Functional Test Generation using Constraint Logic Programming

Zhihong Zeng, Maciej Ciesielski, Bruno Rouzeyre

Dept. of Electrical & Computer Engineering
University of Massachusetts
Amherst, MA 01003, USA
{zzeng, ciesiel}@ecs.umass.edu

LIRMM
Universite de Montpellier
34090 Montpellier, France
rouzeyre@lirmm.fr

Abstract—

Semi-formal verification based on symbolic simulation offers a good compromise between formal model checking and numerical simulation. The generation of functional test vectors, guided by miscellaneous coverage metrics to satisfy the simulation target, can be posed as a satisfiability problem (SAT). This paper presents a novel approach to solving SAT based on Constraint Logic Programming (CLP) technique. The proposed SAT solver allows to efficiently handle the designs with mixed word-level arithmetic operators and Boolean logic. It is applicable for designs specified at different levels, including HDL, RTL, and Boolean. The experimental results are quite encouraging compared with classical CNF-based, BDD-based, and LP-based SAT solvers.

I. Satisfiability in Semi-formal Verification

Numerical simulation remains a dominant design validation method in industry since it scales well with the design complexity. A typical design verification scenario includes random and pseudo-random directed tests, bringing the functional coverage of the design specification to a desired level. Functional coverage metrics typically include line coverage, state coverage, transition coverage, branch coverage, etc. In the early design phase, both random and directed test vectors can help to find design bugs easily and improve functional coverage quickly. When the functional coverage reaches certain level (say 90%) of coverage, improving the coverage and discovering corner cases by adding more random or manual test vectors becomes very inefficient. At this point, the remaining gap in functionality coverage can be solved by applying deterministic tests. The generation of such tests must satisfy a predefined simulation target, such as reaching a particular state of the design, exercising a branch, or covering a piece of HDL code, and is guided by miscellaneous coverage metrics and monitors.The functional test generation problem guided by such constraints can be posed as a satisfiability problem (SAT).

Several tools have been developed in industry and academia to facilitate the generation of deterministic test vectors. SIVA [1] is an example of such a tool, used to generate input vectors to exploit more state space and check the desired properties. The core algorithms in SIVA use a combination of BDD-based and ATPG tools to solve satisfiability. In our context of semi-formal verification, a *symbolic simulation* engine is used to generate a set of symbolic expressions according to the simulation targets. The set of symbolic expressions is then transformed into a SAT instance. A solution to this SAT problem gives a sequence of input vectors to exercise the simulation target, or proves that it is not possible to find such vectors.

SAT belongs to the class of NP-complete problems, with algorithmic solutions having exponential worst-case complexity. Hence the efficiency of the semi-formal verification approach is largely determined by the performance of the SAT solvers. In this paper, we investigate a method for solving SAT problems that originate from RTL designs with mixed arithmetic and control logic, that are common in the datapath of modern microprocessor and DSP designs.

II. Previous approaches in Solving Satisfiability

Classical approaches to SAT are based on variations of the well known Davis and Putnam procedure [2] which works on CNF formulae. Typical versions of this procedure incorporate a chronological backtrack-based search [3]; 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. Recent approaches incorporate learning techniques and other conflict analysis procedures with *non-chronological* backtracks to prune the search space [4].

Another popular approach to solving the Boolean satisfiability problems is based on Binary Decision Diagrams (BDDs) [5]. 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 [6].However, a major limitation of this approach is the memory explosion problem associated with the construction of the product BDD. Recently, Kalla *et al.* [7] 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.

CNF-based SAT solvers can be directly integrated into the semi-formal verification framework. However, the practical RTL or behavioral descriptions often have word-level operators. Collapsing those word-level operators into a single CNF formulae destroys the regularity of the problem and often makes the problem much harder to solve. On the other hand, BDD-based techniques suffer from the size explosion problems. For example, the size of a BDD for a multiplier is exponential, regardless of the variable ordering.

To overcome these drawbacks, Fallah *et al.* [8] proposed a *hybrid* satisfiability approach, HSAT, to generate functional test vectors for structured HDL designs. Working on the RTL descriptions, the hybrid method generates a set of *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 Boolean logic, while a Linear Programming (LP) technique is used to check the feasibility of linear constraints for arithmetic portion of the design. It should be noted that the two problems, SAT for Boolean logic and LP for arithmetic blocks, are solved separately, each in its own domain. In such a framework, backtracks between the two solution engines are inevitable and performed explicitly. Hence the performance of HSAT is limited by the heuristics that select the set of assignments to Boolean variables. Constraint propagation techniques between different domains have been explored to generate test vectors and check assertions on HDL descriptions [9] [10]. Sometimes, word-level ATPG and modular arithmetic constraint-solving technique are combined to solve the SAT problems [10]. Again, those techniques rely on heuristics to propagate the constraints between the arithmetic and Boolean domains.

It would be desirable to use an infrastructure that can represent *both* the 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. Zeng *et al.* [11] presented an enhanced word-level satisfiability solver, LPSAT, based on linear programming. By generating linear constraints for both the Boolean logic and the arithmetic operators, this approach allows to solve the SAT problem in a unified integer linear programming (ILP) domain. By doing so, LPSAT utilizes the implicit constraint propagation of the ILP solver.

However, such generic ILP solvers tend to be inefficient in solving satisfiability problems encountered in RTL verification. First, generic LP solvers are based on numerical procedures that are designed predominantly to solve optimization problems, rather than satisfiability. As a result, they suffer from numerical convergence problems, and are sensitive to a number of internal parameters. Also, they tend to be inefficient in the branch and bound part for solving the decision problems, which are at the heart of SAT problems. Secondly, any non-linear arithmetic operators in LP-based SAT has to be explicitly linearized into linear constraints. This includes the modeling of mixed arithmetic/Boolean blocks, such as comparators, shifters, multiplexors, etc., with integer decision variables, that may lead to the numerical convergence problems. Finally, the ILP can only compute a single solution; it is computationally expensive to force it to produce several different solutions during subsequent runs.

In this paper we investigate a new satisfiability checker based on *Constraint Logic Programming* (CLP). By transforming the SAT instances into predicates in *Logic Programming*, we preserve the regularity of the word-level operators. Compared to LP, the modeling of implications, often encountered in the verification problems, is simpler and more natural for CLP. Also the modeling of mixed blocks is easier: it does not require the introduction of (integer) variables and is not plagued with the numerical convergence problems. Another important aspect of this approach is that it allows to generate multiple vectors, needed for simulation-based functional validation. Finally, efficient modeling of both arithmetic and Boolean domains inherent to CLP makes it applicable not only to satisfiability (or justification), but also to simulation (numerical and symbolic), and equivalence checking.

The rest of the paper is organized as follows: Section III explains how to generate a SAT problem from symbolic simulation using Prolog predicates. Section IV discusses a practical aspect of modeling wide arithmetic operators. Finally, Section V gives the experimental results, and Section VI contains concluding remarks.

III. Formulating a SAT problem for RTL Validation

A. Symbolic Simulation

The first step in our verification flow is to generate symbolic expressions using symbol propagation techniques. The resulting design description remains in the text format, hence minimizing a risk for memory blow-up, commonly encountered in BDD based representation [12]. Then, the symbolic expressions of a SAT instance is translated into an internal representation (such as BLIF format) so that different kinds of SAT solvers can be applied to solve the SAT instance. The generated expressions capture only the portion of the design which lies in the cone of influence of the simulation target, or a static assertion property. Thus a SAT problem generated by this approach remains small even from a large design. A simulation target could be described as simply as: *"Output signal A must take value* 100 *after 5 clock cycles from the current simulation time"*. Or it can be as complex as exercising different branches at different clock cycle. An example of a static property is: *"No multiple drivers are allowed in a common bus at the same time"*. Putting the symbolic expressions together with the constraints encoding the simulation targets or properties, a SAT instance is obtained.

Figure 1 shows a 4-state finite state machine (FSM) for a simple datapath circuit with initial state $S1$. Assume, through numerical simulation that state $S2$ is reached from initial state $S1$ by a sequence of input vectors. Starting from state $S2$, called the *seed environment*, we want to verify the following target: *"is the machine able to return to the initial state $S1$ in two clock cycles?"*. Through a combination of symbolic simulation and SAT, we are able to formally answer the above question. The generated symbolic expressions are as follows:

$$\begin{aligned} S^2 &= f(A^1, B^1, Ctl^1, S^1) \\ &= f(A^1, B^1, Ctl^1, f(A^0, B^0, Ctl^0, (S^0 = S2))) \end{aligned}$$

where $f(A, B, Ctl, S)$ is the next state function, A^i denotes the symbolic input variable A during the i'th clock cycle. Together with the simulation target $S^2 == S1$, a SAT problem is formed.

In case when the SAT problem cannot be solved within reasonable/allowed time, we can decrease the problem size by fixing some symbolic variable, such as Ctl in Figure 1, to a constant value. In this case, we trade the SAT performance for the completeness. In the above example, if the symbolic variable Ctl is fixed to constant '0', then we may fail to explore some parts of state space by the two-cycle symbolic simulation.

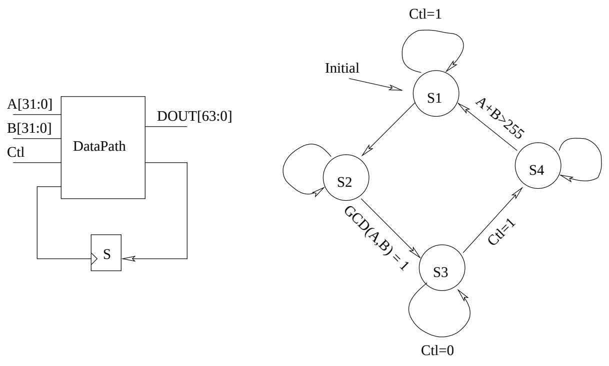

Fig. 1. A FSM for a simple datapath circuit

B. Symbolic Expressions in Prolog

Constrained Linear Programming (CLP) is a constraint solving method based on logic programming. In recent decades, CLP drew extensive research interests and made a lot of progress in solving practical problems [13]. There are many publicly available CLP solvers based on different constraint solving techniques. Among them *GNU Prolog* (GProlog) [14], [15] has reported a good performance, even comparable to some of the commercial tools. It is a native Prolog compiler with constraint solving over the finite domain, which makes it especially suitable for solving our SAT problems.

The symbolic expressions are first translated into the widely accepted BLIF format, with the annotation made for any sub-module, if it is a word-level operator. The BLIF file is then transformed into a Prolog program. The GNU Prolog solver we used supports many built-in predicates in finite domain. These built-in predicates made our translation task from BLIF to Prolog quite straightforward. Table I has some examples illustrating how to model Boolean gates and arithmetic operators in terms of GNU Prolog predicates. Here '#∧' means AND, '#∨' means OR, '#\' means NOT and '# <=>' means equivalence. For more details of the usage of GNU Prolog predicates, the reader is referred to [15].

Type of Operators/Gates	**GNU Prolog Predicates**
Z = and(A, B)	A #∧ B #<=> Z
Z = or(A, B)	A #∨ B #<=> Z
Z = not(A)	A #\ Z
Z = A $< +\|-\|* >$ B	Z #= A $< +\|-\|* >$ B
z = A < B	z #<=> A #< B
A ⇒ B	A #==> B
z = A == B	z #<=> A #= B
Z = mux(A, B, s)	Z #= s * A + (1-s)*B

TABLE I
MODELING OF BOOLEAN LOGC AND ARITHMETIC OPERATORS BY LOGIC PREDICATES

The following is a description of a design containing both arithmetic blocks and Boolean logic, shown in Figure 2. Here $A_{hi} = A[31:28]$, $A_{lo} = A[27:0]$, and similarly for B. For simplicity, the equations are given using standard mathematical notation for compare $(>, <)$, implication $(\Rightarrow)$ and others.

$$\begin{cases} m = (A_{hi} < B_{hi}) \\ n = (A_{hi} == B_{hi}) \\ k = (A_{lo} < B_{lo}) \\ d = n \wedge k \\ c = m \vee d \end{cases} \tag{1}$$

Finally we should comment here on the modeling of non-linear operators, such as multiplier. In general, one of the word-level inputs (operands) of a a non-linear operator must be expanded in terms of its bits. The choice of the operand to be expanded is dictated by its interaction with the remaining part of the circuit. The best candidate for bit-wise expansion is the one which interacts with the Boolean part. In the following hypothetical code, a multiplier is modeled as a sum of partial products, where X is expanded into an n-bit variable, and $P_i = X_i * Y$ is a partial product associated with bit i. In principle, variable X_i has to be declared as a finite domain element, while P_i is left as a *bounded* variable, because it will assume an integer value automatically.

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

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

$$\begin{cases} X_i = 0 \Rightarrow P_i = 0 \\ X_i = 1 \Rightarrow P_i = Y \end{cases} \tag{3}$$

Notice that the above description is simpler than the one modeling an MPLY operator in LPSAT [11] (equations 7-10), as it does not require any auxiliary integer variables.

Furthermore, *Gprolog* solver used in our tool has a built-in predicate '*' that allow us directly model the multiplier as $Z = X * Y$, without an explicit expansion. This is another important advantage over LPSAT.

IV. Handling Wide Word-Level Operators

In realistic designs and RTL specifications, wide word-level signals (with bit width larger than 32 bits) are common. Unfortunately, the largest integer domain that can be allowed in GNU Prolog solver is currently limited to 2^{28}. Any wide operator greater than 28 bits has to be decomposed into smaller blocks. For example, a 32 bits comparator $c = (A[31:0] < B[31:0])$ can be decomposed into three smaller arithmetic operators and two Boolean gates, as shown in Figure 2.

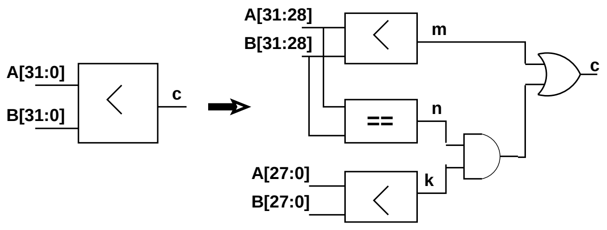

Fig. 2. Decomposing a wide word-level comparator

There are two ways to decompose wide operators. In our experiments, the decomposition is done during the translation of symbolic expressions into BLIF representation. The other possibility is to perform the decomposition during the translation from BLIF (or any other intermediate format) to Prolog. This requires creating user-defined predicates (macros), with wide word-level operands decomposed into a set of sub word-level vectors.

V. Experiments

We did a preliminary implementation of our SAT solver by integrating GProlog into our satisfiability solving framework. The experimental results are quite encouraging compared with those of other satisfiability solvers. The whole process of reading the RTL Verilog design, decomposing wide operators, generating symbolic expressions, and solving SAT using GProlog is done automatically, without nay human intervention. It is implemented in the framework of VIS system [16].

We compared our CLP-based SAT solver, *CLP-SAT*, to another word-level solver, LPSAT [11]; two CNF-based solvers, SATO [17] and GRASP [4]; and a BDD-based tool, B-SAT [7] over a range of available benchmarks. The overhead associated with transforming the SAT instance into CNF formulas were ignored. In order to get a fair comparison, we also ignored the overhead associated with translating SAT instance into linear constraints for LPSAT or predicates for GProlog. We observed that such a translation was within seconds or less for the experiments conducted here. All experiments were performed on a Pentium III/500MHz PC running Linux.

A. Description of Benchmarks

In order to have a better comparison with LPSAT, we used the SAT instances generated for the functional vector generation purpose reported in [11]. The experimental results are shown in Table II.

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 two different output requirements. 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 symbolic simulation of 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 16-bit multiplier circuits (the structure of unsigned multipliers was obtained by a recursive set of adders, and required certain amount of connecting Boolean logic due to wide operator decomposition).

In Table II, column 2 is the code size of the corresponding GProlog program. Column 4 (*# constr*) gives the number of linear constraints generated by LPSAT. Columns 6 gives the number of clauses in the CNF formulae.

B. Experimental Results

Table II shows the experimental results, where '$-$' means not finishing within 3600 seconds. The CPU time is given in seconds. Column 3 shows a CPU time for CLP-SAT using GProlog. It is composed of two parts: one for the compilation (from Prolog input file to executable program), and the other for the actual execution. The compilation time ranges from about 1 to 12 seconds, which is a significant portion of the total solving time. The remaining columns report the size and performance of LPSAT, SATO, GRASP, and BSAT [7].

From the results, we can conclude that the satisfiability solver (CLP-SAT) based on GProlog competes very well with the established CNF-based solvers SATO and GRASP, and with the BDD-based SAT solvers, and is comparable to the performance of LPSAT. An interesting note is that LPSAT and CLP-SAT each failed on only one test case, $square(1)$ and $mdpe(2)$, respectively. As a general observation, the word-level SAT approaches exemplified by CLP-SAT and LPSAT

Benchmark	CLP-SAT		LPSAT		CNF-SAT			BSAT
	# lines	time comp / exe	# constr	time	# clauses	SATO time	GRASP time	time
m13x13(sat)	78	0.23 / 0.00	68	0.04	16704	2.51	187.24	137
m13x13(non)	78	0.24 / 0.00	68	0.60	16704	12.12	1355.8	520
m16x16(sat)	116	0.29 / 0.37	149	44.09	24720	722.35	2819.3	–
m16x16(non)	116	0.24 / 53.1	149	2.34	24720	132.12	–	–
square(1)	529	0.58 / 0.00	701	–	77361	–	1344	–
square(0)	529	0.82 / 0.00	701	0.96	77361	–	–	–
quadratic	413	0.78 / 4.29	469	0.05	72015	10.68	14.38	923.8
linear(1)	1109	1.53 / 0.00	950	0.37	36914	5.01	2.98	–
linear(2)	3527	11.7 / 0.01	2749	1.34	77887	1.27	6.73	–
gcd20	876	1.10 / 0.01	542	0.03	117785	–	–	–
gcd40	1515	1.90 / 0.01	1062	0.08	248449	–	–	–
mdpe(1)	147	0.46 / 0.67	2933	1.12	29560	75.2	572.27	–
mdpe(2)	685	5.32 / –	3673	8.98	30851	4.4	59.1	–

TABLE II
COMPARISON OF DIFFERENT SAT RESULTS

work well on large yet simple sequential designs like *GCD*. For CNF-based solvers these designs are too hard due to a large number of CNF clauses. Similarly, the BDD-based satisfiability tool, BSAT [7], could solve but small examples because of the excessive time/memory needed to create BDDs for the test circuits.

VI. SUMMARY AND FUTURE WORK

We investigated a new word-level satisfiability checker based on CLP. The new SAT checker is successful applied to solve problems posed from semi-formal verification area. The preliminary promising results demonstrated that the proposed CLP based SAT solver can be a good alternative to other word-level SAT solvers like LPSAT [11] in verifying RTL designs with mixed arithmetic and Boolean logic. In future research works, we will continue to try out some other CLP solvers besides GProlog and will try to understand better the modeling SAT instance by Prolog since there are some inconsistent results like $mdpe(2)$ in Table II.

Sequential equivalence checking is one of the verification approach to checking equivalence between specification and implementation, or among different implementation levels. Ritter [18] performs sequential equivalence checking based on symbolic simulation. While generating symbolic expressions on-the-fly by parallel simulating the two compared descriptions, his approach classifies the expressions into different equivalence classes. The techniques of detecting equivalence among symbolic expressions are not limited to one approach, i.e. any technique could be applied to if appropriate. Our CLP based SAT solver is particularly effective on mixed word-level and bit-level SAT problems. So we conceive that the presented SAT approach could be also helpful in the context of sequential equivalence checking.

REFERENCES

[1] M. K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing simulation with BDDs and ATPG," in *Proc. of Design Automation Conf.*, June 1999.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] C. Huang and K.-T. Cheng, "Assertion checking by combined word-level atpg and modular arithmetic constraint-

solving techniques," in *Proc. of 37th Design Automation Conf.*, June 2000, pp. 118–123.

[11] Z. Zeng, P. Kalla, and M. Ciesielski, "LPSAT: A unified approach to rtl satisfiability," in *Proc. DATE*, March 2001, pp. 398–402.

[12] R. E. Bryant, "Symbolic simulation–techniques and applications," in *Proc. of 27th Design Automation Conf.*, June 1990, pp. 517–521.

[13] Joxan Jaffar and Michael J. Maher, "Constraint logic programming: A survey," *The Journal of Logic Programming*, vol. 19 & 20, pp. 503–582, 1994.

[14] Daniel Diaz and Philippe Codognet, "The GNU prolog system and its implementation," in *SAC (2)*, 2000, pp. 728–732.

[15] Daniel Diaz and Philippe Codognet, "gnu.org/software/prolog," 1999.

[16] 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.

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

[18] Gerd Ritter, "Sequential equivalence checking by symbolic simulation," in *Proc. FMACD*, 2000.