Encoding Global Unobservability for Efficient Translation to SAT

Miroslav N. Velev
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev

Abstract. The paper studies the use of global unobservability constraints in a CNF translation of Boolean formulas, where the unobservability of logic blocks is encoded with CNF unobservability variables and the logic output values of the blocks with CNF logic variables. Each block's unobservability variable is restricted by local unobservability constraints, expressing conditions that the output value of the block will not propagate to the primary output, given values of inputs to nearby gates on the path to the primary output. Global unobservability constraints add conditions that a block is unobservable if all paths to the primary output pass through logic blocks that are unobservable. By introducing a cut of unobservability check-points at the inputs of the top gate in a Boolean formula, we can impose global unobservability constraints for every logic block. The results show that global unobservability constraints lead to small additional speedup if local unobservability is exploited, but make the SAT-time less dependent on values of parameters in the translation.

1 Introduction

CNF-based SAT-solvers face two main hurdles to further improvements. First, the operation-intensive *Boolean Constraint Propagation* (BCP), taking up to 90% of the SAT time [13], and generating many non-sequential memory accesses that are prone to cache misses; also, BCP requires data-dependent branches that are hard to predict, and so frequently incur the branch misprediction penalty—at least 19 cycles, and up to 125 instructions in the Intel Pentium 4 [6]. Second, many L2-cache misses for big formulas [28], resulting in expensive accesses to main memory; the L2-cache miss penalty is up to hundreds of cycles currently, and is increasing [6].

Conventional translation to CNF [19] captures only local structural information for logic gates, and so does not exploit the concept of *unobservability* [2][18] (related to the concept of dominators in testing [1])—that the output values of subcircuits may be *unobservable* at the primary output, i.e., do not influence its value, given the values of other signals, and so those subcircuits can be pruned from the solution space.

Gupta et al. [5] implemented a circuit-based SAT-solver that uses structural information to identify gates with unobservable outputs and remove those gates from the solution space. Similar optimizations were exploited by Safarpour et al. [17]. Other circuit-based SAT-solvers identify signals with equal or complemented values in order to prune the solution space [8][11][14], or use a hybrid representation of Boolean circuits [3][14]—gate-level for the circuit, and CNF for constraints and learnt clauses.

This paper makes four contributions: 1) a method to introduce global unobservability constraints in a CNF translation where the unobservability of logic blocks is encoded with variables; 2) a method to increase the number of logic blocks restricted by global unobservability constraints; 3) a correctness proof for this translation; and 4) experimental results on Boolean formulas from formal verification of processors. The presented translation is general and applicable to other classes of formulas.

2 Previous Translations to CNF

In the formal verification tool flow [23] used in the experiments, the final Boolean formula consists of AND, OR, NOT, and ITE ("if-then-else") gates. Hashing [20] ensures that: there are no duplicate gates; merges an AND having another AND as input into a single AND, and similarly for ORs; eliminates duplicate inputs; and replaces an AND/OR with a constant if the gate has inputs that are complements of each other.

2.1 Conventional Translation from Propositional Logic to CNF

By introducing a new CNF variable for the output of every logic gate, and imposing constraints that preserve the gate's function [19], we get a satisfiability-equivalent CNF formula. Both the size of the resulting CNF and the complexity of the translation are linear in the size of the original Boolean formula. Instead of explicitly translating the inverters, we can subsume them in their fanout gates [15], by replacing all instances of the variable for the inverter output with the negated variable for the inverter input, thus eliminating the output variable and the 2 clauses for each inverter.

2.2 Translation to CNF by Merging ITE-Trees and Other Gate Groups

In the formal verification tool flow, we can apply an optimization [26] that produces Boolean formulas with many ITE-trees. An ITE-tree can be translated to CNF with a unified set of clauses [26], without intermediate variables for outputs of ITEs inside the tree. For example, $ITE(c_1, ITE(c_2, e_1, e_2), ITE(c_2, e_3, e_4))$, where c_1, c_2, e_1, ..., e_4 are Boolean variables, will be translated to CNF by introducing a new variable o only for the output of the tree, and using the clauses $(\neg e_1 \vee \neg c_1 \vee \neg c_2 \vee o) \wedge (e_1 \vee \neg c_1 \vee \neg c_2 \vee \neg o) \wedge (\neg e_2 \vee \neg c_1 \vee c_2 \vee o) \wedge (e_2 \vee \neg c_1 \vee c_2 \vee \neg o) \wedge (\neg e_3 \vee c_1 \vee \neg c_2 \vee o) \wedge (e_3 \vee c_1 \vee \neg c_2 \vee \neg o) \wedge (\neg e_4 \vee c_1 \vee c_2 \vee o) \wedge (e_4 \vee c_1 \vee c_2 \vee \neg o)$. That is, for every path from a non-controlling input of the tree, we introduce 2 clauses to express the conditions that if the input is *true* (*false*) and is selected to appear at the tree output by controlling inputs of ITEs, then the tree output should be *true* (*false*). See Fig. 1.a for another example.

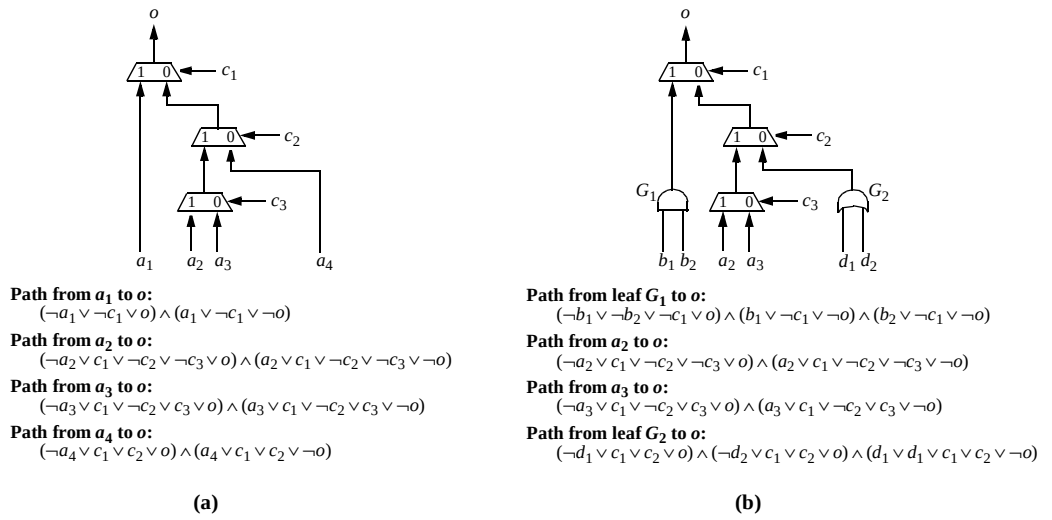


Fig. 1. (a) Example ITE-tree, and its translation to CNF with a unified set of clauses without intermediate variables for outputs of ITEs inside the tree; and (b) Merging an ITE-tree with 1 level of its AND/OR leaves that have fanout count of 1. Each ITE-tree is represented with the conjunction of all clauses for paths from the tree leaves to the tree output. ITEs are shown as multiplexors

ITE-trees can be further merged with one or more levels of their AND/OR leaves that have fanout count of 1 (see Fig. 1.b). We can also merge other gate groups [25], e.g., AND/OR→ITE (an ITE with an AND or OR as its then- or else-input, or both of these inputs), AND/ITE→OR, and OR/ITE→AND, but that results in minor additional improvements if ITE-trees are merged [26]. Since a gate may have many input gates with fanout count of 1, we can choose which input gate to merge by a variant of the FANIN heuristic [12] for BDD-variable ordering: selecting the input gate with highest topological level. The motivation is to shorten the longest path for BCP from a primary input to the output of the driven gate. Thus, if the heuristic is applied to many groups, we could shorten many paths for BCP from primary inputs to the primary output.

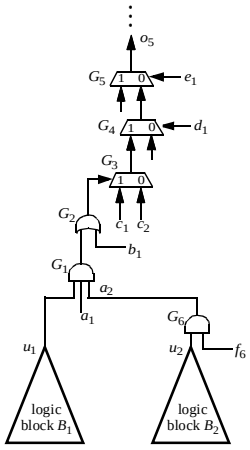
The benefits from merging ITE-trees are: fewer variables and clauses, i.e., reduced solution space, and so less BCP and fewer cache misses; use of signal unobservability—the 2 clauses for each path in an ITE-tree become satisfied as soon as an ITE-controlling signal selects another path; guiding the SAT-solver branching, making it easier for a SAT-solver to prune infeasible paths; and higher ranking of variables controlling ITEs at the top of ITE-trees, leading to better decisions and learning.

2.3 Reflecting the Local Unobservability of ITE-Trees

The *controlling value* of an AND (OR) gate, when applied at an input of the gate, will uniquely determine the value of the gate, regardless of the values of other inputs, i.e., the controlling value of an AND (OR) is 0 (1). We can account for the local unobservability of an ITE-tree by merging it with adjacent gates on the only path from the ITE-tree output to the primary output. Then, if one of those gates has a controlling value on an input that is not along this path, all clauses for the ITE-tree will get satisfied, thus allowing a conventional CNF-based SAT solver to exploit unobservability.

2.4 CNF Unobservability Variables and Constraints for Local Unobservability

An alternative way to encode the local unobservability of a logic block is to introduce a *CNF unobservability variable* [27], representing the conditions when the block's output is unobservable at the primary output—see Fig. 2. Such conditions depend on values of inputs to nearby gates situated on a fanout-free partial path from the block output toward the primary output. A CNF logic variable is still used to represent the logic value of the block output. The unobservability variable for a logic block is disjuncted to each clause for that block, so that when the unobservability variable is 1—meaning that the logic block *is not* observable at the primary output—all clauses for that block will be satisfied and a conventional CNF-based SAT-solver will have more freedom in assigning values to the variables in these clauses, possibly leaving some of those variables unassigned. A value of 0 assigned to a CNF unobservability variable means that the block's output value *may be* allowed to propagate to the end of the fanout-free partial path from the block output toward the primary output, and so may be observable at the primary output. In this paper, a CNF unobservability variable is introduced only for logic blocks with fanout count of 1.



Constraints for unobservability of block B_1 at o_5: — **Corresponding clauses:**

- **Due to gate G_1:** $\neg a_1 \Rightarrow u_1$ — $(a_1 \vee u_1)$
  $\neg a_2 \Rightarrow u_1$ — $(a_2 \vee u_1)$
- **Due to gate G_2:** $b_1 \Rightarrow u_1$ — $(\neg b_1 \vee u_1)$
- **Due to gate G_3:** no constraints added for ITEs where the path enters through the controlling input
- **Due to gate G_4:** $\neg d_1 \Rightarrow u_1$ — $(d_1 \vee u_1)$
- **Due to gate G_5:** $e_1 \Rightarrow u_1$ — $(\neg e_1 \vee u_1)$

Constraint for observability of block B_1 at o_5:

$a_1 \wedge a_2 \wedge \neg b_1 \wedge d_1 \wedge \neg e_1 \Rightarrow \neg u_1$ — $(\neg a_1 \vee \neg a_2 \vee b_1 \vee \neg d_1 \vee e_1 \vee \neg u_1)$

Constraints for unobservability of block B_2 at a_2: — **Corresponding clauses:**

- **Due to gate G_6:** $\neg f_6 \Rightarrow u_2$ — $(f_6 \vee u_2)$

Constraints for observability of block B_2 at a_2:

$f_6 \Rightarrow \neg u_2$ — $(\neg f_6 \vee \neg u_2)$

Fig. 2. Example blocks B_1 and B_2, and the constraints for their unobservability variables u_1 and u_2. When imposing constraints for a partial path, we can skip gates from that path, e.g., gate G_3 above, since those constraints encode the partial unobservability of the block output at the path output. Then, a value of 1 for the block's unobservability variable still means that the block is definitely unobservable at the path output, while a value of 0 means that the block may be observable at the path output. The partial paths, used in the constraints for the unobservability variables of different blocks, should not overlap in order to ensure that the new CNF formula will be satisfiability-equivalent with the CNF formula from conventional translation

3 Constraints for Global Unobservability of Logic Blocks

In order to more fully exploit the unobservability of a logic block, we can account for its global unobservability. An *unobservability check-point* is a signal that has an associated CNF unobservability variable. The *nearest cutset of unobservability check-points* for a block consists of the unobservability check-points that are each situated on a different path from the block output to the primary output, covering all such paths, such that each of these check-points is closest to the block compared to other unobservability check-points on the same path from the block output to the primary output.

A new constraint for unobservability of each block is added, in order to express the condition that if all of the nearest unobservability check-points are unobservable, then the block is unobservable, since each path from the block output to the primary output will go through one of those unobservability check-points. This is illustrated in Fig. 3 for the example block B_1 from Fig. 2. Assuming that there are only two paths from o_5 to the primary output, and that u_3 and u_4 are the nearest unobservability check-points on those paths, then the new unobservability constraint for block B_1 is $u_3 \wedge u_4 \Rightarrow u_1$.

We also need to extend the constraint for local observability of each block at the end of its fanout-free partial path toward the primary output, by accounting for the observability of each of the nearest unobservability check-points. If the block is observable at the end of its partial path toward the primary output, and one of the nearest unobservability check-points is observable, then the block is considered observable. In Fig. 3, the constraint for local observability of block B_1 at o_5 from Fig. 2, i.e., $a_1 \wedge a_2 \wedge \neg b_1 \wedge d_1 \wedge \neg e_1 \Rightarrow \neg u_1$, is extended for each of the two nearest unobservability check-points, u_3 and u_4, resulting in the observability constraints $\neg u_3 \wedge a_1 \wedge a_2 \wedge \neg b_1 \wedge d_1 \wedge \neg e_1 \Rightarrow \neg u_1$ and $\neg u_4 \wedge a_1 \wedge a_2 \wedge \neg b_1 \wedge d_1 \wedge \neg e_1 \Rightarrow \neg u_1$.

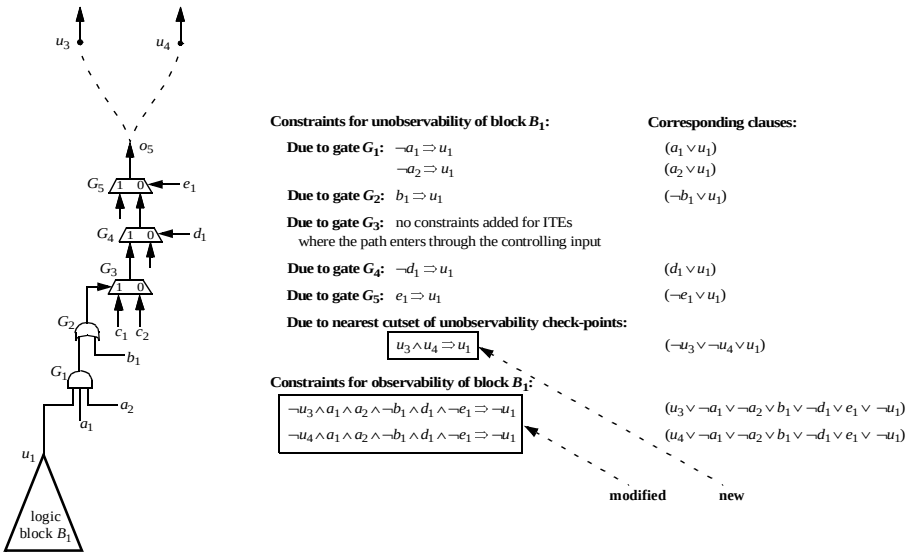


Fig. 3. Example block B_1 (from Fig. 2) and its constraints for unobservability. It is assumed that there are only two paths from o_5 to the primary output, such that u_3 and u_4 are the nearest unobservability check-points on the first and second path, respectively. That is, u_3 and u_4 form the nearest cutset of unobservability check-points. The new/modified constraints, resulting from global conditions, are shown in rectangles.

4 Global Unobservability Constraints for All Logic Blocks

By introducing a cut of unobservability check-points at the inputs of the top gate in a Boolean formula, we can impose global unobservability constraints for every logic block, since each path from

any logic block to the primary output will go through the inputs of the top gate in the formula. Without such a cut, most logic blocks have a path to the primary output without going through an unobservability check-point.

In Boolean correctness formulas from formal verification of microprocessors, the top gate is an OR, and so only this case is discussed here. Let the top OR gate have n inputs, and let each input i (for $i = 1, ..., n$) have a logic variable l_i and unobservability variable u_i. Then, an input is unobservable (its unobservability variable is 1) if a logic variable for one of the other inputs has a value of 1. An input is observable (its unobservability variable is 0) if the logic variables for all other inputs are 0. We also need to impose the constraint that at least one of the inputs is observable (this is a property of AND and OR gates).

5 Correctness Proof of the New Translation to CNF

THEOREM 1. *The new CNF formula, obtained after introducing CNF unobservability variables and corresponding constraints for local and global unobservability, is satisfiable iff the original CNF formula with only CNF logic variables is satisfiable.*

Proof: If. The satisfiability of the original CNF formula implies the satisfiability of the new CNF formula, obtained after introducing CNF unobservability variables and corresponding constraints. A satisfying assignment for the original CNF formula with only logic variables results in values for all logic variables. Since the value of each unobservability variable depends on logic variables from the local partial path to the primary output, and possibly on values of other unobservability variables for signals at higher points, and the implications for each unobservability variable are fully specified, then a unique value will follow for each unobservability variable, thus satisfying all clauses from unobservability or observability constraints for that variable. Thus, it is possible to assign values to the unobservability variables in order to satisfy the new CNF formula, given that the original CNF formula is satisfied.

And only if. The satisfiability of the new CNF formula, obtained after introducing unobservability variables and constraints, implies the satisfiability of the original CNF formula with only CNF logic variables. If we remove the clauses for block unobservability and observability constraints from the satisfied new CNF formula with unobservability variables, then we would get a satisfied CNF formula that is a variant of the original CNF formula, such that the unobservability variable for each block appears in every clause for that block. If a block's unobservability variable is 0, then removing that variable from the block's clauses will not affect their satisfiability. That would also be the case if a block's unobservability variable is 1 and the block's output has value that is consistent with the values of the block's inputs, given the function of the block, since then the original clauses for the block will be satisfied. However, if the block's unobservability variable is 1 and the block's output has value that is not consistent with the values of the block's inputs, then we need to prove that we can flip the value of the block's output—thus making it logically consistent with the values of the block's inputs—while keeping the satisfiability of all clauses. This is possible since the new output value can be propagated along all paths from the block output toward the primary output—without reaching the primary output. For each path toward the primary output, we can flip the output values of gates until reaching the input of a gate where another input has a controlling value that had made the block's unobservability variable to be 1 (either directly, through the local unobservability constraints, or indirectly, by implying a value of 1 for another unobservability variable that then implied a value of 1 for the considered unobservability variable, possibly through a chain of implications involving other unobservability variables). Hence, if the new CNF formula—containing unobservability variables and corresponding constraints—is satisfied, then it is possible to derive a satisfying assignment for the original CNF formula with only logic variables. □

6 Discussion

The CNF unobservability variable for a logic block appears in all the clauses for that block, and in clauses that represent the unobservability and observability constraints for this variable. Then, SAT decision heuristics that favor the most frequent variables will make a decision for the value of a block's unobservability variable before making a decision for the value of the block's output logic variable, since the output logic variable will also appear in all clauses for the block, but in at most two other clauses, since each considered block has a fanout count of 1. That is, SAT-solvers with such heuristics will first make decisions about the unobservability/observability of logic blocks, by assigning values to their unobservability variables, and then will try to justify these decisions with assignments to other variables. Hence, the unobservability variables introduce a higher-level of abstraction in the decisions of a conventional CNF-based SAT-solver, where some of the decisions will represent choices about the unobservability or observability of logic blocks. Furthermore, the SAT-solver will learn clauses to prevent infeasible observability/unobservability configurations from repeating.

Previous approaches for exploiting signal unobservability [5][17] required extensions to CNF-based SAT-solvers in order to dynamically mark logic gates as unobservable, based on the current variable assignments. Furthermore, these previous approaches used gate-level representations of circuits. In contrast, the presented CNF translation allows us to directly use existing CNF-based SAT-solvers, as well as to exploit the block structure of circuits.

In the experiments, unobservability variables were introduced only for ITE-trees with fanout count of 1. However, for small ITE-trees that are represented with a few clauses, the overhead of added unobservability variables and clauses for unobservability and observability constraints usually slows down the SAT-solving. To avoid such effects, two variations of the new translation were also explored: 1) a partial method where the new translation to CNF was applied only to ITE-trees that require more clauses than a given threshold; and 2) a hybrid scheme, where the unobservability of the big ITE-trees, requiring more clauses than a given threshold, was encoded with the new translation, and that of the smaller ITE-trees with the translation from Sect. 2.3.

7 Results

The Boolean formulas used in the experiments are from formal verification of safety of the benchmarks: 1dlx_iq50, a single-issue pipelined DLX [6], modeled as in [20], and extended with a 50-entry instruction queue between the instruction memory and the execution pipeline; 9vliw_iq2, a 9-wide VLIW processor with predicated execution, register remapping, advanced loads (see [22]), and a 2-entry instruction queue; and 9vliw_iq6, a variant a 6-entry instruction queue. The Boolean formulas were generated, and then translated to CNF with a tool flow [23] that was used at Motorola [9] to formally verify a model of the M•CORE processor, and detected bugs. The SAT-solver Siege_v4 [16]—one of the top performers in the SAT'03 competition [10]—was used for the experiments. The computer was a Dell OptiPlex GX260 with a 3.06-GHz Intel Pentium 4, having a 512-KB on-chip L2-cache, 2 GB of memory, and running Red Hat Linux 9.0.

Compared were five translation strategies: the old best strategy, which had the best performance in [26] and had resulted in up to 420× speedup relative to conventional CNF translation—merging ITE-trees, AND/OR→ITE groups, as well as ITE→AND and ITE→OR groups (if several inputs are ITEs with fanout count of 1, the ITE with highest topological level is chosen, based on a variant of the FANIN heuristic [12]); (1) a partial method where the new translation to CNF was applied only to ITE-trees that require more clauses than a given threshold, and only constraints for local unobservability were used; (2) a hybrid scheme, where the unobservability of the big ITE-trees, requiring more clauses than a given threshold, was encoded with the new translation by using only constraints for local unobservability, and the unobservability of the smaller ITE-trees was accounted for with the translation from Sect. 2.3; (1g) and (2g), extensions of Strategies (1) and (2),

respectively, with global constraints.

Table 1. Results from unsatisfiable formulas. Speedup is the ratio between the SAT time with the old best translation to CNF, and the new SAT time.

Boolean Formula (Boolean Variables)	Strategy for Translation to CNF	ITE-trees			SAT-solver Siege_v4			Speedup
		#	Avg. Depth	Max. Depth	Decisions $\times 10^6$	Conflicts $\times 10^6$	Time [sec]	
1dlx_iq50 (3,819)	(old best): merge ITE-trees, etc.	24,723	14.63	101	18.37	0.31	1,134	___
	(1)	24,723	14.63	101	17.29	0.25	899	**1.26**×
	(2)	24,723	14.63	101	17.63	0.27	1,074	1.06×
	(1g)	24,723	14.63	101	16.42	0.24	905	1.25×
	(2g)	24,723	14.63	101	17.33	0.28	930	1.22×
9vliw_iq2 (6,480)	(old best): merge ITE-trees, etc.	3,558	4.71	39	6.01	0.40	184	___
	(1)	3,558	4.71	39	6.37	0.42	146	1.26×
	(2)	3,558	4.71	39	5.26	0.32	107	1.72×
	(1g)	3,558	4.71	39	5.51	0.37	115	1.60×
	(2g)	3,558	4.71	39	5.15	0.28	99	**1.86**×
9vliw_iq6 (21,330)	(old best): merge ITE-trees, etc.	17,225	5.62	71	88.38	1.50	2,564	___
	(1)	17,225	5.62	71	71.56	1.07	1,608	1.59×
	(2)	17,225	5.62	71	64.02	1.11	1,389	1.85×
	(1g)	17,225	5.62	71	70.46	1.12	1,541	1.66×
	(2g)	17,225	5.62	71	64.21	0.97	1,253	**2.05**×

Table 1 summarizes the results for each benchmark. Strategy (2g) had the best performance on 2 of the 3 benchmarks, and resulted in speedup of 2.05× for the most complex benchmark, 9vliw_iq6, relative to the old best strategy that was already a very efficient translation to CNF. For 9vliw_iq6, Strategy (2g) was applied only to ITE-trees with more than 40 clauses (the value of the threshold).

8 Conclusions

The paper studied the use of global unobservability constraints in a CNF translation of Boolean formulas, where the unobservability of logic blocks is encoded with CNF unobservability variables and the logic output values of the blocks with CNF logic variables. An unobservability variable is restricted to be 1, if the output value of the block will not propagate to the primary output, given assignments to inputs of nearby gates on a partial path from the block output to the primary output. Global unobservability constraints add conditions that a block is unobservable if all paths to the primary output pass through logic blocks that are unobservable. By introducing a cut of unobservability points at the inputs of the top logic gate in a Boolean formula, we can impose global unobservability constraints for every logic block. The experimental results showed that global unobservability constraints lead to small additional speedup if local unobservability is exploited, but make the SAT-time less dependent on values of parameters in the translation. Future work will fine-tune this translation to CNF.

References

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, New York, 1990.

[2] M. Damiani, and G. De Micheli, "Observability Don't Care Sets and Boolean Relations," *International Conference on Computer-Aided Design (ICCAD '90)*, November 1990.

[3] M.K. Ganai, L. Zhang, P. Ashar, A. Gupta, S. Malik, "Combining Strengths of Circuit-Based and CNF-Based Algorithms for a High-Performance SAT Solver," *39th Design Automation Conference (DAC '02)*, June 2002.

[4] E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver," *Design, Automation, and Test in Europe (DATE '02)*, 2002.

[5] A. Gupta, A. Gupta, Z. Yang, and P. Ashar, "Dynamic Detection and Removal of Inactive Clauses in SAT with Application in Image Computation," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 536–541.

[6] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, San Francisco, 2002.

[7] D.S. Johnson, and M.A. Trick, *eds., The Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. http://dimacs.rutgers.edu/challenges
[8] A. Kuehlmann, M.K. Ganai, and V. Paruthi, "Circuit-Based Boolean Reasoning," *38th Design Automation Conference (DAC '01)*, 2001.
[9] S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE™ Microprocessor Core," *International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001.
[10] D. Le Berre, and L. Simon, "Results from the SAT'03 Solver Competition," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, 2003. http://www.lri.fr/~simon/contest03/results/
[11] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos, and Z. Hanna, "A Signal Correlation Guided ATPG Solver and Its Applications for Solving Difficult Industrial Cases," *40th Design Automation Conference (DAC '03)*, June 2003.
[12] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovani-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *International Conference on Computer-Aided Design (ICCAD '88)*, November 1988.
[13] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *38th Design Automation Conference (DAC '01)*, June 2001.
[14] R. Ostrowski, E. Grégoire, B. Mazure, and L. Saïs, "Recovering and Exploiting Structural Knowledge from CNF Formulas," *Principles and Practice of Constraint Programming (CP '02)*, P. Van Hentenryck, *ed.*, LNCS 2470, Springer-Verlag, September 2002.
[15] D.A. Plaisted, and S. Greenbaum, "A Structure Preserving Clause Form Translation," Journal of Symbolic Computation (JSC), Vol. 2, 1985, pp. 293–304.
[16] L. Ryan, Siege SAT Solver v.4. http://www.cs.sfu.ca/~loryan/personal/
[17] S. Safarpour, A. Veneris, R. Drechsler, and J. Lee, "Managing Don't Cares in Boolean Satisfiability," *Design, Automation and Test in Europe (DATE '04)*, February 2004.
[18] H. Savoj, and R.K. Brayton, "The Use of Abservability and External Don't Cares for the Simplification of Multi-Level Networks," *Design Automation Conference (DAC '00)*, June 1990.
[19] G.S. Tseitin, "On the Complexity of Derivation in Propositional Calculus," in *Studies in Constructive Mathematics and Mathematical Logic*, Part 2, 1968, pp. 115–125.
[20] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," *Correct Hardware Design and Verification Methods (CHARME '99)*, LNCS 1703, 1999.
[21] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *37th Design Automation Conference (DAC '00)*, June 2000, pp. 112–117.
[22] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution," *Computer-Aided Verification (CAV '00)*, LNCS 1855, Springer-Verlag, July 2000.
[23] M.N. Velev, and R.E. Bryant, "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations," *Computer-Aided Verification (CAV '01)*, LNCS 2102, Springer-Verlag, 2001.
[24] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2 (February 2003), pp. 73–106.
[25] M.N. Velev, "Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors," *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January 2004.
[26] M.N. Velev, "Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors," *Design, Automation and Test in Europe (DATE '04)*, February 2004.
[27] M.N. Velev, submitted for publication.
[28] L. Zhang, and S. Malik, "Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms," *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, May 2003.