LPSAT: A Unified Approach to RTL Satisfiability

Zhihong Zeng, Priyank Kalla, Maciej Ciesielski
Department of Electrical and Computer Engineering
University of Massachusetts Amherst, MA-01003, USA
{zzeng, pkalla, ciesiel}@ecs.umass.edu

Abstract

LPSAT is an LP-based comprehensive infrastructure designed to solve the satisfiability (SAT) problem for complex RTL designs containing both word-level arithmetic operators and bit-level Boolean logic. The presented technique uses a mixed integer linear program to model the constraints corresponding to both domains of the design. Our technique renders the constraint propagation between the two domains implicit to the MILP solver, thus enhancing the overall efficiency of the SAT framework. The experimental results are quite promising when compared with generic CNF-based and BDD-based SAT algorithms.

I. Introduction

The Boolean satisfiability problem (SAT) has many direct applications in the electronic design automation (EDA) arena, including test pattern generation, timing analysis, logic verification, functional testing, etc. SAT belongs to the class of NP-complete problems, with algorithmic solutions having exponential worst-case complexity. This problem has been widely investigated, and continues to be so because efficient SAT techniques can greatly impact the performance of many EDA tools.

With respect to applications in VLSI CAD, most instances of SAT formulations start from an abstract circuit description, for which a required output value needs to be validated. The resulting formulation is then mapped onto an instance of SAT, typically using Conjunctive Normal Form (CNF) formulae. Classical approaches to SAT are based on variations of the well known Davis and Putnam procedure [1]. Typical versions of this procedure incorporate a chronological backtrack-based search [2] [3] [4] [5]; at each node in the search tree, it selects an assignment and prunes the subsequent search by iterative application of the unit clause and pure literal rules [6]. Recent approaches incorporate learning techniques and other conflict analysis procedures with *non-chronological* backtracks to prune the search space [7] [8].

Another popular approach to solving the Boolean satisfiability problems is based on Binary Decision Diagrams (BDDs) [9], [10]. Given a circuit for which a SAT instance needs to be solved, a set of BDDs can be constructed representing the output value constraints. The conjunction of all the constraints expressed as a Boolean product of the corresponding BDDs (referred to as a *product BDD*) represents the set of *all* satisfying solutions. Any element of the resulting constraint set, *i.e.* any path from the root to the terminal vertex 1 in the product BDD, gives a feasible SAT solution [11][12][13]. However, a major limitation of this approach is the memory explosion problem associated with the construction of the product BDD. Recently, Kalla *et al.* [14] proposed a BDD-based SAT technique that overcomes the problems related to BDD size by exploiting elements of the *unate recursive paradigm*. This technique searches for SAT solutions in the *cofactors* of the individual constraint BDDs, thus restricting the growth of the entire BDD search space.

Let us briefly discuss the SAT problem as it appears in the context of our work and analyze the limitations of contemporary SAT approaches. Consider the circuit shown in Fig. 1 which represents a block diagram of a Register-Transfer Level (RTL) design. It contains instances of arithmetic blocks (such as adders, multipliers and comparators) as well as Boolean logic. Given a set of value requirements at the outputs, how do we find the input assignments that satisfy these output requirements?

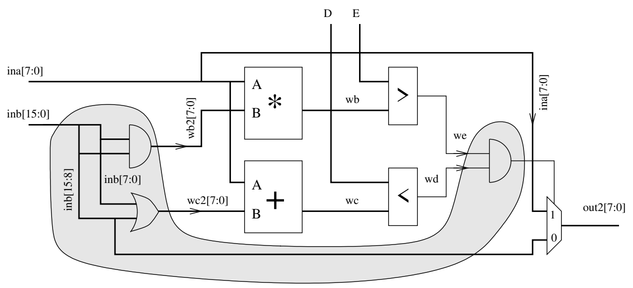

Fig. 1. An example circuit

CNF-based SAT solvers can be directly applied to the above circuit by transforming the entire circuit into CNF formulas. However, since practical gate-level circuits can be quite large, dealing with substantially large CNF formulas results in unacceptable CPU runtime. At the same time, BDD-based techniques suffer from size explosion problems. It is often the case that the product BDD for a large circuit cannot be constructed because of its prohibitive size - wide multipliers are a case in point.

To overcome these drawbacks, Fallah *et al.* [15] proposed a *hybrid satisfiability* approach, HSAT, to generate functional test vectors for HDL designs. Working on the RTL descriptions, the hybrid method generates *CNF clauses* for random Boolean logic, and *linear arithmetic constraints* for arithmetic blocks in the design. Then, a 3-SAT solver is applied to solve SAT for CNF clauses for Boolean logic, while *linear programming* (LP) techniques are used to check the *feasibility* of linear constraints for arithmetic blocks. A point to note here is that 3-SAT checking for Boolean logic and LP techniques for arithmetic blocks are performed *separately*, in different domains. That is, for variables that correspond to the *interface* between the Boolean and arithmetic domains of the design, an assignment is selected from the CNF-clauses, and the resulting constraints are propagated to the arithmetic domain for the linear program to check for consistency. In such a framework, backtracks between the two engines are inevitable and performed explicitly. If the selected binary elements from the Boolean domain (variable assignments that satisfy the CNF clauses) cause the LP constraints in the arithmetic domain to be infeasible, backtracking is needed to select another set of Boolean assignments. Since these two engines operate in separate domains, the performance of HSAT is limited by the heuristics that choose the set of assignments to Boolean variables.

Constraint propagation techniques between different domains have been explored to generate functional tests and high-level ATPG vectors on HDL descriptions [15] [16] [17]. However, it would be desirable to use an infrastructure that can represent *both* Boolean as well as arithmetic constraints in a single unified domain. By doing so, constraint propagation between the arithmetic and Boolean parts can be handled implicitly and efficiently, without any loss of information.

This paper presents a new LP-based SAT engine, called *LP-SAT*. It provides a comprehensive approach to solving satisfiability problems for RTL designs containing instances of both arithmetic blocks and Boolean logic. *Linear arithmetic constraints* are used to model the arithmetic blocks of the design, as in [15] The SAT constraints for the Boolean logic part, however, are represented using an *integer-linear program* (ILP), in contrast to the CNF formulas used in [15]. For this purpose, we map the Boolean logic into a network of AND-OR-NOT gates, and represent the satisfiability constraints by a system of integer/linear equations. The constraints representing both the Boolean and arithmetic parts are combined together to form a *mixed integer linear program* (MILP). An MILP solver is then used to solve the satisfiability problem for the entire design.

By representing the satisfiability constraints for both domains in a single MILP domain, we obviate the drawbacks associated with the use of two different SAT engines operating in isolation. Furthermore, our technique renders constraint propagation between the arithmetic and Boolean domains *implicit* to the MILP solver. This, in turn, helps avoid the explicit backtracking between different SAT engines. This seamless unification of SAT constraints for arithmetic and Boolean parts of the design blocks into a single unified MILP domain, allows to solve SAT problem quickly and efficiently for complex digital designs as verified by extensive experimentation.

II. Partitioning into Arithmetic and Boolean Parts

The criterion for partitioning the circuit into arithmetic and Boolean parts is dictated by the general structure of the circuit. Such a partitioning is not unique and different partitioning schemes may result in different CPU time of the resulting MILP program. The general rule is to put as much as possible into arithmetic portion, and avoid expanding the word-level signals into bits, unless necessary. A notable exception is "bit nibbling" which requires extraction of a portion of the word as a bit vector to interfaces with the Boolean logic.

In the example in Figure 1, the adder, the multiplier, the mux, and the two comparators belong to the arithmetic domain; the shaded part, containing logic gates, belongs to the Boolean part. In principle, the multiplexor can be modeled in either form. However, since one of the multiplexor's inputs $ina[7:0]$ is also an input to other arithmetic blocks, it will be treated as an arithmetic operator here.

The signals in the partitioned circuit can be classified into the following three types:

- *Word-level signals.* These signals are declared as integer variables in the MILP problem. In Figure 1, $ina[7:0]$, wb and wc are integer variables. (It should be noted that, depending on their interaction with other signals, some of those variables can be declared as *continuous* with integer bounds; they will assume integer values automatically).
- *World-level signals with bit-level expansion.* Signals of this type typically exist at the interface of arithmetic and Boolean parts. Depending on the type of interface, the expansion may be *full* or *partial*. If the entire word interfaces with the Boolean logic, it must be *fully expanded* into individual bits, each bit being declared as a binary variable. For example, signal $wc2$ in Figure 1, interfacing with the adder, will be fully expanded as follows:

$$wc2 = wc2[0] + 2^1 * wc2[1] + \cdots + 2^7 * wc2[7],$$

were $wc2[i]$, i=1,...,7, are binary variables, and $wc2$ is an implicit integer variable.
Partial expansion is also possible in case of *bit nibbling*. In this case, a word is broken into several parts in the places defined by the nibbling bits. Each portion of the word can still be expressed as an integer, with bits being declared as binary variables.
- *Single-bit signals.* These are typically control or decision signals, and as such they are declared as *binary* variables. For example, *we* and *wd* in Figure 1 are binary decision variables. It should be noted that their nature is different than the Boolean variables described above, and typically they are the ones responsible for high computational complexity of the MILP program.

In our system the circuit partitioning into arithmetic and Boolean parts and the generation of the respective constraints is fully automatic and does not require any user intervention.

III. Modeling of Arithmetic Operators

Basic arithmetic word-level operators include: adders, subtractors, comparators, multiplexers, shifters, and multipliers. Any other word-level operators can be represented in terms of these using Boolean connectors (AND, OR, NOT). Most of the following linearization techniques of the word-level operators can be found in [15] [18].

A. Adder/Subtractor

Addition and subtraction are both linear operators and can be trivially represented as equality constraints: $C = A \pm B$, where A, B, C are all word-level variables.

B. Comparator

There are 6 types of comparison operators: $\leq$, $<$, $>$, $\geq$, $\neq$, and $=$. Consider, as an example, $s = A < B$, where the s is a binary Boolean variable. It can be modeled as follows:

$$A - B - L*(1-s) \leq -1 \tag{1}$$
$$A - B + L*s \geq 0 \tag{2}$$

Here L is a constant, $L \geq max(A,B)$. All other types of comparison operators can be derived similarly.

C. Multiplexor

In principle, the *multiplexor* can be treated as part of a Boolean logic. However, if the inputs to the multiplexor are word-level signals, it is more efficient to represent it as an arithmetic operator. Consider $Z = MUX(A, B, s)$, where A, B and Z are word-level variables and the selection signal s is a Boolean variable. For $s = 1$, $Z = A$, otherwise $Z = B$. The ILP constraints for the *multiplexer* are:

$$Z - A - L*(1-s) \leq 0 \tag{3}$$
$$A - Z - L*(1-s) \leq 0 \tag{4}$$
$$Z - B - L*s \leq 0 \tag{5}$$
$$B - Z - L*s \leq 0 \tag{6}$$

with constant $L \geq max(A,B)$,

D. Multiplier: $Z = X * Y$

Since multiplication is a non-linear operator, one of its input word-level operands, X or Y, has to be expanded in terms of Boolean variables. The choice of which operand to expand is dictated by its interaction with the rest of the circuit. The best candidate is the one which is driven by the Boolean part, such as port B of the multiplier in Figure 1. In the following, let X be an n-bit variable to be expanded and $P_i = X_i \times Y$ be a partial product. Note that X_i has to be declared as a binary variable, but P_i is left as a *continuous* variable because P_i will take integer value automatically. The ILP constraints for the multiplier are:

$$Z = \sum_{i=0}^{n-1} 2^i * P_i \tag{7}$$

and for each $i \in [0, \ldots, n-1]$,

$$P_i - L*X_i \leq 0 \tag{8}$$
$$P_i - Y + L*(1-X_i) \geq 0 \tag{9}$$
$$0 \leq P_i \leq Y \tag{10}$$

Practical designs often contain operands wider than 32 bits. Due to an inherent 32-bit word representation of the current MILP solvers such wide operators cannot be handled directly. According to our experience with the commercial MILP solver CPLEX, numerical problems might occur for signal wider than 24 bits. As a result, wide operators must be decomposed into smaller ones. For more details regarding the decomposition, the reader is referred to [18].

IV. Modeling of Boolean Logic

We now present the modeling of the Boolean logic using ILP constraints, compatible with those derived for arithmetic portion of the circuit.

A straightforward way of translating Boolean logic into a mixed integer-linear program is to first map the Boolean part into AND-OR-NOT network, and then derive integer-linear constraints for each logic gate. All the primary inputs and outputs of the Boolean network are declared as *binary* variables, while the rest of intermediate nodes are left as *continuous* variables. Due to the binary nature of the Boolean network the continuous variables will automatically assume binary values as well.

A. AND Gate

Consider a 2-input AND gate with inputs A, B and output Z. According to its truth table, the following is always true: $Z \leq A$ and $Z \leq B$. Output Z takes value 1 when both inputs are 1, hence $Z \geq A + B - 1$. $Z \geq 0$ is implicit for LP. In summary, the linear constraints for the 2-input AND gate are as follows.

$$Z \leq A \tag{11}$$
$$Z \leq B \tag{12}$$
$$Z \geq A + B - 1 \tag{13}$$
$$\tag{14}$$

This can be readily extended to an *AND* gate with an arbitrary number of inputs. Let Z be an output and A_i be an i-th input of an AND gate. Then:

$$Z \leq A_i, \quad \forall i \in \{1, ..., n\} \tag{15}$$
$$Z \geq \sum_{i=0}^{n-1} A_i - (n-1) \tag{16}$$
$$\tag{17}$$

B. OR Gate

Similarly, an n-input *OR* gate can be linearized as follows. Here Z is the output and A_i is the i-th input.

$$Z \geq A_i, \quad \forall i \in \{1, ..., n\} \tag{18}$$
$$Z \leq \sum_{i=0}^{n-1} A_i \tag{19}$$
$$Z \leq 1 \tag{20}$$

C. NOT Gate

A *NOT* gate with input A and output Z can be trivially modeled as: $Z = 1 - A$.

The described method for modeling the Boolean logic as ILP constraints proved the most effective from the computational complexity point of view. We also experimented with other methods, such as constructing a BDD and deriving MUX-like equations (see section III-C) for the BDD nodes, but the results were inferior in most cases.

V. Experimental Results

The LPSAT tool was implemented in the framework of VIS [19]. One of the merits of our method is that no manual intervention is needed for the entire SAT process; starting with the Verilog RTL description of the circuit, the partitioning onto arithmetic and Boolean parts and the generation of the MILP constraints are done in a fully automatic fashion. These constraints are then passed to the commercial MILP solver [20] and the results verified by applying the obtained vectors to the original circuit structure in VIS.

We compared the performance of LPSAT with two CNF-based algorithms, SATO [21] and GRASP [7], and with a BDD-based satisfiability tool B-SAT [14] over a range of benchmarks. In order to get a fair comparison, the overhead associated with transforming the circuit network into CNF formulas were ignored. Similarly, we also ignored the overhead associated with translating the Boolean part into a network of AND-OR-NOT gates (which is really minimal). All experiments were performed on a Pentium III/500MHz PC running Linux.

The test cases used in our experiments, shown in Table I, are extracted from RTL models. The circuit *square* corresponds to a design whose output asserts high if $(Z^2 = X^2 + Y^2)$, where X, Y and Z are 16-bit wide operators. The SAT instances $square(1)$ and $square(0)$ correspond to the output being asserted to 1 and 0, respectively. The benchmark *quadratic* is an implementation of a solution to the quadratic equation $X^2 + a*X + b = 0$, where a and b are constants and X is a 16-bit wide variable. Given the constants a and b, the SAT instance corresponds to computing the value of X. Examples $linear(1)$ and $linear(2)$ are circuits with a relatively simple structure (a chain of comparators) but with a large number of primary inputs (over 1200). The two instances differ in their size. $gcd20$ and $gcd40$ are extensions of the greatest common divisor (GCD), a 24-bit input sequential circuit. They are generated by unrolling the GCD circuit over 20 and 40 time frames, respectively. $m13 \times 13$ and $m16 \times 16$ are 13-bit and 16-bit multipliers. Two different SAT instances for each were created: (*sat*) with a feasible solution, and (*non*) with a non-satisfiable requirement. Finally, $mdpe(1)/(2)$, is a circuit composed of a multiplier feeding a dynamic priority encoder, taken from a realistic design. The two cases differ in the size of the Boolean part of the circuit.

It should be emphasized, that all the test cases were comprised of both, the arithmetic and the Boolean parts, including the multiplier circuits (the structure of unsigned multipliers was obtained by a recursive set of adders, and required certain amount of connecting Boolean logic).

In Table I, column 2 (*# constr*) gives the number of linear constraints generated by LPSAT, while column 3 shows CPU time for solving LPSAT. Columns 4 and 5 give the number of literals and the number of clauses, respectively, in the CNF formulas. The CPU times for CNF-based algorithms are listed in columns 6 and 7. Finally, column 8 gives the CPU time for solving the same examples using a BDD-based Boolean satisfiability tool, BSAT [14].

As shown in Table I, in most test cases the run-times of LPSAT are significantly smaller than those of gate-level CNF-based SAT algorithms. There is one exception, $square(1)$, which could be successfully solved only by GRASP. This one was the only test case that LPSAT was unable to solve in the preset amount of time. In contrast, SATO and GRASP could not solve four test cases. Although the CNF-based SAT algorithms have made a lot of progress in the recent years and have been successfully applied in ATPG [4] [5], these approaches have to deal with large numbers of CNF clauses. As a result, they have difficulties on some large circuits with even simple structure, such as $gcd20$ and $gcd40$. Similarly, the BDD-based satisfiability tool, BSAT [14], could solve but small examples because of the excessive time/memory needed to create BDDs for the tested circuits.

It would be desirable to compare our methods with that of HSAT [15]. While these circuits were made available to us, all except $mult16$ are trivial, mainly arithmetic circuits. Hence the comparison would not yield any important insight, especially since HSAT also uses CPLEX. It should also be pointed out that $mult16$ of HSAT is a *purely* arithmetic circuit, with no Boolean part, unlike our $m16 \times 16$ multiplier which contains certain amount of Boolean logic. In order to demonstrate the robustness of our approach, we have used large designs which required up to about 250K clauses in the equivalent CNF representation, while the largest design in [15] requires no more than 1.4K clauses.

Theoretically, the worst case computational complexity of LPSAT is exponential in the number of binary/integer variables. The program has an option to explicitly specify the priorities of binary variables to branch on. We did experiments in which primary inputs were set to assume either higher or lower branch priority over the intermediate signals. Unfortunately, the resulting CPU time over the tested examples was case-dependent and no consistent observation could be made.

VI. Conclusions and Future Work

The presented LPSAT technique solves satisfiability problem for RTL designs by modeling the constraints of arithmetic and Boolean logic in a unified MILP environment. The unification obviates the need for multiple SAT tools operating in isolation, and renders the constraint propagation between the two domains implicit to the MILP solver, thus enhancing the overall efficiency of the SAT framework.

The experimental results are quite promising when com-

Testcase	LPSAT		CNF-SAT				BSAT
	# constr	CPU time	# literals	# clauses	SATO CPU time	GRASP CPU time	CPU time
m13x13(sat)	68	0.04	7146	16704	2.51	187.24	137
m13x13(non)	68	0.60	7146	16704	12.12	1355.8	520
m16x16(sat)	149	44.09	10590	24720	722.35	2819.3	>3600
m16x16(non)	149	2.34	10590	24720	132.12	>3600	>3600
square(1)	701	>3600	33119	77361	>3600	1344	>3600
square(0)	701	0.96	33119	77361	>3600	>3600	>3600
quadratic	469	0.05	30759	72015	10.68	14.38	923.8
linear(1)	950	0.37	16899	36914	5.01	2.98	>3600
linear(2)	2749	1.34	35683	77887	1.27	6.73	>3600
gcd20	542	0.03	50451	117785	>3600	>3600	>3600
gcd40	1062	0.08	106423	248449	>3600	>3600	>3600
mdpe(1)	2933	1.12	12245	29560	75.2	572.27	>3600
mdpe(2)	3673	8.98	12731	30851	4.4	59.1	>3600

TABLE I
COMPARISON OF DIFFERENT SAT RESULTS

pared with generic CNF-based and BDD-based SAT algorithms. The limitations of LPSAT approach surface out when the SAT instance contains large portions of sequential control logic, or long symbolic traces spanning a large number or time frames. In this case the proposed method would reduce to using ILP for solving Boolean SAT, or generating too complex code, which may be prohibitively expensive in terms of computation time. For other SAT instances with balanced mixed arithmetic-Boolean or pure arithmetic circuit, our LPSAT seems to be a desirable approach.

On the final note, the SAT problem differs from the optimization problem as it does not require any specific cost function; any feasible solution is acceptable. Since an MILP solver is actually designed for optimization problems, we wish to explore some heuristics that would guide it to solve the simpler SAT problem faster. We also would like to investigate the application of LPSAT approach to complex sequential circuits with a large number of time frame instances.

REFERENCES

[1] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[2] C. E. Blair and *et al.*, "Some Results and Experiments in Programming Techniques for Propositional Logic," *Comp. and Oper. Res.*, vol. 13, no. 5, pp. 633–645, 1986.

[3] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," *Ph.D. Dissertation, Dept. of Comp. and Inf. Sc., Univ. of Penn.*, May 1995.

[4] T. Larabee, *Efficient Generation of Test Patterns using Satisfiability*, Ph.D. thesis, Dept. of Computer Science, Stanford University, Feb. 1990.

[5] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation using Satisfiability," Tech. Rep. UCB/ERL M92/112, Dept. of EECS., Univ. of California at Berkeley, Oct. 1992.

[6] R. Zabih and D. A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability," in *Proc. Natl. Conf. on AI*, 1988.

[7] J. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," in *ICCAD'96*, 1996, pp. 220–227.

[8] L. G. Silva, L. M. Silvera, and J. Marques-Silva, "Algorithms for Solving Boolean Satisfiability in Combinational Circuits," in *Proc. DATE*, March 1999, pp. 526–530.

[9] R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, August 1986.

[10] K. S. Brace, R. Rudell, and R. E. Bryant, "Efficient Implementation of the BDD Package," *Proceedings of the Design Automation Conference*, pp. 40–45, 1990.

[11] S. Jeong and F. Somenzi, "A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations," in *ICCAD*, 92.

[12] B. Lin and F. Somenzi, "Minimization of Symbolic Relations," in *ICCAD*, 90.

[13] T. Villa and *et al.*, "Explicit and Implicit Algorithms for Binate Covering Problems," *IEEE Trans. CAD*, vol. Vol. 16, no. No. 7, pp. 677–691, July 1997.

[14] P. Kalla, Z. Zeng, M. J. Ciesielski, and C. Huang, "A BDD-Based Satisfiability Infrastructure using the Unate Recursive Paradigm," in *Proc. of DATE 2000*, 2000, pp. 232–236.

[15] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL models using Linear Programming and 3-Satisfiability," in *Proc. DAC*, 1998, pp. 528–533.

[16] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming," *IEEE Tran. on VLSI Systems*, vol. 3, no. 2, pp. 201–214, June 1995.

[17] A. K. Chandra and V. S. Iyengar, "Constraint solving for test case generation: a technique for high-level design verification," in *Proc. of Int'l Conf. on Computer Design: VLSI in Computers and Processors*, 1992, pp. 245–248.

[18] F. Fallah, *Coverage Directed Validation of Hardware Models*, Ph.D. thesis, MIT, 1999.

[19] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vencentelli, F. Somenzi, A. Aziz, S-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, G. Shiple, S. Swamy, and T. Villa, "Vis: A system for verification and synthesis," *Proceedings of the Computer Aided Verification Conference*, 1996.

[20] *CPLEX Reference Manual*, ILOG, 1999.

[21] H. Zhang, "Sato: An efficient propositional prover," in *Proc. of 14th Conference on Automated Deduction*, 1997, pp. 272–275.