Integrating an AIG Package, Simulator, and SAT Solver

Alan Mishchenko Robert Brayton

Department of EECS, UC Berkeley

{alanmi, brayton}@berkeley.edu

Abstract

This paper focuses on problems where the interdependence of simulation and Boolean satisfiability (SAT) is critical. We propose a modified AIG logic data-structure to optimize speed and scalability for large problems of this type. Experimental results confirm that the new implementation is faster compared to the old implementation where runtime and scalability has been a known issue.

1. Introduction

Many applications in logic synthesis and formal verification rely on an integration of incremental Boolean satisfiability (SAT) solving plus random/guided simulation.

In a typical scenario, the simulator helps the SAT solver skip the majority of satisfiable runs saving a lot of runtime. In addition, when the SAT solver finds a counter-example it can be passed to the simulator, for later use in upcoming incremental SAT problems.

This synergy between the simulator and the SAT solver is exploited in many applications, e.g. 1) detecting and proving equivalent nodes in a logic network (aka SAT sweeping [7][10]), 2) sequential signal correspondence [11], 3) computing structural choices to improve the quality of technology mapping [2], 4) transferring net names between netlists if the names are lost during synthesis, etc. These are important practical applications and therefore improving their runtime and scalability is well-motivated.

The main contributions of this paper are:

- a better understanding of how simulation and SAT should be orchestrated, leading to
- a new And-Inverter Graph (AIG) package that includes additional bookkeeping to make simulation and SAT faster and more scalable.

One application that relies on the integration of simulation and SAT, is *SAT sweeping*, which is an engine that detects, proves, and merges (or collects) functionally equivalent nodes in the AIG. If these are merged, the engine works as a combinational or sequential optimization. If the nodes are collected, the engine provides structural choices useful for technology mapping. With minor modifications, the same engine can transfer signal names across two functionally equivalent versions of the netlist, to perform redundancy removal and don't-care-based optimization, etc.

In the experiments, the resulting engine was used to compute sequential equivalences and to accumulate structural choices. These applications were selected because they often dominate the runtime for large designs in a typical synthesis/verification flow implemented in ABC [1].

The rest of the paper is organized as follows. Section 2 contains relevant background. Section 3 discusses the proposed framework and its components. Section 4 contains experimental results. Section 5 concludes the paper.

2. Background

2.1 Boolean function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function f is the set of variables X, which influence the output value of f. The *support size* is denoted by $|X|$.

2.2 Boolean network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes.

A node n may have zero or more *fanins*, i.e. nodes driving n, and zero or more *fanouts*, i.e. nodes driven by n. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A *transitive fanin (fanout) cone* (TFI/TFO) of a node is a subset of nodes of the network, which are reachable through the fanin (fanout) edges of the node. The *TFO support* of a node is the set of primary outputs reachable through the fanouts of the node.

A *window* is defined as a subset of nodes of the network taken together with their TFI cones. In some cases, a window defines the scope where a dedicated computation, such as simulation, takes place. In other cases, the window nodes are copied to a separate network, new node attributes are added, and the resulting network is used by an engine, such as a SAT solver, to perform local computations. In both cases, the window size determines the complexity of the computation and can be used to trade quality for runtime.

2.3 And-inverter graph

An *and-invertor graph* (AIG) is a combinational Boolean network composed of two-input AND gates and inverters. One way to derive an AIG for a general Boolean network, is to factor the functions of logic nodes. Then the AND/OR gates in the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule and added to the AIG in a topological order.

2.4 Boolean satisfiability

A *Boolean satisfiability problem* (SAT) is a decision problem that takes a propositional formula representing a Boolean function and answers the question whether the formula is satisfiable, that is, whether the function is not a

constant 0. The formula is *satisfiable* (SAT) if there is an assignment of variables, called a *counter-example* (CEX), that makes the formula evaluate to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A *SAT solver* is a software program that solves SAT problems.

2.5 Conjunctive normal form

To use the SAT solver, important aspects of the problem are encoded using Boolean *variables*. Presence or absence of a given aspect of the problem is represented by a positive or negative *literal* of the variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a conjunctive normal form (CNF), or a *SAT instance*. The CNF is loaded into the SAT solver and manipulated by it to determine whether the CNF is *satisfiable* or *unsatisfiable*.

2.6 Circuit-based SAT solving

In logic synthesis and formal verification, which deal with hardware designs, a large portion of the CNF is often derived from the circuit representation of the design. This motivates the development of *circuit-based SAT solvers*, which perform constraint propagation directly on the circuit and reserve the CNF representation for application-specific non-circuit-based constraints and learned clauses [4][6][8][12].

In a circuit-based solver, node IDs play the role of SAT variables and constraints are propagated by visiting fanins/fanouts of the nodes that have been assigned a value.

An important aspect of a circuit-based SAT solver that has no analogues in CNF-based SAT solvers, is the *J-frontier*. Given a partial assignment of variables/nodes during SAT solving, consider nodes whose output value is currently not implied by the values of their fanins. These nodes are called *J-nodes* because they need justification by appropriately assigning fanin variables, if the given partial assignment is to become a complete satisfiable assignment. The set of all J-nodes at any time during solving is called the *J-frontier*.

In the case of an AIG composed of only two-input AND-nodes without XORs, the only type of a J-node is a two-input AND-node whose output has value 0 while its fanins are unassigned. In other cases, either the output value is already defined, or a new implication can be made.

J-frontier has two important applications. It allows for the solver to stop and return "satisfiable" when the J-frontier is empty and there are no unpropagated assignments. In many practical applications, this saves a lot of runtime compared to the CNF-based solver, which cannot return "satisfiable" until all variables have been assigned.

Another use of the J-frontier is to make decisions during SAT solving. J-frontier-based decisions can replace or be used interchangeably with other decision-making strategies.

2.7 Incremental SAT solving

Modern SAT solvers, such as MiniSAT [3], offer a mode of solving that allows executing multiple SAT calls without restarting the SAT solver. In this mode, the CNF first loaded into the SAT solver is reused with the possibility of adding new clauses or deleting old ones between the calls. To do this efficiently, the SAT solver accepts assumptions, which are single-literal clauses holding for one SAT call. Checking satisfiability of a SAT instance under different sets of assumptions is called *incremental SAT solving*.

3. Simulation/SAT ecosystem

Figure 1 shows the components of a typical ecosystem for SAT sweeping and the interaction between them.

The *SAT sweeping manager* coordinates and oversees all computations. It starts by allocating the components and computing candidate equivalence classes using several rounds of random simulation. During the main part of the computation, the SAT sweeping manager iterates over all nodes in a topological order and calls appropriate solvers. For example, if a node is part of an unproved equivalence class, the simulator is called to check whether the class can be refined, and if not, the SAT solver is called to check whether the node is equivalent to the representative of the class. Another role of the SAT sweeping manager is to define and enforce the resource limits, such as windows size, which allows the simulator and SAT solver to work on a relatively small subset of nodes of a large original AIG.

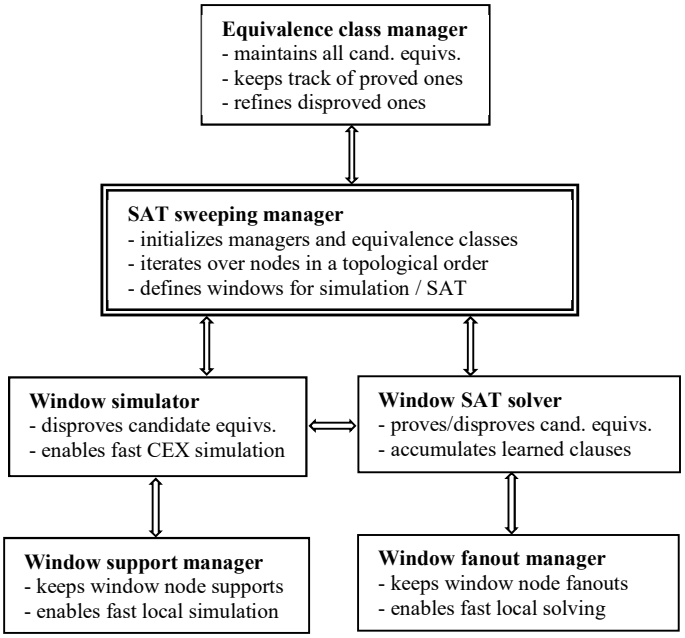

Figure 1. The proposed ecosystem.

3.1 Improved AIG manager

In this section, we present a simplified design of an efficient AIG package. Since it first appeared in academic publications two decades ago [5], the AIG representation has become a de facto standard for logic networks in a variety of applications. In particular, ABC features several AIG packages, which differ in terms of functionality, memory requirements, and the APIs provided.

In the proposed design of the AIG package, there is no separate node data-structure; instead, a node is an *integer ID*. A node augmented with an additional bit is an *AIG literal*; if 1, the literal is complemented; otherwise, uncomplemented.

An AIG node has two fanins represented as AIG literals. The four types of AIG nodes have fanins defined as follows: (1) the constant 0 node has no fanins, (2) primary inputs have no fanins; (3) an internal two-input node has two non-zero fanins; (4) a primary output has one fanin, which can be 0.

Recall that a zero literal stands for the non-complemented node with ID 0, which is the constant 0. An internal node in a constant-propagated AIG cannot have a constant 0 or 1 fanin because a constant would propagate and the node eliminated. Similarly, an internal node cannot have identical fanins. The fanins of the node are ordered as follows: if the first fanin ID is *smaller* than the second, the node is a two-input AND gate. Sometimes it is helpful to consider AIGs containing two-input XOR nodes. We can distinguish XORs from ANDs by changing the order of the fanins: in XORs, the first fanin ID is *larger* than the second.

Structural hashing is performed for the AIG to make sure that there is no structural duplication, that is, there is at most one node with the same two given fanins. For this, a hash table is used, in which the node fanins are hashed into the node ID. If two-input XOR nodes are present, they are hashed in the same hash-table as two-input AND nodes.

When the AIG package is allocated, the constant 0 node is created immediately, while other nodes are added incrementally by the calling application in a topological order. In particular, primary inputs can be added in any order as long as they are added before the nodes that have them as fanins. Similarly, primary outputs can be added in any order as long as their fanin nodes are already added.

Reducing *memory requirements* of the AIG manager is important because of the extremely large sizes of AIGs used in many practical applications. Another reason is that the smaller is the memory foot-print, the faster are AIG-based computations such as traversal, node labeling, level computation, simulation, etc.

Assuming 32-bit node IDs, the proposed simplified AIG package requires only two integers to represent fanins of a node (that is, memory use is 8 bytes/node). In this case, the AIG package can have close to 2 billion (2^31) nodes while one bit is reserved for complemented attributes. If structural hashing is enabled, two additional integer arrays are used for (1) the body of the chained hash table, and (2) the next node ID, linking nodes found in the same hash bin.

In summary, the minimalistic AIG package, with only fanins defined, takes only 8 bytes/node. With structural hashing, it takes 16 bytes/node. With structural hashing and fanout, as explained in Section 3.2, it takes 40 bytes/node. This is less than the AIG packages currently used in ABC.

3.2 Fanout manager

The *fanout manager* allows an application to quickly access the fanouts of an AIG node. The fanout information plays an important role in some AIG-based computations, such as rewriting, retiming, and SAT solving. In particular, a circuit-based SAT solver uses the fanout to propagate constraints by visiting fanouts of nodes recently assigned a value in the process of solving.

Ideally, the fanout representation should be (1) memory efficient with minimal memory fragmentation (that is, avoid frequent resizing of dynamic arrays, which would be needed, if fanouts of each node were stored in a separate array); (2) easy to add/remove nodes incrementally; (3) cache-friendly when iterating over fanouts of a node.

After experimenting with different fanout representations, the following was found to satisfy these requirements. For a fanout representation of an AIG with N nodes, three integer arrays of size N are allocated, requiring 12 bytes/node when 32-bit node IDs are used.

For each node, the third array stores the ID of its first fanout, while the first (second) array stores the next fanout of the first (second) fanin. In this case, the fanout IDs for a node are stored in a NULL-terminated linked list. The head of the list is found in the third array, while the first/second arrays contain IDs of the next fanout in the list. To iterate over all fanouts of a node, after reaching the first fanout f of node n, the next fanout of node n is found by checking what is the number of n among the fanins of f; if n is the first (second) fanin of f, the next fanout of n is found in the first (second) array used to store the fanout information.

This representation does not lead to memory fragmentation because only three arrays are allocated for a given AIG. The arrays can be static if no new nodes are created during the computation. Otherwise, the arrays are periodically resized to accommodate fanouts of the new nodes. Adding a new fanout is similar to adding a new entry into a linked list. Removal of fanouts is not required in applications that recycle logic windows and reconstruct fanout from scratch. Alternatively, we could perform linear-time fanout removal by searching it in the linked list, or enable constant-time removal by using two additional arrays containing previous entries, resulting in a double-linked linked-list.

3.3 Support manager

The *support manager* is an extension of the AIG package that keeps track of the structural primary-input support of the nodes. A node's support is represented as a positionally-encoded bit-string. A 1 in the i-th bit means that the i-th primary input is present in the node's support. The bit-strings are made unique by hashing. Each node is annotated with an integer ID of its structural support bit-string.

One concern is that this representation may take substantial memory for large windows with hundreds of primary inputs. However, in practice, the number of unique supports found in a typical window is relatively small and rarely exceeds 10% of the number of nodes in a window. As a result, the hashed bit-strings typically use a modest amount of memory.

In applications relying on the use of structural support, the support manager is created as soon as the AIG manager is initialized. The manager is updated when new AIG nodes are incrementally added to the window.

Support as bit-strings (compared to support as arrays of sorted node IDs) enables faster checking if a node has a given input in its support. This type of query is frequently performed by various AIG-based computations. For example, a simulator checks the node's support to determine if the node should be re-simulated after a new CEX is produced by the SAT solver. It is shown below that a circuit-based SAT solver typically returns CEXes in terms of a

small subset of the window inputs. Therefore, the simulator can use the support to skip re-simulating parts of the AIG not affected by the new CEX.

3.4 Equivalence class manager

Nodes of the logic network can be divided into disjoint *equivalence classes* as follows: two nodes belong to the same class if their Boolean functions are equivalent up to a complement. A node with a unique Boolean function belongs to a trivial equivalence class composed of the node itself. In practical applications, such as SAT sweeping and computing structural choices, only non-trivial equivalence classes, composed of two or more nodes, are considered. Note that internal AIG nodes whose Boolean function is a constant, belong to an equivalence class of the constant 0.

Equivalence classes are initialized by putting all nodes into the candidate equivalence class of the constant 0 node. Then, candidate equivalences are gradually refined by simulation and SAT solving, resulting in a set of proved equivalences.

Ideally, the equivalence class representation should be low-memory, fast to refine, and easy to handle when checking a node's equivalence class.

In the proposed manager, an AIG containing N nodes uses two integer arrays of size N to represent equivalence classes. For a node, the first array stores the ID of the *representative* node, which is defined as the first node in a topological order belonging to the class. The second array stores the ID of the next node of the class, in the topological order. If the node has a unique Boolean function (that is, belongs to a trivial class), its representative is the node itself and the next node is not defined.

The proposed equivalence class representation avoids memory fragmentation because it allocates only two arrays for the given AIG. It is convenient to update during incremental SAT solving. For example, after re-simulating a new CEX produced by the SAT solver, the nodes whose simulation information was updated, are collected, and all the affected equivalence classes are checked for a possible refinement. Timely refinement of equivalence classes after incremental simulation guarantees that nodes, no longer belonging to a class, are not checked by the SAT solver for equivalence with the class representative.

3.5 Simulation manager

The simulator used in this project is an incremental bit-parallel simulator working on the nodes of a logic window. When a new window is created, no simulation is performed. Only when a node is added to the window, does the simulation manager assign it a memory buffer to store its simulation information. This buffer is recycled when the logic window moves away from the node.

The simulation manager initializes simulation information of the PIs using random simulation, and gradually supplements these when new simulation patterns are produced by the SAT solver.

When a new CEX is derived, the simulation manager incrementally re-simulates the TFO cone of the PIs whose values have changed, using the information provided by the support manager. When a new node is added to the simulation window, only the node itself has to be simulated because its fanins are already in the window and their simulation information is up to date.

When the simulation manager re-starts, it is not known how many simulation patterns will be needed. This is why the manager allocates a fixed amount of memory for each node (say, memory enough to store 256 simulation patterns). When the number of CEXes exceeds the allocated storage, the pattern counter rolls over, and the oldest simulation patterns are over-written. This does not significantly reduce the simulation efficiency because older patterns play a lesser role in the ongoing refinement of the equivalence classes.

The simulation information is transferred between different windows by recording simulation patterns in terms of PIs. When a new simulation window is created, the saved PI patterns are used to populate the starting simulation patterns at the PIs of the new window.

The window-based simulation manager is fast because (1) it only looks at a small fraction of the design's logic, (2) it performs fast incremental simulation only for those nodes whose values have changed, and (3) it shares simulation information across the windows by recording all computed CEXes in terms of primary inputs.

3.6 Circuit-based SAT solver

To fit in with the above ideas, a new window-aware MiniSAT-like circuit-based solver, CBS, was developed to solve numerous relatively easy incremental SAT problems, each of which involves only a small fraction of the design's logic structure. For example, if equivalence $A = B$ has to be proved, the TFI logic cones of A and B are added to the solving window. CBS is window-aware because it works incrementally, dynamically resizing the internal data-structures to include only the nodes needed in the process of incremental SAT solving.

The key difference with MiniSAT is that CBS does not convert the underlying logic into CNF on-the-fly, as do most of the SAT sweepers available in ABC, for example, [10]. It performs an on-the-fly construction of the fanout representation for the logic cone that is being solved, which is faster, compared to CNF construction.

CBS inherits MiniSAT's two-literal watching for learned clauses, conflict analysis, clause learning, and non-chronological backtracking. CBS differs in that it (1) propagates problem constraints directly on the circuit, (2) uses J-frontier to make decisions, (3) does not remove learned clauses. The latter is because CBS is typically applied to a relatively small logic window and a typical problem is solved in 10-100 conflicts. For such relatively easy problems, few learned clauses are created, so there is little need for learned clause removal. Clauses are kept while a window is in use and deleted when the window is recycled.

The reasons why CBS is faster than MiniSAT are:

- construction of CNF is not needed
- J-frontier is used to detect satisfiable calls early
- the scope of solving is dynamically adjusted.

3.7 Interaction of the ecosystem components

This section describes the interactions between the components of the ecosystem illustrated in Figure 1.

The SAT sweeping manager uses the equivalence class data-structure to go through the candidate equivalences in a topological order. First, each candidate is checked with the class representative for equivalence, by adding it to the simulation window and simulating the already available CEXes. If the nodes differ, the equivalence is disproved, and the equivalence classes are incrementally refined. If the nodes cannot be proved equivalent, they are added to the SAT solving window, and the solver tries to prove them. If the nodes are proved, the candidate equivalence is labeled as a real equivalence. If the nodes are disproved, the CEX is produced by the SAT solver, is incrementally re-simulated by the simulator, and the relevant equivalence classes are incrementally refined. If the SAT solver fails to prove or disprove the nodes, they are assumed to be functionally different, and the equivalence class is broken into two classes without re-simulation.

The fanout manager incrementally adds fanouts when new nodes are added to the SAT solving window. Similarly, the support manager incrementally adds support information when new nodes are added to the simulation window. Both the fanout manager and the support manager are re-started when the window exceeds a predefined limit on its size. During window recycling, memory used to store fanouts and supports of the old window is reused for the next window.

3.8 Comparison with previous work

Using CEXes produced by the SAT solver to assist the simulator in disproving candidate equivalences is not new [6][10]. However, in previous work, eager incremental re-simulation of each new CEX was not affordable, which motivated lazy re-simulation using CEX batching. The batching works by accumulating a fixed number of CEXes (typically, 32 or 64), before the simulator is called to re-simulate all in one sweep over the circuit, resulting in refinement of the affected candidate equivalences.

Although batching saves time, it delays the positive effect of re-simulation. The refinement of the candidate classes produced as a result of re-simulation is delayed until the new batch is available. This delay makes it necessary for the SAT solver to come up with new CEXes, which would not be needed, had the CEXes been re-simulated more eagerly.

The eager re-simulation of the new CEXes in the proposed framework has become possible due to the combination of: (1) using the window-based approach to reduce the scope of simulation, (2) employing a circuit-based solver, which returns CEXes in terms of a subset rather than the complete set of the window PIs, and (3) relying on the support manager to limit incremental simulation to only those nodes in the TFO of the windows PIs whose values have changed when the new simulation pattern was added.

We observed that the set of window PIs whose TFO needs to be re-simulated is on average 50% smaller than the set of window PIs present in the CEX. This is because the simulation manager populates simulation information with random patterns for all PIs when the simulation manager is re-started. As a result, resimulation is done only for the TFO of those PIs (a) that are present in the CEX, and (b) whose simulation values in the CEX differ from the previously stored ones.

Another reason why this type of re-simulation is very fast. is because on average only 50% of the affected internal nodes change value when one CEX is re-simulated. This allows the simulation manager to skip having to access and updating their simulation information.

4. Experimental results

A complete framework is implemented in ABC [1] using all the proposed improvements. The goal is to speed up several key applications: (1) sequential synthesis (command *&scorr* [11]), (2) computing structural choices (command *&dch* [2]), (3) computing don't-cares for nodes in the network (*&mfs* [9]), (4) transferring net names from one netlist to another (command *dress*), etc.

At the time of submitting this paper for review, the complete framework is being finished. The circuit-based SAT solver CBS has been developed and integrated in command *&sat* used as a test bench to compare SAT solvers in ABC. The command is applicable to a combinational AIG, whose outputs represent a sequence of SAT problems to be solved. In particular, *&sat* without arguments runs MiniSAT [3] while *&sat -ca* runs CBS. A solver is applied to the POs of the AIG in their natural order. Re-simulation of CEXes remains to be enabled.

Preliminary experimental results compare the new solver against MiniSAT in applications (1) and (2) listed above. One challenging test-case from each application is considered and the sequence of incremental SAT calls is written in the circuit form into an AIGER file.

Table 1 lists the AIG statistics: the number of primary inputs, primary outputs (SAT calls), and internal AIG nodes.

Table 2 lists the number of SAT problems proved, disproved, and undecided, when each solver uses the resource limit of 10 conflicts, which is typical for these applications.

Table 3 compares the runtimes of the two solvers.

The results listed in Tables 2 and 3 allow us to conclude that, for the two challenging test cases considered, CBS almost always dominates MiniSAT in terms of quality, and always wins in terms of runtime.

We expect the results to be even better when the development of the framework is finished and fast eager re-simulation is enabled.

5. Conclusions

The paper analyzes the synergistic interaction between a simulator and a SAT solver used to solve a sequence of incremental SAT problems defined over a circuit. Several types of bookkeeping (e.g. the use of structural support of the nodes) are proposed to enhance the ecosystem. The design of a minimalistic AIG package is discussed as well as its benefits in the ecosystem.

Preliminary experiments have been performed in two application domains: sequential synthesis [11] and computing structural choices [2]. The new computations

have been found faster without compromising quality of results, compared to a well-tuned integration of MiniSAT.

Future work may include:

- Finishing the framework integration.
- Combining the J-frontier-based and the activity-based variable ordering in the circuit-based solver.
- Extending the ecosystem to AIGs containing three-input gates, such as multiplexers and majorities.
- Utilizing the ecosystem to speed up other packages in ABC, such as don't-care-based optimization and property directed reachability.

6. REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *IEEE Trans. CAD*, Vol. 25(12), December 2006, pp. 2894-2903.

[3] N. Een and N. Sörensson, "An extensible SAT-solver", *Proc. SAT'03*, LNCS 2919, pp. 502-518.

[4] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver", *Proc. DAC'02*.

[5] A. Kuehlmann and M. Ganay, "On-the-fly compression of logical circuits", *Proc. IWLS'00*.

[6] 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), December 2002, pp. 1377-1394.

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

[8] F. Lu, M. K. Iyer, G. Parthasarathy, L.-C. Wang, K.-T. Cheng, and K.C. Chen. "An efficient sequential SAT solver with improved search strategies", *Proc. DATE'05*.

[9] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE Trans. CAD*, Vol. 25(5), May 2006, pp. 743-755.

[10] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843.

[11] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*, pp. 234-241.

[12] C.-A. Wu, T.-H. Lin, C.-C. Lee, and C.-Y. (Ric) Huang, "QuteSAT: A robust circuit-based SAT solver for complex circuit structure", *Proc. DATE'07*, pp. 1313-1318.

Table 1: Problem statistics.

Testcase	Primary inputs	Primary outputs	AND nodes
scorr	135476	98192	544369
dch	13601	84460	538014

Table 2: Comparing the quality of results produced by MiniSAT and the new circuit-based solver CBS.

Testcase	Unsatisfiable		Satisfiable		Undecided	
	MiniSAT	CBS	MiniSAT	CBS	MiniSAT	CBS
scorr	31346	31811	65249	65549	1597	832
dch	14053	13268	26660	39287	43747	31905

Table 3: Comparing the runtime, in seconds, of MiniSAT and the new circuit-based solver CBS.

Testcase	Unsatisfiable		Satisfiable		Undecided	
	MiniSAT	CBS	MiniSAT	CBS	MiniSAT	CBS
scorr	1.20	0.18	7.28	0.60	0.51	0.11
dch	1.38	0.11	23.71	4.66	60.23	9.03