Functional Test Generation Using Constraint Logic Programming

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

Bruno Rouzeyre
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 technique. The proposed SAT solver allows efficiently handling 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.

Key words: Functional test generation, Satisfiability, Constraint Logic Programming, Validation, Verification

1. INTRODUCTION

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

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35597-9_40
M. Robert et al. (eds.), *SOC Design Methodologies*
© IFIP International Federation for Information Processing 2002

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 [9] is an example of such a tool, used to generate input vectors to exploit a larger 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 semiformal 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.

2. PREVIOUS APPROACHES IN SOLVING SAT

2.1 Boolean Satisfiability

Classical approaches to SAT are based on variations of the well-known Davis and Putnam procedure [4] that works on CNF formulae. Typical versions of this procedure incorporate a chronological backtrack-based search [8]; 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 [15].

Another popular approach to solving the Boolean satisfiability problems is based on Binary Decision Diagrams (BDDs) [2]. 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 [13]. However, a major limitation of this approach is the memory explosion problem associated with the construction of the *product* BDD. 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.

CNF-based SAT solvers can be directly integrated into the semi-formal verification framework. However, practical RTL or behavioral descriptions often have word-level operators. Collapsing those word-level operators into 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.

2.2 Hybrid Approaches

To overcome these drawbacks, Fallah *et al.* [7] 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.

Constraint propagation techniques between different domains have also been explored to generate test vectors and check assertions for HDL descriptions [10], where word-level ATPG and modular arithmetic constraint-solving technique are combined to solve SAT problems. During the justifications in the Boolean domain, Boolean constraints are propagated to arithmetic domain using word-level implications as early as possible. However, the *word-level implications* on arithmetic operators are much weaker than the constraint propagation of a generic constraint solving technique, for example techniques described in [12], can provide. In other

words, constraint propagation in such framework is not efficient andrelies on heuristics.

The following aspects are the main disadvantages of a hybrid approach with separate domains, compared to a unified approach.

(1) *Constraint propagation across two domains is inefficient.* This can be demonstrated with an example in Figure 1. Assume that $a=1$ is the objective we want to satisfy. Figure 2(a) shows a possible decision tree for a hybrid approach, where either arithmetic constraints are not propagated to the Boolean domain as early as possible [7] or implication engine is not robust enough [10]. A better solution is shown in Figure 2(b), where putting Boolean constraints and word-level arithmetic constraints together results in a decision tree with early reduction of the search space.

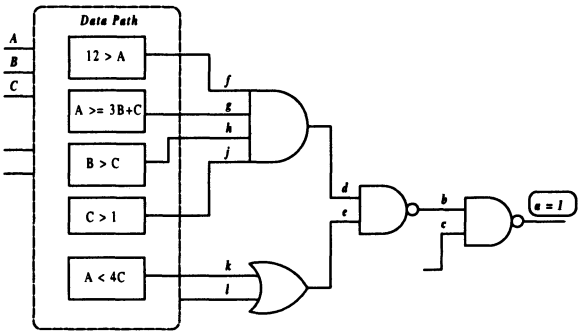

Figure 1. A circuit example for constraint propagation

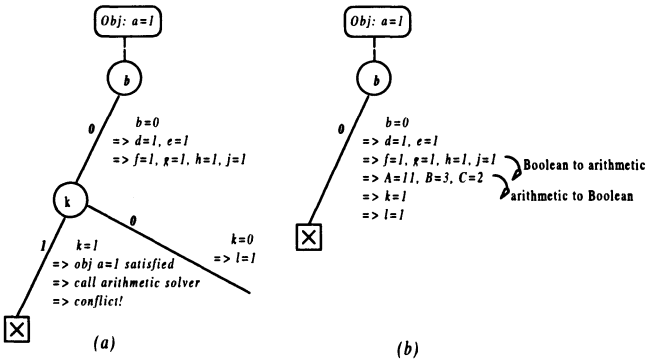

Figure 2. Decision tree with constraint propagation between two domains, (a) after objective justified in Boolean domain (b) as early as possible

(2) *Conflict-based learning across the boundary is difficult, if not impossible.* Assume that we are using a unified approach to solve a SAT problem in Figure 3(a), where in the middle of a Branch-and-Bound process the objectives k=1 and l=1 have to be justified. Figure 3(b) is a possible decision tree corresponding to this search. Upon a conflict occurring at node

x, we can learn that $[p=1 => j=0]$ and $[C=5 => j=0]$ and by contra positive law, $[j=1 => p=0]$ and $[j=1 => C\neq 5]$. Such a learning, especially when involving both Boolean and integer variables, is difficult to be derived and maintained in a hybrid approach.

2.3 Unified Word-Level Satisfiability

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.* [16] 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.

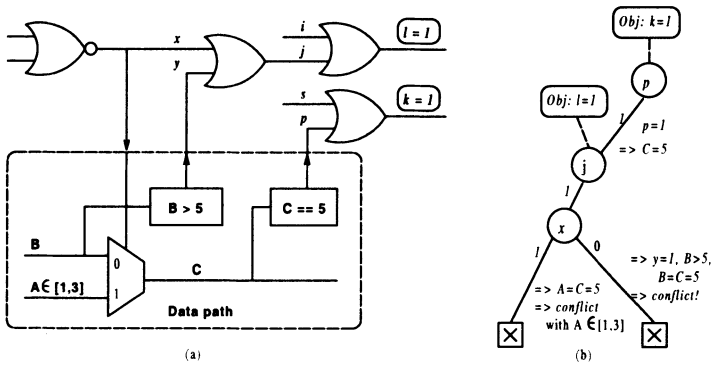

Figure 3. A circuit for conflict based learning (b) a possible decision tree in a unified solver.

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 nonlinear arithmetic operator 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 solver 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 generating 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 3 explains how to generate a SAT problem from symbolic simulation using Prolog predicates. Section 4 discusses a practical aspect of modeling wide arithmetic operators. Finally, Section 5 gives the experimental results, and Section 6 contains concluding remarks.

3. FORMULATING SAT PROBLEM FOR RTL VALIDATION

3.1 Functional Test Generation

Deterministic functional test generation is used to improve the functional coverage, especially targeting corner cases and hard-to-detect functional faults. Figure 4 shows a design validation flow on top of the functional test generation [11]. First, random simulation can be used to bring a design into a certain state, called *seed environment*. Starting at the seed environment, symbolic simulation is performed for several consecutive clock cycles (i.e. bounded time frames). A set of symbolic expressions is then generated and converted to a SAT problem.

The generation of symbolic expressions is guided by a *simulation target*, which is defined as a set of properties that needs to be verified through a simulation run. A simulation target can be specified by the user or derived from a coverage metric. It could be stated as simply as: "*Output signal A must take value h'109 after 5 clock cycles from the current simulation time*". Or it could be as complex as exercising different branches at different time

frames. Static property checking can also become a simulation target since monitor statements can be added for each static property. An example of such a target is: "*Driving a common data bus from multiple sources is not allowed at the same time*". Putting the symbolic expressions together with the constraints encoding the simulation targets or properties, a SAT instance is obtained.

In the test generation flow shown in Figure 4, any generic SAT solver can be invoked to solve a SAT problem. In this paper, we focus on solving SAT problems using Constraint Logic Programming that belongs to word-level SAT solvers.

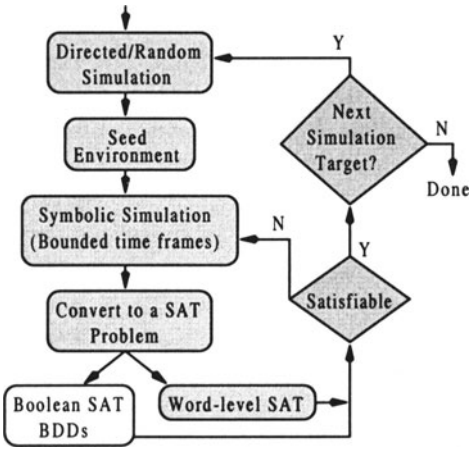

Figure 4. Functional test generation for design validation

3.2 Symbolic Simulation

Given a simulation target, symbolic expressions are generated using symbol propagation techniques. The resulting design description remains in the text format, thus minimizing a risk for memory explosion, commonly encountered in BDD-based representation [3]. Then, the symbolic expressions of a SAT instance are translated into a common intemediate 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 when originating from a large design.

Figure 5 shows a 4-state finite state machine (FSM) of a simple datapath circuit with initial state S1. Assume, that through numerical simulation state S2 is reached from initial state S1 by applying a sequence of input vectors. Starting from state S2 (seed environment), we want to verify the following simulation target: "*Is the machine able to return to the initial state S1 in two clock cycles*?" Through a combination of symbolic simulation and SAT checking, we are able to formally answer the above question. First, a symbolic trace is generated as follows:

$$S^1 = f_1(A^0, B^0, Ctl^0, (S^0 = S2))$$
$$S^2 = f_2(A^1, B^1, Ctl^1, S^1) = f_2(A^1, B^1, Ctl^1, F_1(A^0, B^0, Ctl^0, (S^0 = S2))) \quad (1)$$

where $f(A,B,Ctl,S)$ is the next state function for state variable S, A^i denotes the symbolic value of input variable A at the i'th clock cycle, S^i is the state value at the i'th clock cycle. Together with the simulation target, $S2==S1$, a SAT problem is formed.

In case when the SAT problem cannot be solved within reasonable or allowed time, we can decrease the problem size by fixing some symbolic variables, such as *Ctl* in Figure 5, 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.

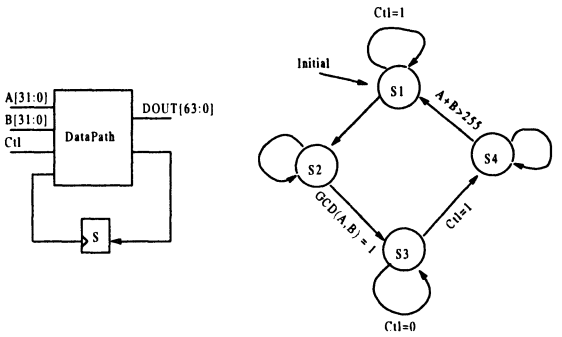

Figure 5. A FSM for a simple datapath circuit

3.3 Symbolic Expressions in Prolog

Constraint Logic Programming (CLP) is a constraint solving method based on logic programming. In recent decades, CLP drew extensive research interest and made a lot of progress in solving practical problems [12]. There are many publicly available CLP solvers based on different constraint solving techniques. Among them *GNU Prolog* (GProlog) [6, 5] 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.

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 2. Modeling of Boolean logic and arithmetic operators by Logic Predicates

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 1 has some examples illustrating how to model Boolean gates and arithmetic operators in terms of GNU Prolog predicates. Here '#∧' stands for AND, '#∨' for OR, '#\' for NOT and '#<=>' means equivalence. For more details of the usage of GNU Prolog predicates, the reader is referred to [5].

4. 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 6.

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.

5. 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 [1].

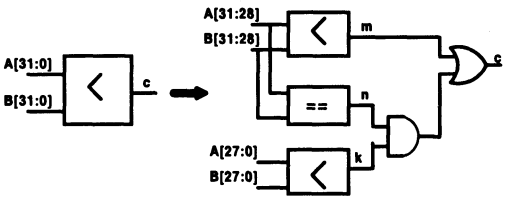

Figure 6. Decomposing a wide word-level comparator

We compared our CLP-based SAT solver, called *CLP-SAT*, to another word-level solver, LPSAT [16]; two CNF-based solvers, SATO [17] and GRASP [15]; and a BDD-based tool, B-SAT [14], 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.

5.1 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 [16]. The experimental results are shown in Table 2.

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 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 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* and *m16* 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 (certain amount of connecting Boolean logic is required due to wide operator decomposition).

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

5.2 Experimental Results

Table 2 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 [14].

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)*. As a general observation, the word-level SAT approaches exemplified by CLP-SAT and LPSAT 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 [14], could solve but small examples because of the excessive time/memory needed to create BDDs for the test circuits.

tests	CLP-SAT		LPSAT		CNF-SAT			BSAT
	# of lines	time comp / exe	# of constr	time	# of clauses	SATO time	GRASP time	time
m13(sat)	78	0.23 / 0.00	68	0.04	16704	2.51	187.24	137
m13(non)	78	0.24 / 0.00	68	0.60	16704	12.12	1355.8	520
m16(sat)	116	0.29 / 0.37	149	44.09	24720	722.35	2819.3	–
m16(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 2. Comparison of different SAT results

6. SUMMARY AND FUTURE WORK

We investigated a new word-level satisfiability checker based on Constrained Logic Programming. The new SAT checker has been successfully applied to solve problems posed from semiformal verification area. The preliminary results demonstrate that the proposed CLP-based SAT solver is a good alternative to other word-level SAT solvers, such as LPSAT [16], in verifying RTL designs with mixed arithmetic and Boolean logic.

In our future work we will continue to examine other CLP solvers besides GProlog. We shall also try to gain a better understanding of the inner workings of GProlog to explain the inconsistencies in some of the obtained results, such as *mdpe(2)* in Table 2. It is also worthy to investigate the applications of our CLP-based SAT solver for other verification purposes, such as equivalence checking and counterexample finding in modeling checking.

7. REFERENCES

[1] R. K. Brayton and et al. Vis: *A system for verification and synthesis.* Proceedings of the Computer Aided Verification Conference, pages 428-432, 1996.

[2] R. E. Bryant. *Graph based algorithms for Boolean function manipulation.* IEEE Transactions on Computers, C-35:677-691, August 1986.

[3] R. E. Bryant. *Symbolic simulation-techniques and applications.* In Proc. of 27th Design Automation Conf., pages 517-521, June 1990.

[4] M. Davis and H. Putnam. *A computing procedure for quantification theory.* Journal of the ACM, 7:201-215, 1960.

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

[6] Daniel Diaz and Philippe Codognet. *The GNU prolog system and its implementation.* In SAC (2), pages 728-732, 2000.

[7] F. Fallah, S. Devadas, and K. Keutzer. *Functional vector generation for HDL models using linear programming and 3-satisfiability.* In Proc. Design Automation Conf., pages 528-533, 1998.

[8] J. W. Freeman. *Improvements to propositional satisfiability search algorithms.* Ph.D. Dissertation, Dept. of Comp. and Inf. Sc., Univ. of Penn., May 1995.

[9] M. K. Ganai, A. Aziz, and A. Kuehlmann. *Enhancing simulation with BDDs and ATPG.* In Proc. of Design Automation Conf., pages 385-390, June 1999.

[10] C. Huang and K.-T. Cheng. *Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques.* In Proc. of Design Automation Conf.,pages 118-123, 2000.

[11] C. L. Huang. *Private communication.* Avery Design Systems, Inc.

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

[13] S. Jeong and F. Somenzi. *A new algorithm for the binate covering problem and its application to the minimization of Boolean relations.* In ICCAD, pages 417-420, 92.

[14] P. Kalla, Z. Zeng, M. J. Ciesielski, and C. Huang. *A BDD-Based satisfiability infrastructure using the unate recursive paradigm.* In Proc. DATE, pages 232-236, 2000.

[15] J. Marques-Silva and K. A. Sakallah. *GRASP - a new search algorithm for satisfiability.* In ICCAD-6, pages 220-227, 1996.

[16] Z. Zeng, P. Kalla, and M. Ciesielski. *LPSAT: A unified approach to RTL satisfiability.* in Proc. DATE, pages 398-402, March 2001.

[17] H. Zhang. Sato: *An efficient propositional prover.* In Proc. of 14th Conference on Automated Deduction, pages 272-275, 1997.