Using Synthesis Techniques in SAT Solvers

Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

Abstract In many application domains in VLSI CAD, like formal verification or test pattern generation, the problem to be solved can be formulated as an instance of satisfiability (SAT). The SAT instance in this cases is usually derived from a circuit description.

In this paper we propose to use techniques known from logic synthesis to speed up SAT solvers. By experiments it is shown that these techniques are orthotogonal, i.e. SAT instances can be simplified by logic synthesis approaches and by this are solved much faster. As a case study the techniques are applied to integer factorization – a class of problems that is known to be hard for SAT solvers. Experiments show that improvements of several orders of magnitude can be observed.

1. Introduction

SAT solvers have recently been applied to many problems in VLSI CAD with very large success [MS00]. The application domains range from "classical" SAT domains, like formal verification or test pattern generation, to routing [NSR99] or debugging [SVV03]. Especially in formal verification SAT algorithms have become the state-of-the-art proof technique in equivalence checking and (bounded) model checking. These fields largely benefit from the recent advances in efficient implementation of SAT solvers, like GRASP [MS99], Chaff [MMZ+01] or BerkMin [GN02].

In most of these VLSI CAD applications the underlying problem is given in the form of a circuit description. This is converted to a conjunctive normal form (CNF) in linear time and space. The resulting CNF is given to the SAT solver.

Motivated by these results many researchers tried to combine existing proof approaches, like BDDs or term rewriting, with SAT (see e.g. [KGP01]). In this context also implication techniques [KS97] have been proposed, i.e. how to speed up SAT solvers using recursive learning [MG99,AS00]. In general, the main idea is to combine the different approaches and get the best of these techniques. But all techniques mainly originate from the verification domain. Furthermore, the SAT instances can be simplified using pre-processing (see e.g. [GW00,LM01]). But these approaches directly operate on the CNF and do not consider the underlying circuit description.

In this paper we discuss how SAT solving can profit from using techniques from logic synthesis. The optimisation goal in synthesis is to minimize a given netlist. On the logic level, the quality of the result in technology independent synthesis is usually measured as the number of literals. This optimisation can be obtained in different ways, e.g. by local circuit transformations. If the SAT problem is derived from a circuit description, a smaller circuit corresponds to a more compact SAT instance with less variables and clauses in the CNF. For this, these instances are often easier to solve.

By an experimental study of integer factorisation this effect is demonstrated. Integer factorisation problems are known as being hard for SAT solvers (see e.g. [HW97, ARMS02]). Starting from an initial description logic optimisation is applied and it is shown that this significantly speeds up the proof process. For our experiments we make use of the latest version of zchaff. It is shown that a pre-processing of the problem instance by classical logic synthesis methods can give improvements of several orders of magnitude.

Since the approach presented can only be seen as a first step in the direction of incorporating logic synthesis approaches in SAT solvers, at the end we discuss directions for future work and give some first experimental results from the field of equivalence checking.

The paper is structured as follows: SAT is introduced in Section 2. We also briefly review how a SAT instance is generated from a circuit description. Synthesis algorithms and their relation to SAT solving are discussed in Section 3. In Section 4 our experimental results for integer factorisation are reported. Finally the results are summarized and directions for future work are discussed.

2. SAT

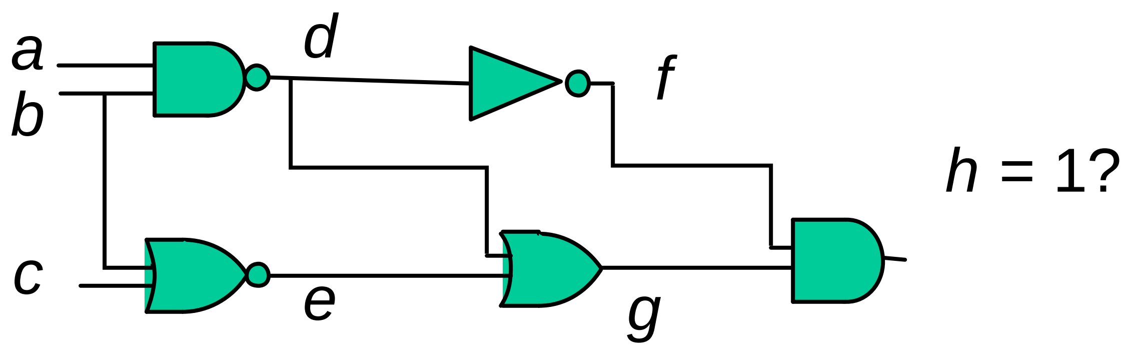
Let f be a Boolean function in conjunctive normal form (CNF), i.e. in a product-of-sum representation. Then the satisfiability problem is to determine an assignment of the variables of f such that f evaluates to 1 or to prove that such an assignment does not exist.

Example: Let $f=(x+y+\neg z)(\neg x+z)(\neg y+z)$, where $\neg x$ denotes the complement of x. Then $x = 1$, $y = 1$ and $z=1$ is a satisfying assignment, since x and y ensure that the first sum becomes 1, while z ensures this for the remaining.

2.1 SAT Instance from a Ciruit Description

In many applications, like formal verification and test pattern generation, the problem is initially given in the form of a circuit. This circuit can be transformed to a CNF by a simple transformation that is briefly described in the following.

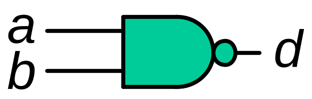
Let C be a circuit given by:



The problem is to determine whether the output h can assume the logic value 1. To derive a CNF for the circuit the netlist is traversed in linear time and at each node a sum is added to the CNF corresponding to the logic gate. E.g. for the AND gate at the output it has to be fulfilled that h=fg. For the whole circuit we derive:

$$h\ [d = \neg(ab)]\ [e = \neg(b + c)]\ [f = \neg d]\ [g = d + e]\ [h = fg]$$

We now have a closer look at a single NAND-gate:



This can be converted as follows:

$$\begin{aligned}\varphi_d &= [d = \neg(a\,b)]\\ &= \neg[d \oplus \neg(a\,b)]\\ &= \neg[\neg(a\,b)\neg d + a\,b\,d]\\ &= \neg[\neg a\,\neg d + \neg b\,\neg d + a\,b\,d]\\ &= (a + d)(b + d)(\neg a + \neg b + \neg d)\end{aligned}$$

If this transformation is applied in an analogous manner to all gates in the circuit, for the example above we derive the set of clauses:

$$\begin{aligned}&h\\ &(a + d)(b + d)(\neg a + \neg b + \neg d)\\ &(\neg b + \neg e)(\neg c + \neg e)(b + c + e)\\ &(\neg d + \neg f)(d + f)\\ &(\neg d + g)(\neg e + g)(d + e + \neg g)\\ &(f + \neg h)(g + \neg h)(\neg f + \neg g + h)\end{aligned}$$

Even though the transformation is linear in the size of the circuit it can be seen that even for the small circuit above the resulting CNF can become rather complex.

3. Synthesis Algorithms

Many different approaches to logic synthesis have been presented in the past 20 years. In the following only some of the essential aspects that are important in our application are reviewed.

The most popular techniques are based on transformations on logic networks as it is done e.g. in SIS [SSL+92]. A network is given as a graph, where each internal node represents a Boolean function (usually with a single output only). In most cases the function is represented as a sum-of-product, but alternatives, like product-of-sums or BDDs, can also be used.

Typical optimisation steps are [DeM94]:

- Elimination
- Decomposition
- Extraction
- Simplification
- Substitution
- (Algebraic) division

Alternatives to these techniques are based on functional decomposition, redundancy addition and removal or implication techniques. (For a recent overview of different synthesis methods see [DG02].)

All these techniques have in common that they try to reduce the number of literals in a given logic circuit. By this, the CNF resulting from the circuit also becomes smaller (see Section 2).

3.1 SAT Simplification based on Synthesis

Even though today's SAT solvers are very efficient, there exist several fast synthesis algorithms that can significantly reduce the problem size. Here, it is important to find a good compromise between run time of the algorithm and quality of the resulting netlist, i.e. it does not make sense to let a synthesis algorithm run for a long time to save a few gates, since the SAT solver will not profit from this.

For this, in the following we restrict ourselves to simple local operations that can be computed efficiently. This is motivated by a parameter study (see Section 4). We only made use of the standard operators available in SIS [SSL+92], like *simplification*. Furthermore, the logic synthesis techniques are only used in the form of a pre-processing.

4. Experimental Results: Integer Factorisation

All experiments are run on a SUN FIRE 280 R (900 MHz) with 4 GByte of main memory running under Unix. The program zchaff (Z-Chaff Version: ZChaff 2003.6.16) [MMZ+01] has been used as a SAT solver and all synthesis steps are run in SIS [SSL+92]. The run times are given in CPU seconds.

To demonstrate the efficiency of the approach described above *integer factorisation* has been chosen as the optimisation problem for generation of SAT instances. The problem can be described as:

> Given an integer z determine a factorisation of z in x and y, i.e. z=x*y, if it exists with x and y not equal to 1. Return the factors in case they exist and zero otherwise.

Thus, the SAT problem is satisfiable, iff z is not a prime number. It is well known that problem instances of this type are hard for SAT solvers (see e.g. [HW97, ARMS02]).

4. 1 Experimental Setup

The multiplier circuits were automatically generated and have an array structure. For the SAT instance some further constraints are needed to ensure a correct and non-trivial decomposition. I.e. it has to be ensured that after the factorisation none of the operands x or y are equal to 0 or 1. But this is equivalent to the fact that at least one bit, beside the least significant bit, has to have the value 1. This constraint can easily by added by OR-ing all bits of each operand except the lowest one and set the OR-output to constant 1. Then the outputs are set to constant values to represent the desired number z. The SAT solver then determines whether this number can be factored by finding appropriate values x and y.

4.2 Selection of Synthesis Algorithm

Since there exist many logic synthesis approaches in a first series of experiments, different techniques have been studied. Only commands integrated in SIS have been used. As a test case a non-satisfiable problem has been used, i.e. the factorisation of z=186917 using a 20-bit multiplier with 40 inputs and 40 outputs. The results are given in the Table 1. In the first column the name of the synthesis command or script is given. *Simplify* corresponds to a single command, while the scripts Boolean, Algebraic and Rugged correspond to a sequence of commands. The second and third columns give the run time of the synthesis algorithm and for zchaff, respectively. In the first row the original circuit is given without any optimisation.

Table 1: Run times for 20-bit multiplier for different logic synthesis algorithms

Synthesis procedure	Synthesis time	SAT time
Original	-	790.94
Algebraic	166.0	13.59
Boolean	65.2	9.96
Rugged	26.1	1.81
Simplify	6.0	1.95

It can be observed that all synthesis algorithms simplify the SAT instance. Compared to the original problem the speed-up is in the range from a factor of 58 to a factor of more than 400. There is no direct correlation between the run time of the synthesis algorithm and the run time needed by the SAT solver. For this, we decided to chose the simple algorithm *Simplify* for the optimisation. In general, this showed to be a good choice (see below).

4.3 Run Times

Problem instances of different complexity have been considered. The results regarding run time are given in the Table 2. In the first column the instance number is given. (The Table 3 later refers to these numbers.) *Bit* denotes the bit-width of the multiplier and *In* and *Out* gives the resulting inputs and outputs, respectively. *Integer* reports the number of z as a decimal. If z is a prime number the resulting SAT instance is not satisfiable, otherwise it is. This is explicitly denoted in column *SAT*. The run times for zchaff started on the original circuit are given in column *zchaff*. The results for the approach suggested above are given in column *sysSAT*. We give the detailed numbers for the pre-processing step *Simplify* (column *Pre*), the run times of zchaff for the simplified complexity (column *zchaff*) as well as the total run times in the last column.

Table 2: Run times for multiplier using *Simplify*

Instance	Bit	In	Out	Integer z	SAT	zchaff	sysSAT		
							Pre	zchaff	total
1	32	64	64	3719	No	-	35.6	9.84	45.44
2	32	64	64	746323	Yes	0.17	35.6	0.52	36.12
3	32	64	64	10007	No	-	35.6	3.94	39.54
4	32	64	64	600011	No	-	35.6	5.61	41.21
5	32	64	64	1073741789	No	-	35.6	-	-
6	16	32	32	678421	No	69.94	2.7	12.9	15.6
7	16	32	32	4153	No	10.65	2.7	0.32	3.02
8	16	32	32	86021	Yes	0.02	2.7	0.48	3.18
9	16	32	32	3719	No	38.04	2.7	0.32	3.02
10	20	40	40	186917	No	790.94	6.0	1.95	7.95

As can be seen the pre-processing often saves significant run time. In many cases – especially the hard ones – zchaff was able to finish the computation after a few seconds, while the instance without the synthesis step was too hard to be solved within 1 CPU hour. It is obvious that in the case of prime numbers the problem is much harder, since no satisfying inputs can be generated. It is interesting to notice that for satisfying assignments, i.e. integer numbers that can be factored, the optimization even slowed down the proof process. But in these cases it is much easier to determine a solution. E.g. for z=746323 (Instance 2) 3 factors exist, i.e. z=746323= 167 * 109 * 41. But the overhead due to the pre-processing is moderate. While in other cases the problem could only be solved within the given time limit if the pre-processing was used.

4.4 Details on Problem Instances and SAT Run

Finally, some further information on the problem sizes and on the added conflict clauses during the run of the SAT solver is given. In Table 3 for each instance the number of clauses, the number of literals, the added number of clauses and the added number of literals are given in columns *clauses, literals, conf. cl.* and *conf. li.*, respectively. The information is provided for the original description (column *Original*) and after the pre-processing using synthesis techniques (column *sysSAT*).

Table 3: Clauses and literals for multiplier using *Simplify*

Instance	Original				sysSAT			
	clauses	literals	conf. cl.	conf. li.	clauses	literals	conf. cl.	conf. li.
1	45673	109783	-	-	42627	100901	8823	922766
2	45673	109783	20	38074	42627	100901	529	76832
3	45673	109783	-	-	42627	100901	4928	400077
4	45673	109783	-	-	42627	100901	7488	848024
5	45673	109783	-	-	42627	100901	-	-
6	11065	26487	49275	11351087	10307	24341	12911	1855765
7	11065	26487	13080	1959975	10307	24341	741	50555
8	11065	26487	5	2160	10307	24341	880	86467
9	11065	26487	30289	5919916	10307	24341	778	53383
10	17509	41983	223481	84589531	16323	38585	3058	340910

It can be observed that the reduction in number of clauses and literals is not very large, but as the run times above showed the effect is significant.

5. Discussion and Future Work

In this paper a new approach to improve the performance of SAT solvers has been proposed. While most methods so far tried to combine existing techniques known from the verification domain, like BDD or ATPG, with SAT proof engines, here we suggest the use of techniques known from logic synthesis. This reduces the size of the problem description at moderate cost, while first experiments show that the reduction in run time for hard instances is significant. Experiments on integer factorisation – a problem known to be hard for SAT solvers – demonstrated the efficiency of the technique. After a pre-processing based on synthesis algorithms many instances could be solved within a few CPU seconds, while the original problem could not be solved within one hour.

The work presented in this paper can only be seen as first step in the direction of combining SAT techniques and logic synthesis. So far the synthesis approach is done in the form of a pre-processing. It is focus of current work to fully integrate the method in the SAT solver run. Motivated by the promising results of the approach obtained from a pure pre-processing step important questions are:

- How to integrate synthesis operation in a SAT solver?
- How to ensure that the synthesis techniques do not slow down the proof process (as has e.g. been observed for the satisfiable instances in Table 2)?
- How to develop fast heuristics in logic synthesis that are dedicated to this problem domain?

In contrast to logic synthesis – where a high optimisation counted in the number of literals is required – here the main focus is on reduction of run time.

Furthermore, other SAT instances resulting from equivalence checking and bounded model checking problems have to be studied. In some preliminary experiments the technique has been applied to equivalence checking. Some of the ISCAS85 circuits have been compared to a version that has been optimized by SIS using script Rugged. For some benchmarks, like *C499* and *C880*, the proof process could be sped up by a factor of 10, while for *C3540* it took nearly two times longer.

It can be expected that the technique presented in this paper works especially well in the domain of bounded model checking. There, beside the circuit a property is synthesized to gates. Usually, the property only influences a small part of the circuit and this can easily be identified by synthesis operations.

It is focus of current work to study these application domains in more detail.

Acknowledgement

The author likes to thank Marc Herbstritt for helpful discussions and for providing the instances of the equivalence checking problems considered in Section 5.

References

[AS00] F. Aloul, K. Sakallah, An Experimental Evaluation of Conflict Diagnosis and Recursive Learning in Boolean Satisfiability, IWLS, pp. 117-122, 2000

[ARMS02] F. Aloul, A. Ramani, I. Markov, K. Sakallah, Solving Difficult SAT Instances in the Presence of Symmetry, DAC, pp. 731-736, 2002

[DeM94] G. DeMicheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994

[DG02] R. Drechsler, W. Günther, Towards One-Pass Synthesis, Kluwer Academic Publishers, 2002

[GN02] E. Golberg, Y. Novikov. Berkmin: a fast and robust sat-solver. DATE'02, pages 142-149, 2002.

[GW00] J. Groote, J. Warners. The propositional formula checker HeerHugo. In Ian Gent, H. van Maaren, T. Walsh (editors), SAT20000: Highlights of Satisfiability Research in the year 2000, Frontiers in Artificial Intelligence and Applications. Kluwer Academic Publishers, 2000.

[HW97] S. Horie, O. Watanabe Hard instance generation for SAT, ISAAC'97, Lecture Notes in CS, Vol.1350, 22-31, 1997

[KGP01] A. Kuehlmann, M. Ganai and V. Paruthi, Circuit-Based Boolean Reasoning, DAC, pages 232-237, 2001

[KS97] W. Kunz, D. Stoffel, Reasoning in Boolean Networks, Kluwer Academic Publishers, 1997

[LM01] I. Lynce, J.P. Marques-Silva, The Interaction Between Simplification and Search in Propositional Satisfiability, CP'01 Workshop on Modeling and Problem Formulation (Formul '01), 2001.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an Efficient SAT Solver, DAC, pages 530-535, 2001

[MG99] J.P. Marques-Silva, T. Glass, Combinational Equivalence Checking Using Satisfiability and Recursive Learning, DATE, pp. 145-149, 1999

[MS99] J.P. Marques-Silva and K.A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability, IEEE Transactions on Computers, Vol. 48, No. 5, pages 506-521, 1999

[MS00] J.P. Marques-Silva and K.A. Sakallah, Boolean Satisfiability in Electronic Design Automation, DAC, pages 675-680, 2000

[NSR99] G.-J. Nam, K.A. Sakallah, R.A. Rutenbar, Satisfiability-Based Layout Revisted: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT, Int'l Symp. on FPGAs for Custom Computing Machines, pages 167-175, 1999

[SSL+92] E. Sentovich, K. Singh, L. Lavagno, Ch. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, SIS: A system for sequential circuit synthesis, University of Berkeley, 1992

[SVV03] A. Smith, A. Veneris, A. Viglas, Design Diagnosis using Boolean Satisfyability, 4th International Workshop on Microprocessor Test and Verification (MTV'03), 2003