Contradictory Antecedent Debugging in Bounded Model Checking [*]

Daniel Große Robert Wille Ulrich Kühne Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{grosse,rwille,ulrichk,drechsle}@informatik.uni-bremen.de

ABSTRACT

In the context of formal verification *Bounded Model Checking* (BMC) has shown to be very powerful for large industrial designs. BMC is used to check whether a circuit satisfies a temporal property or not. Typically, such a property is formulated as an implication. In the antecedent of the property the verification engineer specifies the assumptions about the design environment and joins the respective expressions by logical AND. However, the overall conjunction may have no solution, i.e. the antecedent is contradictory. Since in this case a property trivially holds this situation has to be avoided. Furthermore, the root cause of a contradictory antecedent has to be identified which is a manual and very time-consuming process.

In this paper we propose a fully automatic approach for presenting all reasons of a contradictory antecedent to the verification engineer, i.e. the approach pinpoints to the sub-expressions in the antecedent that form a contradiction. Hence, our approach reduces the debugging time of a contradictory antecedent significantly.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: [Computer-Aided Design (CAD)]

General Terms

Verification

Keywords

Formal Verification, Bounded Model Checking, PSL, Debugging

1. INTRODUCTION

Model checking [1] is a key verification technique to show whether a design satisfies the specification or not. In the last years especially *Bounded Model Checking* (BMC) [2] has become very successful in industrial practice. BMC reduces the verification problem to a *Boolean Satisfiability* (SAT) problem and then searches for counter-examples in bounded executions. If the resulting SAT instance is satisfiable a counter-example of length k has been found. However, BMC can only show that the design is free of errors for the given property up to the bound k. For proving a property, k has to finally reach the sequential diameter of the underlying *Finite State Machine* (FSM), which is infeasible for large circuits. Therefore, approaches for BMC have been developed which can ensure completeness (see e.g. [3, 4]).

In this paper we use the variant of BMC as proposed in [4] which is characterized as follows: First, only properties over a fixed time interval – usually specified as implications – are allowed. Second, the restriction of the starting state for the unrolled circuit logic to the initial state is replaced by assumptions formulated explicitly in the antecedent of the property. As a result, the BMC problem consists only of a single SAT instance build by synthesizing the property and unrolling the design as many times as the property requires. If this SAT instance is unsatisfiable, the property holds.

In practice, during the specification of a property the verification engineer formulates assumptions about the design environment in the antecedent of the property and joins them by logical AND. A typical example for such an assumption is to disable the reset for several cycles. However, for complex designs and hence non-trivial properties the verification engineer can be faced with the problem of a contradictory antecedent, i.e. the antecedent has no solution. Obviously such a situation has to be detected automatically by the BMC tool, since otherwise the property trivially holds and the consequent would not be checked. Typical scenarios that lead to a contradictory antecedent are, e.g. typos in an expression and/or temporal operator in the antecedent, misinterpretation of/or incorrect specification, bug(s) in the design (whereas the antecedent conforms to the specification), or too strong assumptions about the design environment. The latter case can be often observed in practice, since the verification engineer intentionally specifies strong assumptions to understand a complex design. At the beginning it is much easier to focus on a certain design functionality instead of writing a general property.

A closer inspection of these scenarios reveals that two different kinds of a contradictory antecedent have to be distinguished: (1) the contradictory antecedent is solely caused by one or more conflicts of the antecedent sub-expressions or (2) the contradictory antecedent results from one or more conflicts of the antecedent sub-expressions *and* the design.

For BMC as used here, testing whether the antecedent is contradictory is straightforward. Instead of checking the whole property only the antecedent (for case (1)) or the antecedent including the unrolled circuit logic (for case (2)) is tested for satisfiability. However, in case of a negative answer, i.e. the SAT instance is unsatisfiable and hence we have a contradictory antecedent, the verification engineer has to identify what exactly causes the contradiction. As the debugging of a contradictory antecedent is done manually so far, this is a very time-consuming process.

In this paper we present a fully automatic approach to analyze a contradictory antecedent. The result of the ap-

[*]This work was supported in part by the German Federal Ministry of Education and Research (BMBF) and by Concept Engineering GmbH, Freiburg, Germany within the project Herkules under contract no. 01 M 3082.

proach is the presentation of all reasons for the contradictions in the antecedent. A reason is a conjunction of antecedent sub-expressions that evaluates to zero. In addition a reason is minimal in the sense that removing a sub-expression from the conjunction resolves the contradiction. Overall, the methods helps the verification engineer in debugging since he/she understands what exactly causes the contradiction(s). The approach is based on a reformulation of the antecedent using new free variables such that sub-expressions of the antecedent can be disabled. From the assignments to the free variables the approach derives which sub-expressions are "non-relevant", i.e. never part of any contradiction. For the remaining sub-expressions the logical dependencies of the respective values of the free variables are analyzed which allow to determine all reasons.

The basic idea of the approach has been considered already in the context of constraint-based random simulation for debugging contradictory constraints [5]. Besides the different domain and a pure BDD-based implementation on top of a constraint-solver, the approach in this paper additionally has to determine whether a contradiction occurs in combination with the design or not.

In the literature, a property where the antecedent is contradictory is said to be *vacuously satisfied* [6]. Beer et al. considered an *antecedent failure* – for the first time mentioned in [7] – as motivation to study the more general question: can a model or property contain an error if model checking was executed successfully [6, 8]. Searching for errors in this direction is called *vacuity detection* [9, 10]. Improvements have been investigated in [11, 12]. However, all these approaches only address the *detection* of vacuity which in our work is done by checking if the antecedent is unsatisfiable.

Analyzing contraditions in temporal properties previously has been investigated in [13] where a method is proposed to identify a *Temporal Antecedent Failure* (TAF). The authors consider model checking of temporal implication properties specified as regular expressions. The proposed method computes a position in the regular expression that is a reason for a TAF. However, a position may involve a complex Boolean formula that cannot be further analyzed. Besides that, methods for diagnosing over-constrained problems in the area of constraint satisfaction problems address a similar question (see e.g. [14]). But these approaches do not ensure minimality. In the domain of SAT, the computation of so called unsatisfiable cores (i.e. sub-formulas) is of interest [15, 16]. However, to obtain a *minimal* reason the much more complex problem of a *minimal* unsat core has to be considered [17, 18, 19]. Note that the approach of [17] also uses new free variables, but they are introduced for each clause. In general this would lead to a very time consuming process (see e.g. [20]).

2. BOUNDED MODEL CHECKING

We use the BMC variant as described in [4, 21]. Formally, for a design with the transition relation T_δ, the BMC instance for a property p over the finite interval $[0, c]$ is given by: $\wedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \; \wedge \neg\, p$, where p may depend on the inputs, states, internal signals and outputs of the circuit in the time interval $[0, c]$. This BMC instance can be formulated as a SAT problem by unrolling the circuit for c time frames and generating logic for the property. As the property is negated in the formulation, a satisfying assignment corresponds to a case where the property fails. For the specification of the properties, we use a subset of PSL (*Property Specification Language* [22]). A property has the form of an implication $A \rightarrow C$. A is the antecedent and C is the consequent of the property and both consist of a timed expression. A timed expression is formulated on top of variables that are evaluated at different points in time within the time interval $[0, c]$ of the property. The operators in a timed expression are the typical HDL operators like logic, arithmetic and relational operators. The timing is expressed using the temporal operators *next* and *prev*.

3. DEBUGGING APPROACH

In this section we describe our method for determining the reasons of a contradictory antecedent. A reason is a set of sub-expressions that is sufficient to cause a contradiction. Before the details of the approach are presented, some basic definitions are provided.

3.1 Contradictory Antecedent

Definition 1. *For a given design with the transition relation T_δ and a property $p = A \rightarrow C$ over the finite interval $[0, c]$ let $\wedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \wedge \neg(A \rightarrow C)$ be the corresponding BMC instance. Then, A is a* contradictory antecedent *of the property p, iff*

- *A evaluates to* 0 *(considering all inputs, states and outputs used in p as free variables) or*
- $\bigwedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \wedge A$ *evaluates to* 0 *(i.e. the contradiction(s) are caused by both, the antecedent A and the design).*

Remark 1. *The fact whether the contradiction(s) are caused solely by the antecedent or by the antecedent and the design is important for two reasons: First, in the former case debugging is simpler as will be shown later. Second, the size of the SAT instance is much smaller if the design does not have be unrolled as in the second case.*

3.2 Partitioning and Main Flow

We use a partitioning of the antecedent into several sub-expressions. This partitioning is motivated by the typical form of an antecedent, i.e. the antecedent consists of assumptions that are joined by logical AND. In addition, the chosen partitioning allows the identification of contradictions at low computational costs. A refinement can easily be done by performing our analysis again for the first result. The partitioning of the antecedent of a property is defined as follows.

Definition 2. *Let $p = A \rightarrow C$ be a property with a contradictory antecedent A. Then, the antecedent A is partitioned into n sub-expressions $A_0, A_1, \ldots, A_{n-1}$ such that $A = A_0 \wedge A_1 \wedge \cdots \wedge A_{n-1}$, where the position of the logical AND operators is derived from the antecedent according to the conjunction of different assumptions.*

Based on this partitioning, a *reason* in terms of our approach is a subset of all sub-expressions, that forms a contradiction and hence has to be considered by the verification engineer for debugging. The definition for a reason is given as follows.

Definition 3. *Let $p = A \rightarrow C$ be a property with the contradictory antecedent partitioned into $A = A_0 \wedge A_1 \wedge \cdots \wedge A_{n-1}$. Then a* reason *for the contradiction is a non-empty set $R \subseteq \{A_0, A_1, \ldots, A_{n-1}\}$ such that the conjunction of all sub-expressions $A_j \in R$ (either in combination with the design or not) form a contradiction, i.e.*

$$\bigwedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \wedge \bigwedge_{A_j \in R} A_j \qquad \textit{or} \qquad \bigwedge_{A_j \in R} A_j$$

evaluates to 0*, respectively.*

Additionally all reasons are defined to be minimal*, i.e. removing any sub-expression A_j from R resolves the contradiction.*

Remark 2. *In some cases more than one reason for a contradictory antecedent can occur. If in this case only one conflict is fixed the antecedent is still contradictory. Thus, our approach computes* all *reasons and thereby allows the verification engineer to fully understand the problem.*

```
property MYPROP =
  always(
      (A0)  x == 1 &&
      (A1)  x > 5  &&
      (A2)  y == 0
  ) -> (
      next[1](o) == 1
  );
```
(a)

e_0	e_1	e_2
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

(b)

e_0	e_1	e_2
0	–	–
1	0	–

(c)

Figure 1: Simple example for contradiction analysis

To determine the reasons for a contradiction the algorithm uses a *reformulated antecedent* A'. This is done such that each sub-expression A_j can be disabled by the BMC tool and hence each contradiction can be resolved.

Definition 4. *Let A be a contradictory antecedent. Then A is* reformulated *to A' such that*

1. *for each sub-expression A_j a new free variable e_j (called* enable variable*) is introduced and*
2. *A_j is substituted by the implication $e_j \rightarrow A_j$.*

In this way $A = A_0 \wedge A_1 \wedge \cdots \wedge A_{n-1}$ is reformulated to $A' = (e_0 \rightarrow A_0) \wedge (e_1 \rightarrow A_1) \wedge \cdots \wedge (e_{n-1} \rightarrow A_{n-1})$. For the reformulated antecedent A' the following holds:

1. If e_j is set to 1, then the sub-expression A_j is enabled.
2. If e_j is set to 0, then the sub-expression A_j is disabled because $0 \rightarrow A_j$ evaluates to 1 independently from A_j.

Using this reformulation, the main flow for our extended BMC approach is as follows: First it is checked whether a contradiction occurs. In this case, A is reformulated to A' and the contradiction analysis (as described in the next section) is invoked returning the set $\mathcal{R}$ of all reasons. Afterwards, for each reason it is checked if the respective sub-expressions cause the contradiction solely or in combination with the design. The respective output is given to the verification engineer, i.e. for each reason all contained sub-expressions available via a direct link to the syntax tree of the PSL property (and the information with or without design) is shown. This differentiation helps the verification engineer to decide whether only the antecedent has to be considered for debugging or additionally the design as well. If no antecedent contradiction occurs, the property is checked as usual.

3.3 Analysis of the Contradictory Antecedent

If the antecedent is contradictory, then our approach determines all reasons. This task is performed by Algorithm 1. Before the details of the algorithm are provided, we describe the underlying concepts.

The reformulation of the antecedent from A to A' as described in the previous section allows to enable/disable sub-expressions. The basic idea for the computation of all reasons is as follows: Since the enable variables are free variables, we can obtain an assignment to these variables such that the overall contradiction of the antecedent is resolved. Such an assignment contains information which assumptions cannot occur together. But from a single satisfying assignment we cannot conclude which expressions form a contradiction. This is illustrated in the following example:

Example 1. *Consider the property* MYPROP *depicted in Figure 1(a). One satisfying assignment to all enable variables is $e_0 = 0$, $e_1 = 0$, and $e_2 = 0$ (i.e. disabling all sub-expressions). Obviously this assignment resolves the contradiction. However, from this assignment one cannot conclude the minimal reasons of the contradiction (which is $R = \{A_0, A_1\}$).*

In contrast when *all* assignments to the enable variables are available, then all sub-expressions which are either self-contradictory or never part of a contradiction can easily be identified just by applying one of the following observations:

Algorithm 1: contradictionAnalysis(Reformulated antecedent A', unrolled design T_δ, interval $[0, c]$)

```
Result: Set $\mathcal{R}$ of reasons
1  $\mathcal{A} = \emptyset$ ; // Set of assignments for $e_j$ variables
2  while (find new assignment $a$ for $e_j$ in
   $(\bigwedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \wedge A')$ ) do
3      $\mathcal{A} = \mathcal{A} \cup \{a\}$ ;
4  $\mathcal{R} = \emptyset$ ; // Set of reasons
5  $\mathcal{E} = \emptyset$ ; // Set of enable vars for det. analysis
6  for ($j = 0 \ldots n-1$) do
7      if ($\forall a \in \mathcal{A} : a(e_j) = 0$) then
8          $\mathcal{R} = \mathcal{R} \cup \{\{e_j\}\}$;
9      else if ($\forall a \in \mathcal{A} : a(e_j) =$ don't care) then
10         continue;
11     else
12         $\mathcal{E} = \mathcal{E} \cup \{e_j\}$;
13 foreach ($X \in \mathcal{P}(\mathcal{E})$) do from smallest to the largest
14     if ($\exists X' \in \mathcal{R} : X' \subset X$) then
15         continue;
16     else if (( $\bigvee_{a \in \mathcal{A}} \wedge \bigwedge_{e_j \in X} e_j = 1) \equiv 0$) then
17         $\mathcal{R} = \mathcal{R} \cup \{X\}$;
18 return $\mathcal{R}$;
```

Observation 1. *If e_j is 0 for all solutions, then the respective sub-expression A_j is self-contradictory.*

Observation 2. *If the assignment of e_j is don't care for all solutions (i.e. the value of e_j can be either 0 or 1 in all solutions), then the respective sub-expression A_j is never part of a contradiction of A.*

Example 2. *Again the property* MYPROP *shown in Figure 1(a) is considered. Figure 1(b) gives all solutions with respect to enable variables, while Figure 1(c) shows the symbolic representation of them obtained by using a BDD. As can be seen the enable variable e_2 for the assumption A_2 is always don't care and hence we can conclude that this expression is never part of a contradiction.*

In both cases (A_j is self-contradictory or never part of a contradiction) the respective sub-expressions do not have to be considered any longer. This early classification significantly reduces the number of subsets $X \subseteq \{A_0, \ldots, A_{n-1}\}$ to be checked as reasons.

Thus, all satisfying assignments for the enable variables are computed and checked for one of these observations in Algorithm 1. The first step is done by using an *All Solution SAT* solver (line 2), i.e. once a solution has been found a *blocking clause* [23] is added to exclude the same solution for the enable variables from the remaining search space and the search for another solution continues. Each new solution of the enable variables is stored in set $\mathcal{A}$ (line 3).

After this, for each sub-expression A_j it can be checked if A_j is either self-contradictory (line 7) or never part of a reason (line 9). In the former case (e_j is 0 for all solutions and thus A_j is self-contradictory) the enable variable e_j is added as a single reason to a set $\mathcal{R}$ storing all reasons for the contradiction (line 8). Note that $\mathcal{R}$ stores the reasons in terms of e_j variables, not in terms of the respective sub-expressions A_j itself. If the second observation holds (e_j is don't care for all solutions and thus A_j is never part of a contradiction), then this sub-expression can be skipped (line 10). For all remaining cases (line 11) e_j is stored in a set $\mathcal{E}$ including all sub-expressions (in terms of enable variables) which cannot be classified by the two observations and thus have to be considered in the detailed analysis (line 12).

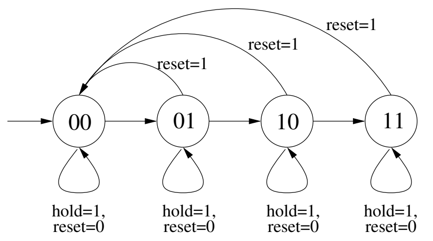

Figure 2: FSM

```
1  property P1 = always(
2   (A0) reset == 1 &&
3   (A1) next( curr_state == "10" ) &&
4   (A2) next_a[0..4]( reset == 0 ) &&
5   (A3) next_a[0..4]( hold == 0 )
6   ) -> ( next[5]( curr_state == "00" ) );
```

Figure 3: PSL property

```
1  property P2 = always(
2   (A0) reset == 1 &&
3   (A1) next_a[1..5](reset == 0) &&
4   (A2) next_a[0..5](!(hold && next(hold))) &&
5   (A3) next[5](curr_state == "01")
6   ) -> ( next[6](curr_state == "01") ||
7          next[6](curr_state == "10") );
```

Figure 4: PSL property

The detailed analysis checks subsets consisting of remaining enable variables (i.e. sub-expressions) for being a reason of the contradiction. To this end, the respective subsets X are obtained by creating the power set $\mathcal{P}(\mathcal{E})$ of $\mathcal{E}$ (line 13). Thereby, we start forming the subsets with at least two elements. Furthermore, by ordering the subsets according to their cardinality, the smaller conjunctions of sub-expressions are checked first. In this way, by excluding all supersets of reasons determined so far (line 14), minimality is guaranteed.

For each remaining subset (i.e. for each combination) the conjunction of the respective sub-expressions is tested for a contradiction. Therefore, all variables $e_j \in X$ are assigned to 1 to enable all respective sub-expressions of X. Then, the resulting cube is combined with a disjunction of all solutions $a \in \mathcal{A}$ as shown in line 16. If the cube of enable variables (i.e. the enabling of the respective sub-expressions) leads to a contradiction, then a reason has been found and thus, X is added to $\mathcal{R}$ (line 17). The final result of the algorithm is the set of all minimal reasons.

4. EXPERIMENTAL RESULTS

The presented techniques have been implemented and evaluated during the verification of different designs.

First we illustrate our method for the FSM depicted in Figure 2. There are four states encoded by two bits. The FSM falls back to state 00 when the signal `reset` is set to 1. Otherwise, it steps through the states in increasing order and wraps around to state 00 unless the signal `hold` is set. For this design we consider the property shown in Figure 3. In the antecedent it is assumed that there is a reset at time point 0 (line 2), the state should be 10 at time point 1 (line 3) and there is no reset and no hold during the cycles 0 to 4 (lines 4 and 5 respectively). This antecedent leads leads to a contradiction. For the analysis, the antecedent is split into the four expressions $A_0, \ldots, A_3$ as mentioned above. It is reported that A_3 is irrelevant and that there are two different reasons for the contradiction. The first reason is $R_1 = \{A_0, A_2\}$. This is because the expression $A_2 = \texttt{next\_a}[0..4](reset == 0)$ implies that there is no reset at time point 0 which contradicts expression A_0. The second reason is $R_2 = \{A_0, A_1\}$, because after the reset at time point 0 – as demanded by A_0 – the state will be 00 at time point 1, which contradicts A_1.

In case of the `next_a` statements the analysis can still be refined, as the expression $\texttt{next\_a}[i..j](x)$ can be rewritten as $\texttt{next}[i](x)$ && $\texttt{next}[i+1](x)$ && ... && $\texttt{next}[j](x)$. By splitting up the `next_a` operator the analysis points exactly to the expression related to time point 0 for the first reason.

A more complex property is considered in Figure 4. The antecedent is a conjunction of four expressions $A_0, \ldots, A_3$. It is assumed that there is a reset at time point 0 followed by 5 cycles with no reset (lines 2 and 3). Furthermore there may not be two consecutive cycles with the hold signal activated (line 4). Finally, at time point 5 it is assumed that the FSM is in state 01 (line 5). When checking this property it is reported that its antecedent is contradictory. The single reason that is given by our approach is the conjunction of all four expressions together with the design. This means that removing any of the assumptions would remove the contradiction. In this case it is the combination of the antecedent with the functionality of the design that makes the scenario impossible.

Our contradiction analysis approach has also been used during the formal verification of a RISC CPU. This CPU implements parts of the MIPS instruction set architecture [24]. It is based on a 5-stage pipeline and contains 32 general purpose registers. The overall design has a gate count of approximately 300.000. During the design process BMC has been applied for early debugging of the basic functionality of the CPU. For this purpose relatively restrictive properties are written to check aspects of the design that have recently been implemented. When a contradictory antecedent occurs our method was able to identify the root cause very fast (e.g. for the analysis of the property for the "load word" instruction the result was obtained in less then a CPU minute on a Intel Xeon CPU with 3 GHz and 32 GB of main memory).

5. REFERENCES

[1] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[2] 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.

[3] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Int'l Conf. on Formal Methods in CAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[4] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. 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.

[5] D. Große, R. Wille, R. Siegmund, and R. Drechsler. Contradiction analysis for constraint-based random simulation. In *Forum on Specification and Design Languages*, pages 130–135, 2008.

[6] I. Beer, S. Ben-David, U. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 279–290, 1997.

[7] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Design Automation Conf.*, pages 596–602, 1994.

[8] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[9] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 82–96, 1999.

[10] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Computer Aided Verification*, pages 485–499, 2002.

[11] H. Chockler and O. Strichman. Easier and more informative vacuity checks. In *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, pages 189–198, 2007.

[12] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik. Exploiting resolution proofs to speed up LTL vacuity detection for BMC. In *Int'l Conf. on Formal Methods in CAD*, pages 3–12, 2007.

[13] S. Ben-David, D. Fisman, and S. Ruah. Temporal antecedent failure: Refining vacuity. In *CONCUR*, pages 492–506, 2007.

[14] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence*, pages 276–281, 1993.

[15] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe*, pages 886–891, 2003.

[16] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe*, pages 880–885, 2003.

[17] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Design Automation Conf.*, pages 518–523, 2004.

[18] J. Huang. MUP: A minimal unsatisfiability prover. In *ASP Design Automation Conf.*, pages 432–437, 2005.

[19] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques-Silva, and K. A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. pages 467–474, 2005.

[20] M.H. Liffiton and K.A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Theory and Applications of Satisfiablity Testing Conference*, volume 3569 of *LNCS*, pages 173–186. Berlin: Springer-Verlag, 2005.

[21] M.D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Trans. on CAD*, 27(11):2068–2082, Nov 2008.

[22] *Accellera Property Specification Language Reference Manual, version 1.1.* http://www.pslsugar.org, 2005.

[23] K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. *Computer Aided Verification*, pages 250–264, 2002.

[24] David A. Patterson and John Hennessy. *Computer Organization and Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.