Simulation and Satisfiability in Logic Synthesis[1]

Jin S. Zhang[1] **Subarna Sinha**[2] **Alan Mishchenko**[3] **Robert K. Brayton**[3] **Malgorzata Chrzanowska-Jeske**[1]

[1] Department of ECE, Portland State University, Portland, OR
[2] Synopsys Inc., Mountain View, CA
[3] Department of EECS, UC Berkeley, Berkeley, CA

Abstract

Simulation and Boolean satisfiability checking (SAT) are common techniques used in logic verification. In this paper, we show how simulation and satisfiability (S&S) can be tightly integrated to efficiently solve two computationally intensive problems in logic synthesis: computing SPFDs for nodes in a Boolean network, and computing sets of candidate nodes for resubstitution. In the first application, simulation is used to quickly enumerate some solutions while SAT is used to complete the enumeration. In the second application, simulation is used to filter out infeasible solutions while SAT is used to prove the desired properties in the remaining candidates. Experimental results confirm that the combination of simulation and SAT offers a computation engine that outperforms BDDs, traditionally used in similar applications. We also discuss how S&S can be applied to other synthesis problems.

1. Introduction

Simulation has always played an important role in testing and verification of digital designs. Its main advantage is that it can catch many bugs very quickly; however, it is an incomplete technique. On the other hand, *Boolean satisfiability* (SAT) is able to prove, by an efficiently performed exploration of the search space, that a certain property always holds. Because of its exhaustive nature, SAT can be slow in solving large problems or many instances of small problems. Recently, the remarkable progress in SAT [15][24][6] has made it possible to extend its range of applications.

The two methods, simulation and SAT (abbreviated S&S), make a powerful combination, taking advantage of the strengths of each method. Simulation quickly disproves properties, saving the runtime that SAT would need to prove or disprove the properties using exhaustive search. However, simulation quickly saturates, at which point SAT can step in to solve the remaining instances. When the computational resources are intelligently divided among the methods, S&S becomes a formidable competitor to alternate methods such as BDDs [5][1]. While S&S is widely used in formal verification and ATPG, it is much less common in logic synthesis.

Recent experience with re-implementing traditional logic synthesis methods [25] showed that a surprising number of computationally intensive problems can be re-formulated and efficiently solved using S&S. Moreover, S&S almost always led to more robust implementations compared to using BDDs.

BDDs and SAT appear to have complementary strengths. For a detailed exposition, please refer to [13]. Overall, SAT appears to be more robust because most problems efficiently solved by BDDs can also be efficiently solved by SAT. For example, *BDD sweeping* [13], which detects functionally equivalent nodes in a combinational circuit of limited logic depth, can be replaced by *SAT sweeping* [14] without losing performance. In contrast, many properties proved by SAT cannot be proved by BDDs because of the BDD sizes encountered. A recent study in logic synthesis [21] shows that a SAT-based don't-care computation scales better than a BDD-based one.

One reason why SAT outperforms BDDs is the following. On many problems, although both BDDs and SAT require, in the worst case, exponential time in the problem size, the BDD-based approach starts by constructing the canonical representation for the given functions. SAT starts by searching the solution space immediately, relying on the available circuit to represent the functions. Thus SAT avoids the overhead of constructing the canonical representation, which can overwhelm BDDs in many practical instances; the BDD size is large or dynamic variable reordering is slow. Similar reasons for the development of SAT-only combinational equivalence checkers are stated in [8].

In our applications, the SAT solver is called many times to enumerate through the whole space for satisfying assignments (ALL-SAT problem), which increases its complexity. However, the input space is bounded, and simulation detects a significant number of satisfying assignments, leaving SAT to work only on the hardest ones. Therefore, solution enumeration works well in practice, resulting in affordable runtimes for many benchmarks. These results can be further improved by incorporating the recent work on solving the ALL-SAT problem efficiently[11].

The *main contribution* of this paper is in presenting two new algorithms based on S&S, to compute flexibilities in multi-level Boolean networks. The flexibilities computed include SPFDs [30] and node resubstitution [10]. Both computations are known to be hard in practice. Experimental results show that S&S is an effective solution for these problems.

We also review a number of Boolean problems in synthesis and discuss how they might be solved efficiently using S&S without reporting any experimental results. These problems include:

- identifying functionally equivalent nodes in a network [7][14],
- computing don't-cares [21] for a node in a network,
- detecting variables in the true support of a node in a network,
- computing symmetries of Boolean functions, and
- checking the existence of a bi-decomposition [19].

The paper is organized as follows. Section 2 gives a brief background on simulation and SAT. Sections 3 and 4 discuss how S&S is used to compute SPFDs and derive resubstitutions, respectively. Experimental results are also given in these sections.

[1] This work was supported in part by the NSF grant CCR-9988402

Section 5 briefly reviews other applications of S&S in logic synthesis. Section 6 concludes and suggests directions for future work.

2. Background

2.1. Boolean network

Definition 2.1 A *Boolean network N* is a directed acyclic graph (DAG) such that for each node *i* in *N*, there is an associated representation of a Boolean function f_i and a Boolean variable y_i, where $y_i = f_i$. A node *i* is a fanin of a node *j* if there is a directed edge $\{i, j\}$ and a fanout if there is a directed edge $\{j, i\}$. A node *i* is a transitive fanin (TFI) of a node *j* if there is a directed path from *i* to *j* and a transitive fanout (TFO) if there is a directed path from *j* to *i*. The sources of the graph are the *primary inputs* (PIs) of the network; the sinks are the *primary outputs* (POs). The functionality of a node in terms of its immediate fanins is its *local function*. Its functionality in terms of the primary inputs of the network is its *global function*.

2.2. Simulation

Simulation computes the values of the internal signals of a network, given the values of the PIs. One *round* of simulation involves propagating one particular set of values through the network. Its complexity is linear in the network size.

We consider both *random* simulation where values of the PIs are assigned randomly, and *guided* simulation where PIs are assigned based on certain information, such as that provided by a SAT solver about assignments proving or disproving a property. Guided simulation effectively provides additional coverage beyond random simulation. Our applications currently use random simulation.

To ensure the efficiency of simulation of logic networks, we employ the following techniques in our implementation:

- Using AND-INV graphs (AIGs) [13] as our network representation. These are compact and homogenous. Simulating an AIG node involves bit-wise operations on the simulation information of the fanins.
- Performing simulation in a bit-wise parallel fashion, that is, simulating 32 or 64 bits simultaneously.
- Controlling the amount of simulation. Although the amount of random simulation performed can be determined dynamically by stopping when "saturation" is reached, in the experiments for this paper we use a fixed amount of simulation. The goal is to show that simulation works leaving fine-tuning for future investigation.
- Allocating memory for storing the simulation information in one large memory array. The nodes as well as the chunks of memory associated with the nodes are ordered topologically in a DFS order. This reduces the number of cache misses and improves the speed of simulation-intensive applications.

These techniques contribute differently to the efficiency of the overall approach. Both bit-wise parallel simulation and AIG representation lead to very noticeable speed-ups in computation. Controlling the amount of simulation is application-dependent and may lead to an additional 50% speed-up.

2.3. Satisfiability

Boolean satisfiability (SAT) [15][24] is a process of proving that a given Boolean formula has a satisfying assignment. Although solving a general SAT problem is NP-hard in the problem size, many practical problems have properties, which dramatically reduce the complexity. For example, if a SAT problem is formulated for a circuit, using the circuit structure can reduce the problem complexity [27].

The performance of SAT solvers has improved greatly in the last few years. State-of-the-art SAT solvers, such as [24][6], are based on techniques which dramatically speed up exploration of the search space. The following are the most successful techniques:

- *Non-chronological back-tracking* [15] is a way of exhaustively exploring the search tree by skipping some branches.
- *Dynamic variable ordering* [24]. The branching variable (on which the next decision is made) is determined based on the "activity" of variables. The more active variables participate more often in recent implications and conflicts.
- *Two-literal watching* [24]. The clause data-base is organized in such a way that only a small number of clauses are visited when literals are assigned, and no clauses are visited when literals are unassigned.

3. Computing SPFDs

SPFDs (Set of Pairs of Functions to be Distinguished) [37][30] express flexibilities of nodes in a multi-level network. They offer greater flexibility by allowing not only the node function to change, but also the function of its immediate fanout. SPFDs can be seen informally as representing the "information flow" in a network.

Modifying a node's local function while preserving the essential "information flow" may result in having to update all node functions in the transitive fanout of the node. Limiting the propagation of SPFDs to only a few levels of the TFO can control the size of the area of change. Compatible observability don't-cares (CODCs) [28] can be used to block the propagation of change.

SPFDs generalize observability don't cares (ODCs) but differ from the notion of multi-output Boolean relations [36]. Some of the applications of SPFDs are presented in [31][33]. The computation of SPFDs is complex and challenging. This section proposes an efficient way of computing SPFDs using on S&S.

The following presentation of SPFDs is based on [30].

3.1. Background

Consider two networks *N* and *N'* with identical structure. Let *X* and *X'* be their respective PIs, *Z* and *Z'* be their respective POs. For corresponding nodes *n* and *n'* in network *N* and *N'*, denote their output variables by *y* and *y'* and fanins by *Y* and *Y'*, respectively. Let the local and global functions at *n* and *n'* be given by $y = f(Y)$, $y = g(X)$ and $y' = f(Y')$, $y' = g(X')$, respectively.

Definition 3.1 The *global SPFD of* node *n*, $SPFD^n(X, X')$, is a Boolean function over the product of PI spaces $X \times X'$, taking value 1 for minterms *x* and *x'* iff the value computed by node *n* differs from the value computed by *n'*:

$$SPFD(X, X') = g(X) \oplus g(X').$$

Similarly, the *local SPFD* of node *n* is:

$$SPFD(Y, Y') = f(Y) \oplus f(Y').$$

For example, the local SPFD of an OR-gate is {(00,01), (00,10), (00,11)}, i.e. the off-set minterm (00) has to be distinguished from all the on-set minterms (01, 10, 11). Intuitively, a pair of minterms (*x, x'*) in an SPFD of a node is an elementary unit of information distinguished by the node, while the total SPFD of a node (the set of input minterms it can distinguish) represents its information processing capability.

Definition 3.2 A *cut C* of network *N* is a subset of nodes, such that every path from the POs to the PIs passes through at least one node in *C*.

Definition 3.3 A cut *C* is *redundant* if there exists $n \in C$ such that $C \setminus n$ is a cut. Otherwise, the cut is *irredundant*.

Definition 3.4 Let *C* be an irredundant cut containing node *n*. $C \setminus n$ is called a *separator* of *n*, denoted *Sep*(*n*).

Intuitively, a separator is a set of nodes whose information combined with the information from node *n*, contains the information required at the POs.

Definition 3.5 $Sep_2(n)$ *dominates* $Sep_1(n)$ if the union of SPFDs of nodes in $Sep_2(n)$ contains the union of SPFDs of nodes in $Sep_1(n)$:

$$\sum_{\alpha \in Sep_1(n)} SPFD^{\alpha}(X) \Rightarrow \sum_{\beta \in Sep_2(n)} SPFD^{\beta}(X)$$

Definition 3.6 *The largest* (*smallest*) *separator* of *n*, $Sep_{\max(\min)}(n)$, is the separator that dominates (is dominated by) all other separators of *n*.

The largest separator of *n* is composed of all PIs not in TFI(*n*) plus the nodes in TFI(*n*), which have a fanout outside TFI(*n*). The smallest separator is composed of all POs not in TFO(*n*) plus any node with a fanout to TFO(*n*)*n*.

Definition 3.7 The *minimum global SPFD* of *n* with respect to a separator σ = *Sep*(*n*), $SPFD_{\sigma}^{n}(X, X')$, is the set of pairs of minterms of the global SPFDs of the POs not contained in the SPFDs of nodes in the separator:

$$SPFD_{\sigma}^{n}(X, X') = \sum_{\alpha \in PO} SPFD^{\alpha}(X, X') \wedge \overline{\sum_{\beta \in \sigma} SPFD^{\beta}(X, X')}.$$

Thus minterm pair (*x, x'*) belongs to the mininum global SPFD of node *n* with respect to some separator σ. if *x* and *x'* are distinguished by at least one PO but not by any of the nodes in σ. Intuitively, σ is a barrier through which information must flow to get to the $PO \in TFO(n)$, and the minimum global SPFD is the information provided by *n,* but not available at any node of σ.

Definition 3.8 The *minimum local SPFD* of *n* with respect to a separator σ = *Sep*(*n*), $SPFD_{\sigma}^{n}(Y, Y')$, is the image of $SPFD_{\sigma}^{n}(X, X')$ in the local space of node *n*:

$$SPFD_{\sigma}^{n}(Y, Y') = \exists_{X, X'} SPFD_{\sigma}^{n}(X, X') \wedge M(X, Y) \wedge M(X', Y')$$

where *M*(*X, Y*) and *M*(*X',Y'*) are mappings from the PI spaces into the fanin spaces *Y* and *Y'* of nodes *n* and *n'* in networks *N* and *N*'. Note that the minimum SPFD is always defined with respect to a particular separator σ.

Example 3.1. Consider the circuit shown in Figure 3.1. The largest separator of node g_1 is {*a, b, c*}. The global SPFD of z_1 is equal to {(1--, 0--)} (the minterms are in the form *abc*). The global SPFD of *a,b* and c are {(1--,0--)}, {(-1-, -0-)} and {(--1,--0)}, respectively. All the minterm pairs of the global SPFD of z_1 are contained in the global SPFD of *a*. Hence, the minimum SPFD of g_1 with respect to the largest separator is empty. However, the node is neither s-a-0 nor s-a-1 redundant. This is because nodes in the transitive fanout of g_1 depend on the specific information flow through node g_1.

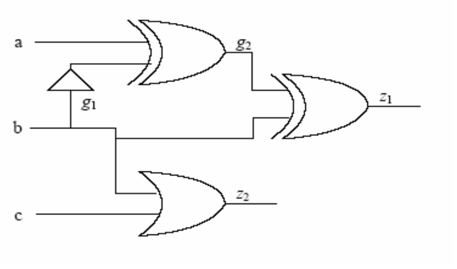

Figure 3.1. *Example circuit.*

The usefulness of the minimum SPFD is that the current function at a node *n* can be replaced by any function that contains the node's minimum SPFD. After the replacement, in order to preserve the functions of the primary outputs, the functions on the output side of σ may need to be changed. The property of minimum SPFDs guarantees that the new functions of these nodes can always be derived.

If the minimum SPFD of *n* is empty, then *n* does not provide any "useful" information not supplied by other nodes in σ. Therefore, *n* can be removed while other nodes can be re-synthesized so that the network's behavior remains unchanged. The procedures used to re-synthesize nodes are described in [30].

In theory, it can be proved that minimum SPFDs provide more optimization power than complete don't cares or ATPG. Some interesting applications of the minimum SPFD in logic synthesis include node optimization using the windowing concept proposed in [21] or combined rewiring and mapping during resynthesis optimizations.

3.2. Computing SPFDs using S&S

Although the computation of SPFDs can be implemented using BDDs by evaluating the formula in Definition 3.8, this limits the applicability of SPFDs to medium-sized circuits. In [32], an improvement computes the image using a SAT solver. Simulation was not used and the problem formulation was not tuned for SAT. As a result, although [32] can handle larger circuits than previous approaches based entirely on BDDs, its runtime is also larger. In this paper, we propose a much more efficient algorithm based on S&S to compute SPFDs. It offers better runtimes than the previous approaches and can be applied to larger circuits.

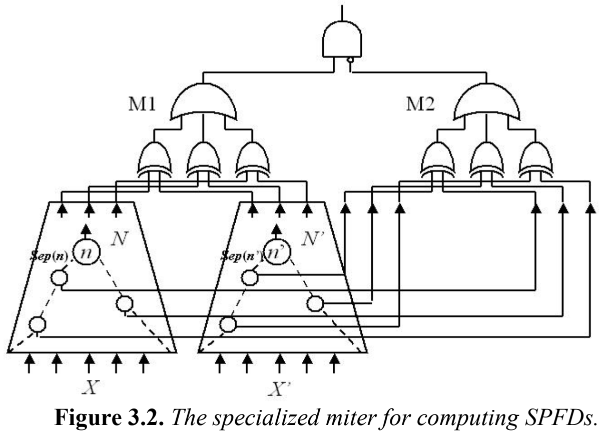

Figure 3.2. *The specialized miter for computing SPFDs.*

The minimum SPFD computation for a node initially builds a miter for the node. The concept of a miter was introduced in [2] and typically refers to a specialized circuit built such that the output evaluates to 0 if and only if some property holds. The following steps describe how the miter is constructed and used in the S&S-based minimum SPFDs computation:

- Construct the miter cone M1 as shown in Figure 3.2. The miter cone feeds PIs *X* into the first network *N* and PIs *X'* into the second network *N'*. In addition, identical separators σ of the node *n* and *n'* are introduced in both networks. The outputs of the nodes in the two separators are connected pair-wise to form the miter cone M2. The output of M2 is 1 if and only if the separator can distinguish a PI minterm pair (*x*, *x'*). Finally the M1 output is ANDed with the complement of the M2 output so the final output is 1 if and only if the POs of the network can distinguish (*x*, *x'*) but the separator cannot distinguish (*x*, *x'*). By construction, the final output is 1 only for the primary input minterm pairs that belong to the minimum SPFD of *n* with respect to separator σ.
- Perform bit-parallel simulation on the miter cone network. Assign random simulation vectors at the PIs and propagate them to the POs. Each pair of simulated minterms *x* and *x'* in the primary input space has a corresponding pair of local minterms *y* and *y'* in the local space *Y* and *Y'*. If the output of the network is 1, then the minterm composed of *y* and *y'* belongs to $SPFD_{\sigma}^{n}(Y,Y')$. Currently a static simulation model is used where a fixed number of input vectors are tried. However, a dynamic model might also be used where simulation is continued until it saturates, i.e. no new care minterms are found.
- Convert the miter cone network into a SAT instance. The local SPFD minterms (in *Y,Y'*) already computed (which belong to the minimum SPFD) are complemented and added to the SAT instance as breaking clauses, and SAT is used to enumerate through the remaining satisfying (*Y,Y'*) assignments. When it returns "unsatisfiable", the complete minimum SPFD of the node is obtained. A more efficient implementation of ALL-SAT can be used [11].

3.3. Experiment results

Both the S&S-based and BDD-based computation of the minimum SPFDs with respect to the *smallest* separator, of each node in the network, were implemented in SIS [29]. The smallest separator is more useful for network optimization because it allows for an efficient control of changes performed on the network after resynthesis of the node. These changes are limited to the transitive fanout cone of the node that is being resynthesized. The detailed discussion about limiting the area of change and modifications to the fanout nodes can be found in [30].

For S&S, the miter cone network was generated for each node in the circuit and the minimum local SPFD was computed using S&S. The BDD-based computation used BDDs for all the computations, including building the global SPFDs of the PO nodes and the nodes of the separator, constructing the minimum global SPFD of the node, and the image computation.

In most examples, the limiting factor of the BDD-based method was the construction of the minimum global SPFD of the node. This implies that a combination of simulation and BDDs is not suitable for computing the minimum SPFDs of nodes in the network. Even if all the minterm pairs in a minimum SPFD can be enumerated using simulation, the BDD-based computation needed for ensuring the completeness of the minimum SPFD computation could still encounter the same blowup problems.

The experiments were performed on a 4X "400MHz UltraSparc II" with 4.0GB of RAM. The results comparing the S&S-based and the BDD-based computation of the minimum SPFDs are shown in Table 3.1. Columns 1 and 2 report the name and the number of network nodes in each circuit, respectively. The minimum SPFDs were computed for all network nodes. Columns 3 and 4 report the resulting runtimes for the BDD-based and the S&S-based implementations, respectively. The overall time limit was set to 4,000 seconds for each circuit. The gain in runtime is reported in Column 5.

Circuit	#Nodes	BDD(s)	S&S(s)	Gain	Quitted (BDD)	Quitted (SAT)
dalu	1131	>4000	291.4	>13.7	737	661
frg2	522	1769.2	119.4	14.9	75	135
pair	824	323.9	97.8	3.3	289	119
C499	202	44.9	14.9	3.0	202	133
C880	357	3515.1	20.1	174.9	245	87
C1355	514	1813.6	64.2	28.2	514	314
C1908	880	3980.4	177.7	22.4	638	492
C3540	1667	>4000.0	554.4	>7.2	1663	1192
C6288	2416	>4000.0	2446.7	>1.6	2416	2312
C7552	3266	>4000.0	1651.4	>2.4	3256	1363
C5315	2288	>4000.0	798.7	>5.1	2247	0
i10	2488	>4000.0	1070.3	>3.7	2337	1117
Ave.				> 23.0		

Table 3.1. *Runtime Statistics of S&S-based versus BDD-based minimum SPFD computation.*

In the results shown above, S&S demonstrates an average performance improvement of at least 23x compared to BDDs. In general, the gain of S&S is much more, since the runtime of the BDD-based implementation for some of the larger circuits is much larger than the 4000 sec. timeout limit. The BDD-based implementation is very slow for large circuits and hence limitations on the size of the intermediate BDDs had to be introduced. The largest size, to which an intermediate BDD is allowed to grow, was set to 50,000 BDD nodes. To ensure some fairness in comparison, the limit on the number of backtracks in the SAT solver was set to 200. The number of network nodes, for which these resource limits were reached, is reported in Columns 6 and 7 for BDDs and S&S, respectively. In addition, Column 6 includes the number of nodes that could not be processed when the total runtime of the BDD-based computation exceeded the 4,000 sec. timeout. These numbers indicate that the BDD-based computation is unable to process a larger number of nodes than the S&S-based implementation for the same design. This could become an issue when the minimum SPFDs are being used for network optimization since the BDD-based approach might miss some optimization opportunities.

The results in Table 3.2 summarize the contribution of simulation to the S&S-based minimum SPFD computation. Column 1 reports the name of each circuit. Column 2 reports the total number of minterm pairs in the minimum SPFDs of the nodes in the circuit. The number of minterm pairs identified by simulation is reported in Column 3. The runtimes of simulation and SAT in the S&S-based implementation are presented in Columns 4 and 5, respectively.

Circuit	#Total minterm pairs	#Minterm pairs using SIM	SIM Runtime (s)	SAT Runtime (s)
dalu	3088	1329	5.8	288.0
frg2	8982	2867	35.4	84.6
pair	8450	3519	20.2	78.5
C499	458	178	2.4	12.2
C880	1584	697	4.7	15.2
C1355	596	238	5.9	57.9
C1908	2542	664	10.3	167.3
C3540	1278	615	10.2	540.4
C6288	324	162	20.3	2256.2
C7552	8758	3771	31.1	1593.9
C5315	5600	2522	22.6	764.3
i10	9896	2935	36.7	1025.7

Table 3.2. *Contribution of simulation to S&S-based minimum SPFD Computation.*

The results above show that simulation can identify about 40% of all minterm pairs in the minimum SPFD and contributes to a small fraction of the total runtime. Thus, simulation plays a significant role in the S&S-based minimum SPFD computation. However, simulation has a worst-case quadratic complexity in the number of patterns being simulated because it is necessary to compare the pairs of the values of the POs and the separator outputs. Hence, the simulation used in this application needs to be carefully tuned to maintain the runtime improvements.

The two experiments demonstrate that S&S is better suited than a pure BDD engine for the efficient computation of minimum SPFDs, even for large circuits. In future work, we intend to use this to better exploit the added flexibility provided by the minimum SPFDs in an optimization framework.

4. Computing Resubstitutions

This section describes an S&S-based algorithm to compute sets of nodes useful for Boolean resubstitution. Resubstitution plays an important role in technology-independent [3] and technology-dependent [17] logic synthesis. In particular, we focus on speeding up resubstitution used in the re-synthesis flow [12].

4.1. Background

Definition 4.1 *Resubstitution* of a node in the network replaces the node's local function by a new local function, which depends on a *different* set of fanins, but does not alter the functionality of the network.

Resubstitution can be used to restructure the network to minimize delay, area, routeability, or some other cost function. Delay and routeablilty can be improved if the new fanin(s) arrive earlier or are closer than those replaced. Area can be improved if after resubstitution, some of the old fanins of the node have no fanouts and, therefore, can be removed from the network.

The existence of a resubstitution is closely related to the concept of functional dependency [10]. The definitions and the theorems below are taken from that work, followed by a novel computation procedure, which relies on S&S rather than BDDs, as in [10]. Another way of computing resubstitution functions is using interpolation [18].

Definition 4.2 Given node n with global function $g(X)$ and nodes $m_1, m_2, \ldots, m_k$ with global functions $y_{m1}(X), y_{m2}(X), \ldots, y_{mk}(X)$, nodes $m_1, m_2, \ldots, m_k$ can resubstitute node n if the global function of n can be expressed as follows:

$$g(X) = F(y_{m1}(X), y_{m2}(X), \ldots, y_{mk}(X)),$$

where $F(y_1, y_2, \ldots, y_k)$ is some Boolean function called a *resubstitution function*.

Theorem 4.1 Nodes $m_1, m_2, \ldots, m_k$ with global functions $y_{m1}(X), y_{m2}(X), \ldots, y_{mk}(X)$ can resubstitute node n with global function $g(X)$ if and only if there does not exist a minterm pair (x_1 , x_2), such that $g(x_1) \neq g(x_2)$ but $y_{mj}(x_1) = y_{mj}(x_2)$, for all j: $1 \leq j \leq k$.

Thus, resubstitution is possible if and only if the distinguishing power (information) required of g (output) is not greater than the union of the distinguishing powers of all the functions y_{mj} (inputs).

Example 4.1. Suppose the global function of node n is $g = (a \oplus b)(b \vee c)$, where a, b and c are the PIs. Consider two sets of resubstitution candidates with global functions: ($y_1 = \overline{a}b$, $y_2 = a\overline{b}c$) and ($y_3 = a \vee b, y_4 = bc$). Table 4.1 shows the truth tables of all the global functions. The set (y_3, y_4) is not a valid resubstitution candidate for g because minterm pair (101, 110) is distinguished by g but not by either y_3 or y_4. On the other hand, the set (y_1, y_2) satisfies Theorem 4.1 because all the minterm pairs that are distinguished by g are also distinguished by at least one node in the set.

a b c	g	Set 1		Set 2	
		$y_1=\overline{a}b$	$y_2=a\overline{b}c$	$y_3=(a+b)$	$y_4=bc$
000	0	0	0	0	0
001	0	0	0	0	0
010	1	1	0	1	0
011	1	1	0	1	1
100	0	0	0	1	0
101	1	0	1	1	0
110	0	0	0	1	0
111	0	0	0	1	1

Table 4.1. *Truth table of g and the candidate sets.*

Theorem 4.2 Let $Y = \{y_1, y_2,\ldots,y_k\}$. Node n with global function $g(X)$ can be resubstituted using nodes $m_1, m_2, \ldots, m_k$, each with global functions $y_{m1}(X), y_{m2}(X), \ldots, y_{mk}(X)$, iff $F^{on}(Y)F^{off}(Y) = 0$, where:

$$F^{off}(Y) = \exists_X [\overline{g}(X) \wedge \prod_{j=1}^{k} y_j \equiv y_{mj}(X)],$$

$$F^{on}(Y) = \exists_X [g(X) \wedge \prod_{j=1}^{k} y_j \equiv y_{mj}(X)]$$

If this property holds, then any function $F(Y)$ that agrees ($F^{on}(Y) \subseteq F(Y) \subseteq \overline{F^{off}(Y)}$) with the above incompletely specified function (represented by its on-set and off-set) could be the new resubstitution function for node n.

Example 4.2. Using the same functions as in the above example, we compute the on-set and the off-set based on Theorem 4.2. with respect to local nodes y_1 and y_2: $F^{off}(Y) = \overline{y}_1\overline{y}_2$ and $F^{on}(Y) = \overline{y}_1 y_2 \vee y_1\overline{y}_2$. Minimizing this incompletely specified Boolean function, we derive the simplest resubstitution function for node n using candidates y_1 and y_2: $g = y_1 \vee y_2 = \overline{a}b \vee a\overline{bc}$.

4.2. Computation using S&S

Given a node and a set of resubstitution candidate sets (derived using the structural support of $g(X)$), the goal is to determine those candidate sets that can be used for resubstitution. The following steps describe the computation of feasible candidate sets:

1. Assign pairs of random vectors at the PIs corresponding to minterms x_1 and x_2. Perform bit-parallel simulation of the network and compare the outputs for functions g and resubstitution candidates $y_1, y_2, \ldots, y_k$. Using Theorem 4.1, filter out the candidate sets that cannot be used for resubstitution. Repeat random simulation for a pre-determined number of rounds.
2. Apply SAT to each of the remaining candidate sets. SAT is used to compute the on-set and off-set of F (the images) in Theorem 4.2 by enumerating the satisfying assignments that were not filtered out by simulation. (The combinations that appear during simulation at the root node and the candidate nodes are complemented and added to the solver as breaking clauses). If the images computed by SAT satisfy Theorem 4.2, resubstitution exists, and the resubstitution function is found by minimizing the derived incompletely specified function.

4.3. Experimental results

The S&S-based and BDD-based resubstitution algorithms were implemented in the resynthesis package of MVSIS [25]. The following experiments were done using a Pentium 4 computer with 1.8GHz CPU and 512MB RAM. The benchmarks were read into MVSIS followed by technology mapping [23] using *mcnc.genlib* from the SIS distribution [29]. Next the netlist was re-synthesized to reduce area, with a total runtime limit of 3 minutes for the resubstitution of all nodes in the network. During the resynthesis step, the S&S-based and BDD-based resubstitution computations were compared for runtimes. The results of both computations were verified against each other.

Table 4.2 gives the experimental results on a set of MCNC [38], ITC [9], ISCAS [4] and PicoJava [35] benchmarks. Columns 1 and 2 list the names and the characteristics of each benchmark. Column 3 is the runtime for the BDD-based approach. Columns 4, 5 and 6 give the runtimes (simulation, SAT, and total) for the S&S-based approach. The runtimes include only the time to filter out invalid candidates and do not include the runtime to select the initial candidate sets or to perform the actual resubstitution after a valid candidate set is chosen. Table 4.2 shows an average 29x performance improvement of the S&S-based approach over the BDD-based approach.

To give more insight into the S&S-based resubstitution in these experiments, additional statistics are given in Table 4.3. Column 2 gives the total resynthesis runtime. Since the runs are limited to 3 minutes, for benchmark *b17*, resynthesis did not complete for the whole circuit. However, the area improvements and the resubstitution set counters are valid. Column 3 is the percentage of area improvement due to re-synthesis. The total number of sets of candidates (over all nodes) considered is reported in Column 4. Column 5 (DSIM) and Column 6 (DSAT) give the percentages of the candidate sets filtered out by simulation and SAT respectively. Column 7 (PSAT) reports the percentages of the candidate sets that SAT proved to be useful for substitution. The last column (USAT) is the percentage of candidate sets not proved/disproved by SAT due to a resource limit on the number of backtracks. Table 4.3 shows that an average 94% of the candidate sets are filtered out by simulation, indicating that simulation plays a significant role in S&S.

Name	**In/Out/ Latch**	**BDD (s)**	**Simulation & Sat (s)**			**Gain**
			SIM	**SAT**	**Total**	
dalu	75/16/0	78.74	1.38	2.12	3.50	22.5
des	256/245/0	104.66	3.48	12.75	16.23	6.5
frg2	143/139/0	30.78	1.57	2.42	3.99	7.7
i10	257/224/0	172.99	2.24	2.34	4.58	37.8
k2	45/45/0	53.41	1.99	9.38	11.37	4.7
pair	173/137/0	75.44	1.52	1.27	2.79	27.0
C432	36/7/0	83.00	0.36	0.29	0.65	127.7
C2670	233/140/0	48.04	0.69	1.28	1.97	24.4
C5315	178/123/0	109.95	1.13	1.86	2.99	36.8
C7552	207/108/0	145.04	2.68	6.11	8.79	16.5
s15850	14/87/597	170.60	2.80	3.45	6.25	27.3
s35932	35/320/1728	40.45	1.40	1.28	2.68	15.1
pj1	1769/1063/0	163.90	3.98	4.82	8.80	18.6
b14	32/54/245	173.46	1.72	2.86	4.58	37.9
b17	37/97/1414	274.01	2.46	3.93	6.39	42.9
b20	32/22/490	166.99	2.46	6.17	8.63	19.4
b22	32/22/703	169.49	1.81	3.98	5.79	29.3
Ave.						**29.5**

Table 4.2. *Runtime comparison of S&S-based vs. BDD-based resubstitution computation.*

Name	**Run Time (s)**	**Area Imp. (%)**	**Total cand. sets**	**DSIM (%)**	**DSAT (%)**	**PSAT (%)**	**USAT (%)**
dalu	4.82	13.96	417,162	94.68	1.12	4.20	0.00
des	17.74	2.94	731,665	89.32	9.82	0.83	0.02
frg2	4.53	15.41	341,118	92.61	0.04	7.33	0.01
i10	7.61	7.58	825,859	97.35	1.31	1.33	0.01
k2	11.12	8.09	313,343	83.21	14.14	2.65	0.00
pair	3.23	12.35	399,039	96.06	0.00	3.94	0.00
C432	0.84	7.68	97,375	96.89	1.79	1.32	0.00
C2670	2.18	20.98	150,979	93.44	2.62	3.86	0.08
C5315	3.86	13.15	369,656	96.21	0.25	3.62	0.01
C7552	9.87	15.83	578,142	92.12	3.73	4.10	0.05
s15850	9.75	11.45	825,209	95.16	2.64	2.20	0.01
s35932	5.71	3.21	451,714	96.79	0.00	3.21	0.00
pj1	66.79	10.29	3,988,250	95.00	2.99	2.00	0.01
b14	25.56	5.17	2,367,501	96.64	2.36	0.97	0.02
b17	180.67	8.44	9,050,793	93.20	5.73	1.05	0.02
b20	61.93	7.52	5,672,596	96.95	1.64	1.38	0.03
b22	101.04	7.14	8,130,898	96.69	2.00	1.26	0.05
Ave.		**10.07**		**94.25**	**3.07**	**2.66**	**0.02**

Table 4.3. *Resynthesis statistics.*

Tables 4.2 and 4.3 together demonstrate the power of using S&S to solve this computationally hard problem in logic synthesis.

We also performed an experiment to compute resubstitutions using simulation and BDDs, instead of SAT. BDDs are used to check the validity of the remaining 6% candidate sets after simulation, as opposed to the case reported in Table 4.2 when BDDs are used to check all candidate sets. In terms of runtime, the overall results are: on the set of 17 benchmarks, BDD wins in 5 cases and SAT wins in 12 cases. The average SAT/BDD runtime ratio is 0.79. This demonstrates that SAT wins over BDDs in performance alone, without the advantages of simulation.

5. Other Applications of S&S in Logic Synthesis

S&S can be used in a variety of other applications in synthesis. In this section, for completeness, even though the experiments are not part of this paper, we discuss a number of these applications. Some have been experimented with in other papers while others remain for future experimentation.

5.1. Detecting functionally equivalent nodes

Detecting functionally equivalent nodes is important for both formal verification and logic synthesis. It simplifies the verification problem to be solved and reduces the area of the implemented circuit. Many techniques, such as BDD sweeping [13] or SAT sweeping [14], have been used to achieve functional reduction as a post-processing step after the initial construction of the AIG graphs.

In [22], an algorithm was proposed based on S&S, to perform functional reduction during the AIG construction. Random simulation is employed to create the simulation signature for each node. When a new node is to be added to the AIG, the signature is checked to see if it is unique. If it is, the node is added to the network. Otherwise, SAT is used to check if the new node is functionally equivalent to the nodes with the same simulation signature. Experiments show that S&S can sweep away a substantial amount of duplicated logic and that this improvement is orthogonal with respect to other area optimization techniques.

5.2. Computing internal don't-cares

Optimization of Boolean networks using don't-cares [28][20] plays an important role in technology independent logic synthesis and incremental re-synthesis of mapped netlists. In [21], an efficient method was proposed to compute complete don't-cares (CDCs) based on S&S. The algorithm uses a windowing scheme to trade off quality and runtime. Random simulation with a fixed number of simulation patterns is used to detect a subset of care minterms. Then, a SAT solver enumerates through the remaining satisfying solutions of the resulting problem representing the remaining part of the care set. The complete don't-care sets are derived based on both simulation and SAT results.

Experiments show that the algorithm is faster than a similar one based on BDDs. This makes it possible to compute CDCs for large designs leading to the enhanced optimization quality and reduced runtime, making the use of CDCs affordable for industry designs.

5.3. Detecting the true support of a node

Definition 5.1 The *functional support* of a node in a network is the set of primary inputs, such that the global function of the node depends on each input.

Functional support is useful when Boolean functions are represented as multi-level circuits. If a PI is not in the functional support of a node, the circuit representing the node could be simplified by propagating a constant instead of the PI. However, in general, this PI cannot be removed from the network because it may belong to the functional support of other nodes.

Simulation can be used to show that some variables belong to the functional support. For each random pattern simulated, a pattern with a complemented value for a given variable is also simulated. If the outputs are different for at least one pattern, then the variable belongs to the functional support. When simulation saturates, the remaining variables are processed by SAT. In this processing, the positive and negative cofactors of the function with respect to each undecided variable are derived and the miter circuit consisting of an XOR gate, that has the two cofactors as its inputs, is constructed. If no satisfying assignment exists to cause the output of the miter to be 1, then the variable does not belong to the functional support.

5.4. Detecting symmetries

Many applications in EDA exploit symmetric functions to achieve better results and/or improve performance.

Definition 5.2 Boolean function $F(\dots x_i, \dots, x_j, \dots)$ is *symmetric* in variables x_i and x_j if and only if

$$F(\dots x_i, \dots, x_j \dots) = F(\dots x_j, \dots, x_i, \dots).$$

Theorem 5.1. Let F_{00}, F_{01}, F_{10} and F_{11} be the four cofactors of F with respect to variables x_i and x_j. The symmetry in variables x_i and x_j exists iff $F_{01} = F_{10}$.

Simulation can be used to show that some pairs of variables are not symmetric. Random simulation is inefficient because we need two input patterns that are swapped only in the values for variables x_i and x_j to determine if they are not symmetric. Guided simulation based on this requirement finds many non-symmetric pairs. When SAT is invoked on the remaining pairs, two cofactors with respect to two variables, F_{01} and F_{10}, are compared by constructing a miter similar to the one in Section 5.3, and solving the resulting SAT problem.

A similar approach to the computation of symmetries using simulation and ATPG was proposed in [26].

5.5. Detecting combinational redundancies

Detecting combinational redundancies (untestable faults) is a well-studied area [16][34]. Random simulation is used to filter out signals that are not redundant. If simulation cannot prove this, SAT is employed. The redundant signals can be replaced by constants in the netlist, resulting in savings.

5.6. Detecting feasible bi-decompositions

A method for performing Boolean decomposition of incompletely specified functions proposed in [19] is known as *bi-decomposition* because it decomposes the function into two logic nodes, whose outputs feed into a two-input gate (AND, OR or EXOR). This method produces good quality decompositions by propagating don't-cares through the decomposed network and caching intermediate results.

The usefulness of bi-decomposition is limited because of its prohibitive runtime. The bottleneck of this method is in computing the subsets of variables, for which bi-decomposition is feasible. This computation requires evaluating Boolean formulas with quantifiers, which is done in [19] using BDDs. Our recent experience suggests that S&S might provide a better solution for this task, which may substantially speed-up bi-decomposition. Providing a comprehensive S&S formulation of bi-decomposition and evaluating it experimentally is deferred to the future work.

6. Conclusions and future work

We demonstrated that computationally hard problems in logic synthesis could be solved more efficiently using a combination of simulation and Boolean satisfiability. Two applications were studied in detail: the computation of SPFDs, and the derivation of sets of nodes that can be used to resubstitute a node in a multi-level network. We also discussed briefly how a few other applications have been or could be solved using S&S.

Future work will include evaluating the use and impact of these computed flexibilities after technology independent optimization and technology mapping, and developing new ways of using S&S in re-synthesis. Our experience suggests that S&S can replace BDDs in many applications in logic synthesis.

References

[1] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient implementation of a BDD package", *Proc. DAC '90*, pp. 40-45.

[2] D. Brand, "Verification of large synthesized designs". *Proc. ICCAD '93*, pp. 534 -537.

[3] R. K. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.

[4] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," *Proc. ISCAS '89.*

[5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE TC*, vol. C-35(8), Aug 1986, pp. 677-691.

[6] N. Eén, N. Sörensson, "An extensible SAT-solver", *Proc. SAT '03.* http://www.cs.chalmers.se/~een/Satzoo/

[7] F. Lu, L. Wang, K. Cheng, R. Huang. "A circuit SAT solver with signal correlation guided learning". Proc. DATE '03, pp. 892-897.

[8] E. Goldberg, M.Prasad, R.K.Brayton. "Using SAT for combinational equivalence checking". *Proc. DATE '01*, pp. 114 -121.

[9] *ITC '99 Benchmarks* http://www.cad.polito.it/tools/itc99.html

[10] J.-H. R. Jiang, R. K. Brayton, "Functional dependency for verification reduction", *Proc. CAV '04,* pp. 268-280.

[11] H.-S. Jin, H.-J. Han, F. Somenzi, "Efficient conflict analysis for finding all satisfying assignment of a Boolean circuit", *Proc. TACAS '05* (eds. N. Halbwachs and L. Zuck), LNCS 3440, pp. 287-300.

[12] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization", *Proc. DAC '04*, pp. 438-441.

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification", *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.

[14] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking". *Proc. ICCAD'04*, pp 50-57.

[15] J. P. Marques-Silva, K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Trans. Comp*, Vol. 48(5) , May 1999, pp. 506-521.

[16] T. Larrabee, "Efficient generation of test patterns using Boolean difference", *Proc. Intl. Test Conference '89*, pp. 795-801.

[17] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. CAD*, 16(8), 1997, pp. 813-833.

[18] K.L. McMillan, "Interpolation and SAT-based model checking". *Proc. CAV '03*, LNCS 2725, Springer, 2003, pp. 1-13.

[19] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," *Proc. DAC '01*, pp. 103-108.

[20] A. Mishchenko, R. K. Brayton. "Simplification of non-deterministic multi-valued networks". *Proc. ICCAD '02*, pp. 557-562.

[21] A. Mishchenko, R. K. Brayton. "SAT-based complete don't-care computation for network optimization". *Proc. IWLS'04,* pp.353-360.

[22] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton. "FRAIGs: A unifying representation for logic synthesis and verification". *Technical Report*, UC Berkeley, 2004. http://www.ece.pdx.edu/~alanmi/publications/fraigs08_full.pdf

[23] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, T. Kam. "Reducing structural bias in technology mapping". To appear in *Proc. IWLS '05*

[24] M. Moskewicz, C. Madigan, Y. Zhao, L.Zhang, S. Malik. "Chaff: engineering an efficient SAT solver". *Proc. DAC '01*, pp. 530–535.

[25] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[26] I. Pomeranz and S.M. Reddy, "On determining symmetries in inputs of logic circuits", *IEEE Trans. CAD*, vol. 13(11), Nov. 1994, pp. 1428-1434.

[27] M. Prasad, P. Chong and K. Keutzer, "Why is ATPG Easy?", *Proc DAC*'99, pp. 22-28.

[28] H. Savoj, R. K. Brayton, "The use of observability and external don't-cares for the simplification of multi-level networks". *Proc. DAC' 90*. pp. 297-301.

[29] E. Sentovich, et al. "SIS: A system for sequential circuit synthesis", *Tech. Rep. UCB/ERI*, M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.

[30] S. Sinha and R. K. Brayton, "Implementation and use of SPFDs in optimizing Boolean networks", *Proc. ICCAD '98*, pp. 103-110.

[31] S. Sinha, S. Khatri, R. K. Brayton and A. Sangiovanni-Vincentelli, "Binary and multi-valued SPFD-based wire removal in PLA networks", *Proc. ICCD '00*, pp. 494-503.

[32] S. Sinha and R. K. Brayton, "Improved robust SPFD computations", *Proc. IWLS '01*, pp. 156-161.

[33] S. Sinha, A. Mishchenko and R.K. Brayton, "Topologically constrained logic synthesis", *Proc. IWLS '02*, pp. 13-20.

[34] P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. CAD*, vol. 15(9) , September 1996, pp. 1167-1176.

[35] SUN Microelectronics. *PicoJava Microprocessor Cores*. http://www.sun.com/microelectronics/picoJava/

[36] Y. Watanabe, L. Guerra and R. K. Brayton, "Logic optimization with multi-output gates", *Proc. ICCD '93*, pp. 416-420.

[37] S. Yamashita, H. Sawada, A. Nagoya, "A new method to express functional permissibilities for LUT based FPGAs and its applications". *Proc. ICCAD '96*, pp. 254-261.

[38] S. Yang. *Logic synthesis and optimization benchmarks*. Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.