ | 145 | 3.01e+38 | 73 | 2.48e+38 | 1.6 |
|  |  | 1794 | 4.15e+39 | 1065 | 3.12e+39 | 1.3 |
|  | $P_2$ | 4541 | 2.05e+39 | 1019 | 1.80e+39 | 3.9 |
| s5378-179 | $P_1$ | 187 | 3.59e+53 | 50 | 1.20e+53 | 1.2 |
| s3384 | $P_2$ | 468 | 3.82e+54 | 230 | 3.34e+54 | 1.8 |
|  |  | 2283 | 7.29e+55 | 496 | 6.02e+55 | 3.8 |

there are cases in which we are able to reduce the AIG size by about one order of magnitude maintaining almost the same number of minterms (see for example the numbers for circuit s1423 and s5378-179). The increase in terms of density ranges from a factor of a few unity to more a than 10.

## 7. Conclusions

Circuit-based techniques, and more specifically And–Inverter Graph structures, have been adopted in numerous applications in formal verification. Nevertheless, one of the limiting factors has been the difficulty to control their sizes during key operations. For example, existentially and universally quantifying a variable implies doubling the circuit size in the worst case.

In this paper, we presented a probabilistic approach to keep under control the size of circuit-based representations while manipulating them. The idea was to build an under- or an over-approximation of the exact result whenever necessary, i.e., whenever AIGs become too complex. The goal of this approximation was to build dense circuit representations, i.e., compact in size but expressive for the given verification problem.

We applied the technique to circuit-based Bounded and backward Unbounded Model Checking. We presented experimental results showing our ability to produce denser estimates, and to use these estimates within the two approaches on large verification instances. We obtained estimates of more than one order denser in several cases, speedups of more than one order of magnitude in some cases, and a slight reduction in terms of memory usage. Among the possible future developments, we envisage the application/integration of the method within alternative verification strategies, such as abstraction and refinement and interpolant-based verification.

## Acknowledgments

The authors would like to thank the VTT group of the STMicroelectronics, Agrate, Italy, for their industrial benchmarks.

## References

[1] P. A. Abdulla, P. Bjesse, and N. Een. Symbolic Reachability Analysis based on SAT-Solvers. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, **1785**, pages 411–425, Berlin, Germany, April 2000. Springer-Verlag.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT procedures instead of BDDs. In *Proc. 36th Design Automat. Conf.*, pages 317–320, New Orleans, Louisiana, June 1999. IEEE Computer Society.

[4] P. Bjesse and A. Boralv. DAG-Aware Circuit Compression For Formal Verification. In *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, November 2004. IEEE Computer Society.

[5] R. E. Bryant. Graph–Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, **C–35**(8):677–691, August 1986.

[6] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Trans. on Computer-Aided Design*, **13**(4):401–424, April 1994.

[7] M. L. Bushnell and V. D. Agrawal. *Essential of Electronic Testing – for Digital memory and mixed-signal VLSI circuits*. Kluwer Academic Publisher, Boston, USA, 2000.

[8] G. Cabodi, S. Nocco, and S. Quer. Circuit Based Quantification: Back to State Set Manipulation within Unbounded Model Checking. In *Proc. Design Automation & Test in Europe Conf.*, Munich, Germany, March 2005. IEEE Computer Society.

[9] N. Eén and N. Sörensson. Minisat SAT Solver, http://minisat.se/, 2003.

[10] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *Proc. Design Automat. Conf.*, Las Vegas, Nevada, June 2001. IEEE Computer Society.

[11] K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. Computer Aided Verification*, **2404** of *LNCS*, pages 250–264, Copenhagen, Denmark, 2002. Springer.

[12] A. Mishchenko. ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/~alanmi/abc/, 2005.

[13] K. Ravi and F. Somenzi. High–Density Reachability Analysis. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 154–158, San Jose, California, November 1995.

[14] F. Somenzi. CUDD: CU Decision Diagram Package – Release 2.4.1, `http://vlsi.colorado.edu/~fabio/CUDD/`, 2005.

[15] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In E. Allen Emerson and A. Prasad Sistla, editors, *Proc. Computer Aided Verification*, **2102** of *LNCS*, pages 124–138, Chicago, Illinois, July 2000. Springer-Verlag.