Integrating Observability Don't Cares in All-Solution SAT Solvers

Sean Safarpour
Dept. Elec. & Comp. Eng.
University of Toronto
Toronto, Canada
sean@eecg.toronto.edu

Andreas Veneris
Dept. Elec. & Comp. Eng.
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

Rolf Drechsler
Dept. Comp. Sci.
Bremen University
Bremen, Germany
drechsle@informatik.uni-bremen.de

***Abstract*— All-solution Boolean satisfiability (SAT) solvers are engines employed to find all the possible solutions to a SAT problem. Their applications are found throughout the EDA industry in fields such as formal verification, circuit synthesis and automatic test pattern generation. Typically, these engines iteratively find each solution by calling a standard SAT solving procedure. Each solution is minimized using different post-processing techniques and the problem is constrained to prevent recurring solutions. In this work, instead of applying post-processing techniques, the objective is to minimize the size of the solution "on the fly" during the all-solution SAT solving process. This is achieved by allowing the solver to exploit the structural circuit Observability Don't Cares (ODC) arising from the problem. The solver makes decisions such that the number of ODCs is maximized in each solution thus leading to an overall smaller number of iterations. Through extensive experiments, it is demonstrated that integrating ODC techniques within an all-solution SAT solver results in increased performance and more compact solutions.**

I. Introduction

Due to recent developments, Boolean satisfiability (SAT) solvers are now employed in many EDA applications such as formal verification [1], circuit synthesis [2], and circuit testing [3]. The success of SAT solvers has led to the increased popularity of a subtle derivative, the *all-solution SAT solver*. Where the SAT solver seeks to find a single solution to a SAT problem, the all-solution SAT solver seeks to find all solutions to a SAT problem. Applications of all-solution SAT solvers are found in Unbounded Model Checking (UMC) [4], Automated Test Pattern Generation (ATPG) [3], formal design debugging [5], and circuit optimization [6], [2], among others.

Typically, all-solution SAT solvers *iteratively* call a standard SAT solving procedure to find each solution to a problem. At each iteration, when the standard SAT solver returns a solution, a *blocking clause* [4] is added to the problem to prevent it from discovering the same solution in future iterations. Additionally, most all-solution SAT solvers attempt to "generalize" the solutions by applying post-processing logic minimization or reduction techniques [7], [8], [4]. In this manner, they convert sets of single solutions into a *solution cube* which "contains" a number of individual solutions. Since the number of solutions can be exponential to the problem size, "compacting" the solutions at each iteration is critical for the efficiency of the solver.

For circuit-based problems, most solution reduction techniques *implicitly* make use of the circuit's Observability Don't Care (ODC) space to reduce the size of each solution. Informally, ODCs are signal values that do not affect the outcome of the circuit under a set of signal assignments. It has been shown that managing ODC signals is an effective way of increasing the efficiency of SAT solvers for many circuit-based problems [9], [10]. In this paper, we argue that ODCs may also be exploited to return more compact solutions in an all-solution SAT solver thus improving its efficiency. For example, consider two solutions $\{a = 1, b = 0, c = 1\}$ and $\{a = 1, b = 0, c = 0\}$ to some problem. If the SAT solver can deduce that signal c is a don't care, the single solution cube $\{a = 1, b = 0\}$ provides the exact same information as the two previous solutions.

In this paper we develop techniques based on ODCs that enhance SAT solvers for problems where all the solutions are required. Unlike many existing techniques that reduce the size of the solution cubes in a *post-processing* manner, our approach modifies the internal SAT solver to explicitly consider don't cares dynamically during execution so that it *dynamically* reduces the size of each solution and improve performance.

This paper is structured as follows. We briefly discuss related work in Section II and provide background material in Section III. Section IV presents the benefits of ODCs for all-solution SAT solvers and introduces a novel scoring scheme used in the decision making procedure. Section V discusses the experiments, while Section VI concludes this work.

II. Related Work

Recently, there has been much work done on all-solution SAT solving frameworks mostly for UMC problems. Both [4] and [8] develop all-solution SAT frameworks which use post-processing implication graph analysis or justification procedures to reduce the size of the cubes. In [11], a reduction algorithm is developed to determine the "necessary" input assignments by performing a forward traversal of the circuit. In [7], the main contribution is the use of *cofactoring* to reduce the solutions cubes for UMC problems. Most of the work on all-solution SAT solvers has been concerned with reducing the size of the blocking clauses or the solution cubes after they are found. In contrast, our work is concerned with using a SAT solver that is explicitly aware of ODCs and makes decisions "on the fly" to reduce the number of solution cubes.

III. Preliminaries

In this work, the terms *line, signal*, and *variable* are used interchangeably. A signal with multiple fanouts is called a *stem* and each fanout is called a *branch*. We distinguish between a stem and each of its branches by allocating a different SAT variable for each. A signal is a *neighbor* of another signal iff they are fanins of the same gate. A *literal* refers to a Boolean variable or its complement. Over n variables, a *minterm* is a conjunction of exactly n literals corresponding to each variable. A *cube* is a conjunction of $m \leq n$ literals [12]. We say that a cube A *covers* another cube B, if for every minterm in B the same minterm exists in A [12]. The *cube size* refers to the number of literals it contains.

A. Observability Don't Cares

Since there are many variations of ODCs in the literature [12], [10], in this section we briefly describe ODCs as explained in [10]. Informally, given a combinational circuit where some lines are assigned Boolean logic values, a signal is an ODC if assigning it a

0 or 1 logic value does not change the value of any primary output. In a SAT problem derived from a circuit, we say that a variable is *marked lazy* iff its corresponding circuit signal is an ODC and is unassigned.

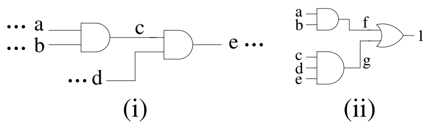

Fig. 1. Examples showing benefits of ODCs

As an example, consider Figure 1 (i) assuming that signal e is a primary output and all variables are unassigned. When line d is assigned a logic 0, the value on the output e remains the same regardless of the logic value on lines a, b, c. As such, variables a, b, c are declared ODCs and marked *lazy*.

B. All-solution SAT solvers

An all-solution SAT solver finds *all* the satisfiable variable assignments to a Boolean satisfiability problem. To distinguish between a SAT solver that finds a single solution and one that finds all the solutions to a particular problem, we refer to the former as a *standard* SAT solver. Figure 2 presents a typical algorithm for an all-solution SAT solver.

```
1: $A = \emptyset$
2: while (1) do
3:   while ($decide()$) do
4:     if ( $deduce() = conflict$) then
5:       $blevel = analyze\_conflict()$
6:       if ( $blevel = 0$) then
7:         if ($A = \emptyset$) then
8:           return UNSATISFIABLE
9:         else
10:          return DONE
11:        end if
12:      end if
13:      $backtrack(blevel)$
14:    end if
15:  end while
16:  // problem is SATISFIABLE
17:  $full\_sol = get\_assignments()$
18:  $reduced = reduce\_assignments(full\_sol)$
19:  $A = A \cup reduced$
20:  $add\_blocking\_clause(reduced)$
21:  $blevel = analyze\_conflict()$
22:  if ($blevel = 0$) then
23:    return DONE
24:  end if
25:  $backtrack(blevel)$
26: end while
```

Fig. 2. All-solution SAT solver

In Figure 2, the while-loop in lines 3–15 is a standard DPLL-based SAT algorithm [13]. Once a satisfying assignment (solution) is found, the `get_assignments()` and `add_blocking_clause()` procedures generate a *blocking clause* [4] in terms of *variables of interest* to prevent finding the same solution in subsequent iterations as well as to force the solver to backtrack and look for other solutions. Depending on the application, variables of interest may be primary inputs, state variables, or variables corresponding to cuts through the implication graph or the original problem circuit [7], [8], [4]. After each found solution, all-solution SAT solvers typically call a procedure such as `reduce_assignments()` to generate cubes to cover the found solutions [7], [8], [4].

IV. ODCs and All-Solution SAT

In an all-solution SAT solver, the size of each solution cube returned by the procedures `get_assignments()` and `reduce_assignments()` of the algorithm in Figure 2 is critical to the engine's overall efficiency. A solution cube containing fewer literals covers more solutions and reduces the number of iterations. In this regard, SAT solvers that exploit ODCs can have the inherent advantage of finding small solution cubes before applying the `reduce_assignments()` procedure [9], [10].

As an example consider the circuit in Figure 1 (ii) where the objective is to find all primary input assignments such that the output is 1. An all-solution SAT solver using observability don't cares can make the assignment $f = 1$ leading to signals c, d, e, g becoming ODCs and resulting in the solution $\{a = 1, b = 1\}$. Similarly, in the next iteration it can make the assignment $g = 1$ leading to signals a, b, f becoming ODCs and resulting in the solution $\{c = 1, d = 1, e = 1\}$. In contrast, an all-solution SAT solver without any assignment reduction procedures needs up to 11 iterations to find all the 11 solution minterms (8 minterms covered by the cube $\{a = 1, b = 1\}$ plus four minterms covered by the cube $\{c = 1, d = 1, e = 1\}$ minus the common minterm $\{a = 1, b = 1, c = 1, d = 1, e = 1\}$).

In [9], [10], ODCs were used to improve the performance of SAT solvers without considering their effect on the size of the satisfying assignments. Here, our objective is to use ODCs to achieve solution cubes with few literals and thus improve the performance of all-solution SAT solvers.

A. Variable Assignment Procedure

In [10] it was shown that the decision making procedure of DPLL-based SAT solvers can safely avoid deciding on ODC variables. For most circuit-based SAT problems, the problem constraints are constructed in such a way that the value of ODC variables is irrelevant. It can be proved that for common circuit-based SAT problems, ODC signals can be altogether ignored by the all-solution SAT solver's assignment procedure without affecting the outcome of the SAT problem. The complete proof can be found [14].

Since assignments lead to Boolean Constraint Propagation (BCP) and for most problems BCP can take over 80% of the overall run time [13], ignoring assignments on ODC or *lazy* variables can decrease the SAT solver's overhead and potentially increase its efficiency. In our ODC aware SAT solver, we ignore all lazy variables in the variable assignment procedure.

B. Decision Making Procedure

An all-solution SAT solver using ODCs can reduce the number of iterations needed to find all solutions and improve its run time by tuning its decision making heuristic in favor of small solution cubes. For example, consider the circuit partitioned in fanout free cones as illustrated in Figure 3. A SAT solver such as [10] decides on variables such that the largest cones become lazy first. In this respect, decisions are made such that cones C and D are marked lazy first. Assuming that primary inputs are the variables of interest, to generate small solutions cubes, decisions should be made to mark cones A and B as lazy first since they produce the most number of lazy primary inputs.

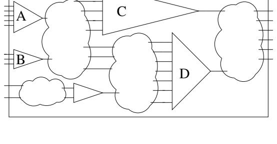

Fig. 3. Partition of circuit in fanout free cones

In the rest of this paper we consider primary inputs as the variables of interest, but the procedures can be easily modified for other variables of interest such as state variables or variables corresponding to cuts in the circuit. We develop a decision making procedure that branches on variables with the highest scores similar to VSIDS [13]. The score for each variable is calculated in a quick pre-processing phase that takes linear time with respect to the problem size. The novel scoring scheme is based on a heuristic that assesses a variable's ability to mark primary inputs lazy. The objective of each decision is to branch on the variable which has the highest probability of

```
1: //Calculate Pip value
2: for all (gates $g$ in a breadth-first manner) do
3:   if $g$ is primary input then
4:     if number of fanouts for $g > 19$ then
5:       $g$.Pip=1
6:     else
7:       $g$.Pip=100 − 5∗(number of fanouts-1)
8:     end if
9:   else
10:    for all (fanins $pred$) do
11:      $g$.Pip=$g$.Pip + $pred$.Pip
12:    end for
13:    if number of fanouts for $line > 3$ then
14:      $line$.Pip = $line$.Pip/4
15:    else
16:      $line$.Pip = $line$.Pip/(number of fanouts)
17:    end if
18:  end if
19:  //Initialize Lipi
20:  $g$.Lipi_0 = 0
21:  $g$.Lipi_1 = 0
22: end for
23: //Calculate Lipi value
24: for all (gates $g$ in a breadth-first manner) do
25:   for all (fanins $line$) do
26:     if ($g$ controlling value = 0) then
27:       for all (fanins $neighbor$) do
28:         $line$.Lipi_0 = $line$.Lipi_0+$neighbor$.Pip
29:       end for
30:     else
31:       for all (fanins $neighbor$) do
32:         $line$.Lipi_1 = $line$.Lipi_1+$neighbor$.Pip
33:       end for
34:     end if
35:     $line$.Lipi = max($line$.Lipi_0,$line$.Lipi_1)
36:   end for
37: end for
```

Fig. 4. Calculating Lipi

marking as many primary inputs lazy as possible. As such, when a satisfying assignment is found in terms of the variables of interest, many of the variables may be lazy and left unassigned leading to a small solution cube.

We call the variable score *Lazy Influence on Primary Inputs* or `Lipi` for short. The `Lipi` value for all variables is found in a pre-processing phase using two breadth-first traversals of the circuit. In the first pass, an intermediate value called *Primary Input Predecessor* or `Pip` is calculated for each variable. For each line, `Pip` is based on the number of primary inputs in its transitive fanin while taking into account its number of fanouts or branches. There are two `Lipi` scores for each line, one for each assignment phase (0 or 1). Each `Lipi` score for a line is calculated based on the number of other neighboring lines that get marked lazy when this line takes on a controlling value. More specifically, each `Lipi` score corresponds to the sum of the `Pip` scores of the neighboring fanins when the particular phase is a controlling value for the gate under consideration. Finally, the largest of the `Lipi` scores for each phase is selected and used in the branching procedure. The algorithm in Figure 4 illustrates how the `Pip` and `Lipi` variables are calculated. Lines 1-22 calculate the `Pip` value depending on the fanins and the fanout number, while lines 23-37 calculate the `Lipi` score based on the `Pip` value of the neighbors.

Figure 5 shows the `Pip` and `Lipi` values on a sample circuit. For

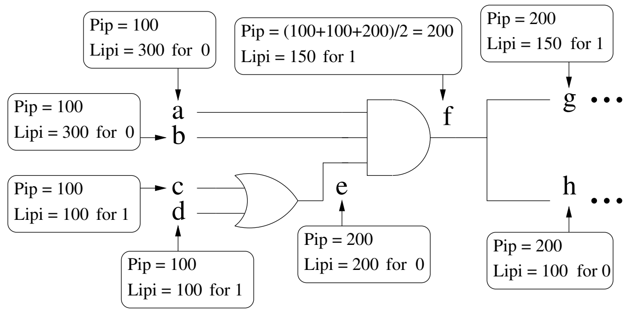

Fig. 5. Example of Pip and Lipi assignments

example, to calculate the `Pip` value for signal e, we add the `Pip` value of its fanins (i.e. 100 for input c + 100 for input d). The `Lipi` score for e is calculated by adding the `Pip` value of the neighboring lines for each phase (i.e. 100 for neighbor a + 100 for neighbor b).

Consider the circuit problem of Figure 5. Since the `Lipi` score for variable a (along with b) is the largest among all variable scores, the first decision made by the SAT solver is $a = 0$ ($b = 0$). The result of this decision is that the primary inputs b, c and d are marked lazy. If the problem is satisfied without backtracking on decision $a = 0$, the only non-lazy primary input variable is a leading to the solution cube $\{a = 0\}$ covering 8 distinct solutions/minterms.

V. Experiments

We develop our proposed ODC techniques on top of the SAT solver zChaff [13] and combine them with the solution reduction technique of [8]. We generate over 1000 problems using all of the ISCAS'89 benchmarks where current state and next state lines are replaced with primary inputs and primary outputs, respectively. In these problems, an arbitrary number of primary outputs are constrained to 0 or 1 at random and the objective is to find all of the primary input assignments that satisfy the constraints. These problems are similar to scenarios found in many problems such as pre-image computation [4] and circuit optimization[6], [2]. The experiments are run on a Sun Blade 1000 machine with a 750MHz CPU and 2.5GB of memory.

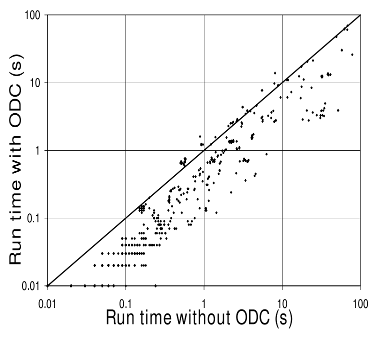

Fig. 6. Performance comparison with/without ODCs

We first demonstrate the benefits of using a SAT solver that accounts for ODCs within the all-solution framework. Figure 6 plots the run time of the SAT solver with and without ODCs against each other. Points below the diagonal line signify that our ODC techniques result in faster run times. Since the majority of points are below the diagonal line, we deduce that the ODC methods developed here are effective for all-solution SAT solvers.

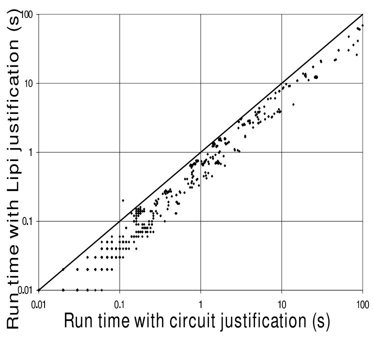

Fig. 7. Comparison of the justification procedure

Most all-solution SAT solvers perform a solution reduction technique after each iteration based on backward justification of the circuit or implication graph [8], [4]. Here, we propose using the

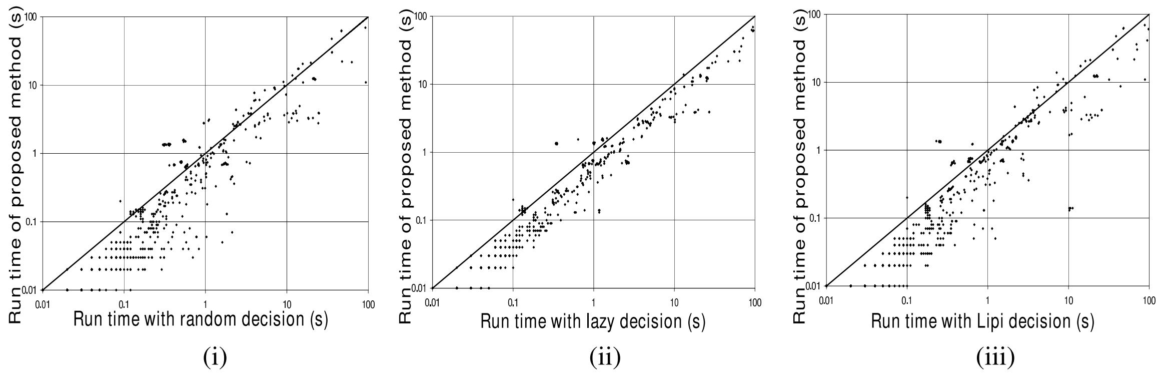

Fig. 8. Comparison of different decision making heuristics

computed `Lipi` scores during justification. During backward justification, if there is a choice on the circuit line to justify, the line with the greatest `Lipi` score is selected. Figure 7 compares the run time of the all-solution SAT solver using a justification procedure similar to [8] against our justification procedure based on the `Lipi` scores. This result shows that when the internal SAT engine accounts for ODCs, the `Lipi` score can be used during the justification procedure to further increase the efficiency.

Figure 8 illustrates the benefits of our decision making heuristic over three other methods. As discussed in Section IV-B, the `Lipi` scoring scheme is quite effective at making decisions that produce many lazy primary inputs, but to achieve a balance between solving each iteration quickly and finding small solution cubes, we employ a strategy similar to [10]. In this strategy the variable with the highest `Lipi` score is selected from a set of variables with the highest VSIDS scores. This decision making approach is found to be well suited for all-solution SAT problems. In Figure 8 (i) our scoring scheme is compared against a random scoring scheme to ensure its efficiency in the general case. Figure 8 (ii) compares our approach against the scoring scheme used in [10] which is successful in producing the maximal number of lazy variables at each decision. In Figure 8 (iii), we compare our scoring scheme (combined `Lipi` and VSIDS) against the pure `Lipi` scoring scheme to illustrate the trade-off between solving for solutions quickly and finding small solution cubes. In all cases, our decision making procedure and variable scoring scheme is found to be the most efficient.

Method name	# cubes	run time	wins	sole wins
without ODC	128	12.8	198	17
random score	110	11.2	165	11
[10] score	107	10.4	209	23
`Lipi` score	116	12.5	210	21
without Lipi just.	88	10.3	233	26
all ODC dev.	**83**	**9.1**	**279**	**60**

TABLE I

EVALUATION BASED ON SEVERAL CRITERION

In Table I we demonstrate the efficiency of our proposed techniques based on different criteria. Rows 2-6 represent the results of the different strategies we compare against. Row 2 shows the results for the all-solution SAT solver with the reduction technique from [8] without our ODC-based developments. Rows 3, 4 and 5 represent the results for the all-solution SAT solver using ODCs with a random scoring scheme, with the scoring scheme from [10], and with the pure `Lipi` scoring scheme, respectively. Row 6 represents the results for the all-solution SAT solver with the reduction technique from [8] including all our ODC developments. Finally, Row 7 presents the results incorporating *all of our ODC developments* including our `Lipi` justification procedure instead of the reduction technique from [8]. Column 2 and 3 show the average number of solution cubes and average run times in seconds for each method. For each experiment, we give a *win* to the approach requiring the minimum number of iterations. If only one approach is given a win for an experiment, we call this condition a *sole win*. The number of *wins* and *sole wins* can be used to rate the compactness of the solution cubes for each method. Column 4 and 5 show the number of *wins* and *sole wins*, respectively, for each approach.

The results of Table I show that our ODC developments lead to an all-solution SAT solver with faster run times and more compact solution cubes on average. Overall, the experimental results demonstrate that integrating ODCs in all-solution SAT solvers is beneficial for most problems.

VI. Conclusion

We demonstrated that integrating ODCs in all-solution SAT solvers can inherently lead to a faster engine which returns smaller solutions. Our novel variable scoring scheme is successful at biasing the decision making procedure in favor of minimal solution cubes. As demonstrated through many experiments, our developments result in an overall increase in the all-solution SAT solver's efficiency.

References

[1] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Design Automation Conf.*, 1999, pp. 317–320.

[2] S. Sapra, M. Theobald, and E. Clarke, "SAT-based algorithms for logic minimization," in *Int'l Conf. on Comp. Design*, 2003, pp. 510–518.

[3] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 11, pp. 4–15, 1992.

[4] K. McMillan, "Applying SAT methods in unbounded symbolic model checking." in *Computer Aided Verification*, 2002, pp. 250–264.

[5] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir, "Debugging sequential circuits using Boolean satisfiability," in *Int'l Conf. on CAD*, 2004, pp. 204–209.

[6] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *Design, Automation and Test in Europe*, 2005, pp. 418–423.

[7] M. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based symbolic unbounded model checking using circuit cofactoring," in *Int'l Conf. on CAD*, Nov. 2004, pp. 510–517.

[8] H.-J. Kang and I.-C. Park, "SAT-based unbounded symbolic model checking," *IEEE Trans. on CAD*, vol. 24, no. 2, pp. 129–140, 2005.

[9] A. Gupta, A. Gupta, Z. Yang, and P. Ashar, "Dynamic detection and removal of inactive clauses in SAT with application in image computation," in *Design Automation Conf.*, 2001, pp. 536–541.

[10] S. Safarpour, A. Veneris, R. Drechsler, and J. Hang, "Managing don't cares in Boolean satisfiability," in *Design, Automation and Test in Europe*, 2004, pp. 260–265.

[11] M. Iyer, G. Parthasarathy, and K. Cheng, "SATORI - a fast sequential SAT engine for circuits." in *Int'l Conf. on CAD*, 2003, pp. 320–325.

[12] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.

[14] S. Safarpour, "Managing circuit don't cares in Boolean satisfiability," Master's thesis, Department of Electrical and Computer Engineering, Univeristy of Toronto, 2005. [Online]. Available: http://www.eecg.toronto.edu/~sean/thesis.pdf