Acceleration of SAT-based Iterative Property Checking

Daniel Große Rolf Drechsler

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

Abstract

Today, verification is becoming the dominating factor for successful circuit designs. In this context formal verification techniques allow to prove the correctness of a circuit and meanwhile are standard in many design flows. Formal property checking is used to check whether a circuit satisfies a temporal property. An important goal during the development of properties is the formulation of general proofs. Since assumptions of properties define the situations under which the commitments are checked, in order to obtain general proofs assumptions should be made as general as possible. In practice this is accomplished iteratively by generalizing the assumptions step by step. Thus, the verification engineer may start with strong assumptions and weakens them gradually.

In this paper we propose a new approach to speed up SAT-based iterative property checking. This process can be exploited by reusing conflict clauses in the corresponding SAT instances of consecutive property checking problems. By this the search space is pruned, since re-computations of identical conflicts are avoided. Experiments demonstrate the efficiency of our approach. In many cases up to 100% of the learned conflict clauses can be reused.

1. Introduction

Nowadays, for successful circuit designs *Property Checking* (PC) is very important. Typically such a property consists of two parts: an assume part which should imply the proof part. In the last years tools based on *Satisfiability* (SAT) performed better than classical BDD-based approaches since SAT procedures do not suffer from the potential “size explosion” of BDDs. In SAT-based PC the initial SAT instance is generated from the circuit description together with the property to be proven. This is shown in Figure 1. Usually, the largest part will result from the unrolled circuit description. In comparison,

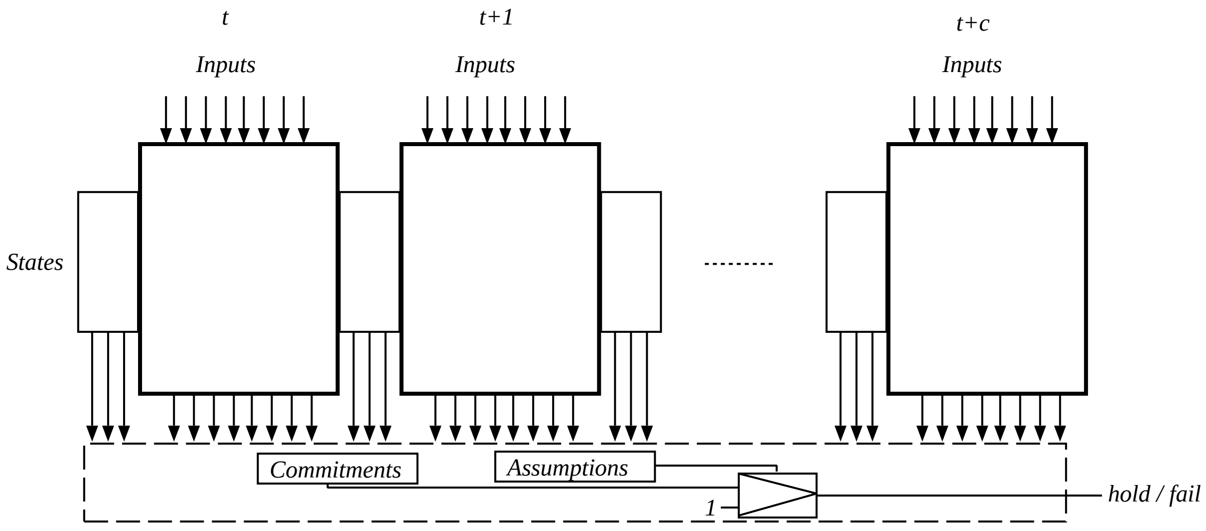

Figure 1. Input for generation of SAT instance

the parts for the commitments, assumptions, and the extra logic are much smaller. From a practical perspective, during PC as long as no design bug is found the circuit design remains unchanged, but the verification engineer modifies and adds new properties. Thus, the PC tool is used interactively. For the verification engineer on the one hand, proving becomes more easy if the assumptions of a property are very strong, i.e. the property is very restrictive and argues only over a small part of the design space. On the other hand, such proofs are not very general. Hence in practice, the formulation of a property is an iterative process. Typically, the engineer starts writing a property with strong assumptions. Then, the engineer stepwise weakens some of the assumptions to obtain a more general proof.

The basic idea is to exploit the iterative process of PC. As can be seen only a very small part of the verification problem changes in consecutive PC runs if the assumptions are weakened. Considering again Figure 1 shows that re-computations can be avoided if for the corresponding SAT problems the information learned can be used repeatedly. *Bounded Model Checking* (BMC) as introduced in [1] reduces the verification problem to a SAT problem and then searches for counter-examples in executions whose length is bounded by k time steps. For BMC, it has been suggested to reuse constraints on the search space deduced in instance k for solving the consecutive instance $k+1$ faster [7]. However, this concept is used to prove only one single property.

In this paper we use BMC as described in [8], thus, a property only argues over a finite time interval and during the proof there is no restriction to reachable states. In contrast to [7], here two SAT instances for slightly different PC problems are considered and information from the two properties with respect to the underlying circuit is utilized. This enables to reuse learned conflict clauses in the SAT instance of the consecutive PC problem.

To allow for access of necessary information during PC we have implemented a SAT-based property checker on top of zChaff [5]. For a design and a given property the property checker stores resulting conflict clauses and relevant information in a data base. In consecutive runs for a property and a derived version of the property (by weakening assumptions) some of the stored clauses can be reused. Typically this makes the current instance easier to solve, since the search space is pruned by the reusable clauses and re-computations of identical conflicts can be avoided. So a speed-up of the current proof can be

```
theorem lowestWins_2 is
assume:
    /* + is addition */
    at t: (cell1.token + cell2.token + cell3.token) = 1;     /* a1 */
    at t: (cell1.persistent + cell2.persistent               /* a2 */
           + cell3.persistent) = 0;
    at t: req1 = 0;                                          /* a3 */
    at t: req2 = 1;                                          /* a4 */
    at t: req3 = 0;                                          /* a5 */
prove:
    at t: ack2 = 1;
end theorem;
```

Figure 2. Example property `lowestWins_2`

expected. Reusable conflict clauses are such clauses that can be deduced by the intersection of the resulting clauses of the two property checking problems. We will show that these conflict clauses can be identified efficiently if the information about the source of conflicts in terms of the property and a variable mapping is preserved. Experiments show that often up to 100% of the clauses can be reused. This results in speed-ups of nearly a factor of 30 for our case studies.

The remaining part of the paper is structured as follows: In Section 2 the property language is introduced and the transformation of a PC problem into a SAT instance is given. Section 3 describes the concepts and algorithms for reusing conflict clauses in iterative PC. Experimental results demonstrating the benefits of the approach are presented in Section 4. Section 5 concludes the paper and summarizes the results.

2. Property Checking

2.1. Property Language

In the following we use a notation similar to the property checker from Infineon Technologies AG (see e.g. [3, 2] for more details). A property consists of two parts: a list of assumptions (*assume part*) and a list of commitments (*proof part*). An assumption/commitment has the form

```
      at t+a:  expression;
   or during[t+a,t+b]:  expression;
   or within[t+a,t+b]:  expression;
```

where `t` is a time point, and `a`, `b` $\in \mathbb{N}$ are offsets. If all assumptions hold, all commitments in the proof part have to hold as well. Since `a` and `b` are finite a property argues only over a finite interval which is called the *observation window*.

Example 1. *In Figure 2 the property* `lowestWins_2` *is shown. The property has been written for a scalable bus arbiter. The arbiter consists of n cells (here $n = 3$) and combines priority arbitration with a round robin scheme. The property* `lowestWins_2` *states that if exactly one token is set (a1) and no* *cell is waiting (persistent signals) (a2) and exactly request2 is high (a3, a4, a5) then the corresponding acknowledge will be set in the same clock cycle.*

In general a property states that whenever some signals have a given value, some other (or the same) signals assume specified values. Of course it is also possible to describe symbolic relations of signals.

2.2. SAT Formulation of Property Checking

In this section we briefly describe how a PC problem formulated on top of the property language introduced in the previous section can be translated into a SAT problem. The initial sequential property checking problem is converted into a combinational one by unrolling the design, i.e. by identifying the current state variables with the previous next state variables of the circuit.

A BMC instance of a property p arguing over the finite interval $[t, t+c]$ for a design D is given by:

$$\begin{aligned} b \quad = \quad & \bigwedge_{j=0}^{c-1} T_\delta(\ i(t+j), s(t+j), s(t+j+1)\ ) \ \wedge \\ & \neg\, p(\ i(t), s(t), o(t), \ldots, i(t+c), s(t+c), o(t+c)\ ) \end{aligned}$$

with

- $i(t) = (i_1^t, \ldots, i_m^t)$ inputs at time point t,
- $s(t) = (s_1^t, \ldots, s_n^t)$ states at time point t,
- $o(t) = \lambda(i(t), s(t))$ outputs at time point t and
- T_δ the transition relation.

The BMC instance b depends only on the states $s(t)$ and the inputs $i(t), \ldots, i(t+c)$. It is unsatisfiable if for all states $s(t)$ and all input sequences $i(t), \ldots, i(t+c)$ the property p over the interval $[t, t+c]$ holds for the design D. If b is satisfiable a counter-example for the property p has been found.

3. Acceleration of Iterative Property Checking

In this section the approach for reusing conflict clauses during iterative PC is presented. Before the details are given, the work flow is illustrated in Figure 3.

At first the design and the property are compiled into an internal representation. In this step information to allow for a syntactic comparison between properties is stored in the data base (A). Then the internal representation is converted into a BMC problem expressed as a CNF formula. While solving this SAT instance the references to the clauses that lead to a new conflict clause are stored in a data structure. After termination of the SAT solver this conflict clause information can be related to the single assumptions and commitments of the checked property. Finally this information is minimized and added to the data base (B). Now assume that PC is repeated but the property has been weakened. Then, this is detected (X) and before the BMC problem is given to the SAT solver conflict clauses are read from the data base, analyzed and reused (Y), if possible.

In the following, the approach is explained by an example.

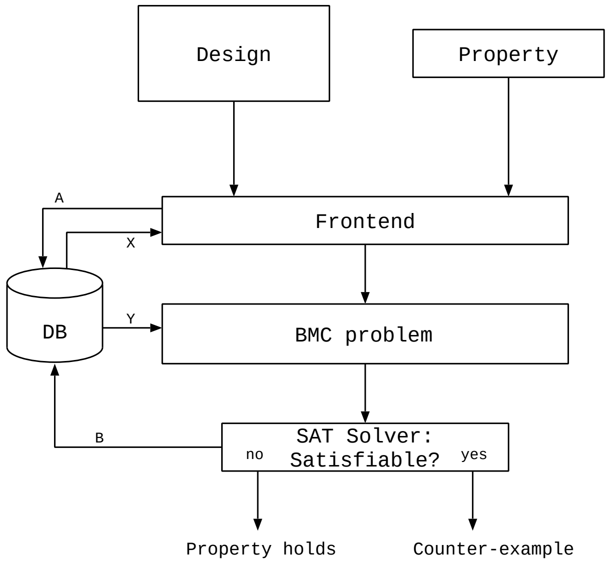

Figure 3. Property Checking Flow

Example 2. *Consider again Example 1. The property* `lowestWins_2` *has been proven and the conflict clause information has been written in the data base as explained above. Now the question is raised whether all assumptions are necessary. A detailed analysis of the arbiter and the property shows that assumption (a5) can be removed, since this assumption is too restrictive. (If req2 is 1, the value of req3 does not matter.) Proving this modified property is now faster because the re-computation of already learned conflicts is superfluous, since conflict clauses from the previous proof are reused.*

3.1. Reusing Conflict Clauses

Let M be the set of clauses resulting from the translation of the design D, let P be the set of clauses resulting from the property p. Then P can be partitioned into $P = A \cup C \cup R$, where A are the clauses from the assumptions, C from the commitments and R the clauses to “glue” the assumptions and the commitments of the property together. Now consider two consecutive runs of the property checker for the unchanged design D and for two properties p^F(first) and p^S(second). Assume that the property p^S has been derived from the property p^F by weakening some of the assumptions. Let $P^F = A^F \cup C^F \cup R^F$ be the resulting clauses of the property of the first run and $P^S = A^S \cup C^S \cup R^S$ the clauses for the second run, respectively. Further assume that the variables in P^S are renamed with a variable mapping function which maps a variable from the second set of variables V_S to the according variables of the variable set V_F from the first run. Then the following holds:

1. $C^S = C^F$, since the commitments of properties p^S and p^F are equal.
2. $R^S = R^F$ since the variables to combine the assumptions and commitments can be identified.
3. $A^S \subset A^F$ because the assumptions of p^S are weaker than the assumptions of p^F.

Since the clauses M of the design do not change only the clauses resulting from the two properties p^F and p^S have to be compared. Under the assumptions and conclusions from above the following holds:

$$\begin{aligned} P^F - P^S &= (A^F \cup C^F \cup R^F) - (A^S \cup C^S \cup R^S) \\ &= A^F - A^S \end{aligned}$$

With this result it can be concluded that all conflict clauses can be reused which are *not* a result due to an implication caused by a clause of $A^F - A^S$. In other words we have to identify the conflict clauses which have been deduced exclusively from the intersection of the two consecutive PC problems. This intersection is given by:

$$(M \cup P^F) \cap (M \cup P^S) = M \cup A^S \cup C^F \cup R^F$$

Thus, for each conflict clause of the first run the sequence of clauses which produced that conflict clause have to be determined, since with this information we can exactly identify the source of the conflict in terms of the two properties p^F and p^S. This becomes possible, if we further know which clauses have been produced by the design, the individual expressions in the assume part and the individual expressions of the proof part of both properties. Finally for a conflict clause cl the minimal source information is stored which allows to check if cl was produced by a clause of the design or by an assume part or a proof expression. Altogether it can be decided which conflict clauses of the first run can be reused to speed up the current proof.

4. Experimental Results

All experiments have been carried out in the same system environment on an Athlon XP 2800 with 1 GByte main memory. The following experiments always consist of two steps. First, for a circuit a property with “overly” strong assumptions is proved. This is done with and without our approach to measure the time overhead. Next, we prove the same property but in a more general version, i.e. some of the assumptions of the property have been weakened. In this case we measure the speed-up that can be achieved by reusing conflict clauses.

In a first series of experiments we considered the scalable bus arbiter which has already been used in the Examples 1 and 2. This circuit has been studied frequently in formal hardware verification (see e.g. [4, 6]). The considered properties for the arbiter circuit are mutual exclusion of the outputs of the arbiter and the lowestWins property already described in the Examples 1 and 2. In Table 1 the overhead for our approach is given for different arbiter instances (column *Cells*). In the second column the name of the considered property is shown. The next two columns provide information on the corresponding SAT instance. In column *Result* it is shown whether the property holds or not. Next, the run time needed without and with our approach is given in column *std* and column *reuse*, respectively. The difference between the two given run times is the time needed to store learned information into the data base. As can be seen the overhead is neligable, i.e. less than 1% of the run time for the larger examples.

The achieved improvement of the proposed approach for the arbiter is shown in Table 2. In the weakened variant of the property `mutualexclusion` the assumption that no arbiter cell is waiting is no longer assumed. In case of the property `lowestWins_50` we follow exactly Example 2. The first seven columns give the same information as in Table 1. Because the considered properties have been weakened the resulting number of clauses and literals decreases. However, since for each property

Table 1. Overhead for arbiter

Cells	Property	Clauses	Literals	Result	Time (sec)	
					std	reuse
100	mutualexclusion	240,776	541,742	holds	9.15	9.57
100	lowestWins_50	161,399	363,193	holds	14.15	14.49
200	mutualexclusion	961,576	2,163,542	holds	176.65	177.78
200	lowestWins_50	642,799	1,446,393	holds	588.30	590.45

Table 2. Acceleration for arbiter

Cells	Property	Clauses	Literals	Result	Time (sec)		Reused	Speed-up
					std	reuse	Cl. (%)	
100	mutualexclusion	161,076	362,442	holds	13.26	13.01	20.23	1.0
100	lowestWins_50	161,247	362,839	holds	8.71	4.54	100.00	1.9
200	mutualexclusion	642,176	1,444,942	holds	1078.80	343.77	6.23	3.1
200	lowestWins_50	642,347	1,445,339	holds	656.35	22.70	100.00	28.9

learned information can be found in the data base, conflict clauses can be reused. Thus, column *Reused Cl.* gives the percentage of reused clauses. In the last column the achieved speed-up is shown. As can be seen for the 100 cell arbiter in case of the property `mutualexclusion` no speed-up results. But for the three remaining examples a significant speed-up was reached, i.e. up to nearly a factor of 30.

In a second series of experiments we studied FIFOs of different depth. As a property we prove that the content of a FIFO does not change under the assumption that no write operation is performed. In the initial version of this property it has also been assumed that no read operation is performed. Similar information as for the arbiter examples is provided in Tables 3 and 4, respectively. Also in this case for larger examples a speed-up of more than a factor of 10 can be observed.

5. Conclusions and Future Work

In this paper a new approach to reuse conflict clauses in consecutive PC runs for SAT-based BMC has been proposed. The conflict clauses are stored in a data base. While the overhead for the clause recording is negligible, speed-ups of more than an order of magnitude have been observed in our experiments.

It is focus of current work to extend the experimental studies. Based on the presented technique it is also easily possible to propose an automatic approach for finding weak assumptions at low computational costs.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[2] J. Bormann and C. Spalinger. Formale Verifi kation für Nicht-Formalisten (Formal verifi cation for non-formalists). *Informationstechnik und Technische Informatik*, 43:22–28, 2001.

Table 3. Overhead for FIFO

Size	Property	Clauses	Literals	Result	Time (sec)	
					std	reuse
64	nochange	68,077	156,723	holds	14.82	14.92
128	nochange	156,595	361,173	holds	101.83	102.03

Table 4. Acceleration for FIFO

Size	Property	Clauses	Literals	Result	Time (sec)		Reused	Speed-up
					std	reuse	Cl. (%)	
64	nochange	68,072	156,712	holds	14.80	2.16	100.00	6.9
128	nochange	156,590	361,162	holds	101.72	6.42	100.00	15.8

[3] P. Johannsen and R. Drechsler. Formal verification on register transfer level – utilizing high-level information for hardware verification. In *IFIP Int'l Conf. on VLSI*, pages 127–132, 2001.

[4] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

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

[6] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued ar-automata. In *Design, Automation and Test in Europe*, pages 742–748, 2001.

[7] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME*, pages 58–70, 2001.

[8] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. A cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Design, Automation and Test in Europe*, volume 1, pages 162–167, 2004.