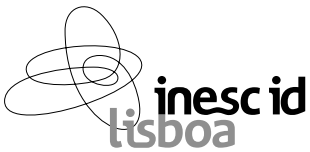
On Computing Minimum Unsatisfiable Cores

I. Lynce and J. Marques-Silva

Technical Report RT/12/2003

October 2003



Software Algorithms and Tools for constraint solving - SAT

Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento, Lisboa

Abstract

Certifying the correctness of a SAT solver is straightforward for satisfiable instances of SAT. Given a computed set of assignments to the problem instance variables, the certification process just consists of verifying that all clauses are satisfied. In contrast, certifying a SAT solver for unsatisfiable instances is a computationally hard problem. Nevertheless, in the utilization of SAT in industrial settings, one often needs to be able to generate unsatisfiability proofs, either to guarantee the correctness of the SAT solver or as part of the utilization of SAT in some applications (e.g. in model checking). As part of the process of generating unsatisfiable proofs, one is also interested in unsatisfiable sub-formulas of the original formula, also known as unsatisfiable cores. This paper overviews recent work on computing unsatisfiable cores, and proposes a solution for computing the unsatisfiable core with the minimum number of clauses.

1 Introduction

The utilization of SAT in industrial settings has motivated work on certifying SAT solvers [10, 5]. Given a problem instance, the certifier needs to be able to verify that the computed truth assignments indeed satisfy a satisfiable instance and that, for an unsatisfiable instance, a proof of unsatisfiability can be generated. Certifying a SAT solver for a satisfiable instance is easy; certifying a SAT solver for an unsatisfiable instance is hard. This paper concerns with the objective of certifying SAT solvers for unsatisfiable instances, and so for generating proofs of unsatisfiability. Besides focusing on generating a proof of unsatisfiability for a target unsatisfiable formula, this paper addresses the problem of identifying a sub-formula that is also unsatisfiable (i.e. an *unsatisfiable core*), and also of computing the *smallest* sub-formula that is also unsatisfiable (i.e. the *minimum unsatisfiable core*).

Besides the theoretical interest of computing unsatisfiable cores, or minimum unsatisfiable cores, the recent utilization of SAT technology in Unbounded Model Checking [7, 8] also relies extensively on the ability of SAT solvers for generating proofs of unsatisfiability and for computing unsatisfiable cores. As a result, the utilization of SAT solvers in Model Checking requires their ability for efficiently generating proofs of unsatisfiability and also for computing *small* unsatisfiable cores.

This paper overviews recent work on computing unsatisfiable cores, and proposes a solution for computing the unsatisfiable core with the minimum number of clauses.

The paper is organized as follows. The next section presents the definitions and background for the remainder of the paper. Afterwards, the paper surveys the recent work on computing unsatisfiable cores [5, 10]. This motivates the question of how to compute the minimum unsatisfiable core, and section 4 proposes a first model for solving this problem. Besides the model, the paper also suggests several optimizations that can be applied.

2 Definitions and Background

The standard SAT definitions of clauses, variables and literals are assumed. A formula φ is a conjunction of clauses, a clause ω is a disjunction of literals, and a literal l or $\neg l$ is either a variable or its complement. The set of clauses is denoted by Ω and the set of variables is denoted by X. For the CNF formula φ and for each clause ω we can also use set notation. Hence, $\omega \in \varphi$ means that clause ω is a clause of CNF formula φ, and $l \in \omega$ means that l is a literal of clause ω. Each clause $\omega \in \varphi$ denotes an implicate of the boolean function f_φ associated with φ. Clearly, not all implicates of f_φ are represented as clauses of φ. Finally, for a CNF formula φ, n denotes the number of variables, m the number of clauses, and l the number of literals.

Propositional variables are denoted $x_1, \ldots, x_n$, and can be assigned truth values 0 (or F) or 1 (or T). The truth value assigned to a variable x is denoted by $\nu(x)$. (When clear from context we use $x = \nu_x$, where $\nu_x \in \{0, 1\}$). A clause is said to be *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is *unsatisfied* if at least one clause is unsatisfied. A *truth assignment* for a formula is a set of pairs of variables and their corresponding truth values. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland [2]. The backtrack search algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n problem variables. A *decision level* is associated with each variable selection and assignment. The first variable selection corresponds to decision level 1, and the decision level is incremented by 1 for each new decision assignment. In addition, and for each decision level, the *unit clause rule* [3] is applied. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfied. In this case, the value of the literal and of the associated variable are said to be *implied*.

In chronological backtracking, the search algorithm keeps track of which decision assignments have been toggled. Given an unsatisfied clause (i.e. a *conflict* or a *dead end*) at decision level d, the algorithm checks whether at the current decision level the corresponding decision variable x has already been toggled. If not, the algorithm erases the variable assignments which are implied by the assignment on x, including the assignment on x, assigns the opposite value to x, and marks decision variable x as toggled. In contrast, if the value of x has already been toggled, the search backtracks to decision level $d - 1$.

Recent state-of-the-art SAT solvers utilize different forms of non-chronological backtracking [6, 1, 9, 4], in which each identified conflict is analyzed, its causes identified, and a new clause (*nogood*) recorded to explain and prevent the identified conflicting conditions. Recorded clauses are then used to compute the backtrack point as the *most recent* decision assignment from all the decision

Algorithm 1 Computing Unsatisfiable Core (in [10])

```
COMPUTE_UNSAT_CORE1(ClauseStack S)
(1)  while S is not empty
(2)     Clause C = POP_CLAUSE(S)
(3)     if C is not marked
(4)        continue
(5)     ClauseSet CS = GET_REASONS_FOR_RECORDING_CLAUSE(C)
(6)     MARK_CLAUSE_SET(CS)
```

assignments represented in the recorded clause.

3 Computing Unsatisfiable Cores

It is well-known that a CNF formula is unsatisfiable if it is possible to generate an empty clause by resolution from the original clauses. The set of original clauses involved in the derivation of the empty clause is referred to as the *unsatisfiable core*.

Observe that an unsatisfiable core can be defined as any subset of the original formula that is unsatisfiable. Consequently, there may exist many different unsatisfiable cores, with different number of clauses, for the same problem instance, such that some of these cores are subsets of others. Also, and in the worst case, the unsatisfiable core corresponds exactly to the set of original clauses.

There have recently been proposed two different approaches for computing unsatisfiable cores [10, 5]. These two approaches basically extract unsatisfiable cores based on the conflict analysis procedure [6]. Observe that the SAT solvers used in these two approaches (Chaff [9] and BerkMin [4]) utilize techniques for performing conflict analysis and non-chronological backtracking.

3.1 Two Different Approaches

The first step for computing an unsatisfiable core consists in identifying the clauses (either original or recorded) that were involved in the steps that led to deriving the empty clause, and thus proving unsatisfiability. However, and since the unsatisfiable core must only include original clauses, it is necessary to develop a procedure for producing a trace from the recorded to the original clauses. The two different approaches for computing unsatisfiable cores [10, 5] differ in the algorithm used for producing this trace. Basically, in [10] the information required is recorded *during* the search, whereas in [5] it is recorded *after* the search.

Algorithms 1 and 2 present the pseudo-code for the algorithms proposed in [10] and [5], respectively. For both the algorithms, ClauseStack S represents a stack with the recorded clauses, ordered by creation time (the clause on the top is the most recently recorded clause). Also, it is necessary to consider a marking scheme for the clauses. Initially, only the clauses involved in deriving the empty clause are marked. At the end, the marked original clauses correspond to the unsatisfiable core. For

Algorithm 2 Computing Unsatisfiable Core (in [5])

```
COMPUTE_UNSAT_CORE2(ClauseStack S)
(1)   while S is not empty
(2)     Clause C = POP_CLAUSE(S)
(3)     if C is not marked
(4)       continue
(5)     LiteralSet LS = ASSIGNMENTS_ENCODED_BY(C)
(6)     DELETE_CLAUSE(C)
(7)     ClauseSet CS = MAKE_AND_ANALYZE_CONFLICT(LS)
(8)     if C is empty then return ERROR
(9)                 else MARK_CLAUSE_SET(CS)
```

each iteration in the algorithm, a new set of clauses is marked (MARK_CLAUSE_SET).

Algorithm 1, proposed by Zhang and Malik [10], keeps a file with all the reasons for creating each recorded clause, i.e. with all the clauses involved in the resolution steps utilized for creating a recorded clause. This file is updated during the search for each new recorded clause. For computing the unsatisfiable core, a breath-first traversal over the file is used, allowing to traverse the marked recorded clauses (GET_REASONS_FOR_RECORDING_CLAUSE(C)) in the order as they appear in the clause stack.

Algorithm 2, proposed by Goldberg and Novikov [5], only tries to identify the clauses involved in creating a recorded clause after the search. For this purpose, all literals in the recorded clause C (ASSIGNMENTS_ENCODED_BY(C)) are assigned value 0 and after unit propagation a conflict has to be found (otherwise the SAT solver is incorrect, and therefore the algorithm returns an `ERROR`). This procedure is called MAKE_AND_ANALYZE_CONFLICT. Observe that the recorded clause C has to be deleted in order to reconstruct what happened during the search. By analyzing this conflict, one is able to identify the clauses involved in creating the recorded clause. The rest of the algorithm is similar to Zhang and Malik's algorithm.

3.2 Comparing the Two Approaches

In this section we give experimental results for the algorithms described in the previous section. The experimental results presented in [10, 5] are for different problem instances, gather different experimental data and use different SAT solvers. As a result it is hard to compare the two algorithms. For these reasons, we decided to make a comparative study, using a single SAT algorithm, running the same problem instances and measuring the same experimental data.

Both algorithms were implemented in our SAT solver: CQuest. CQuest is implemented in C++ and includes the most competitive techniques for industrial benchmarks: clause recording and non-chronological backtracking [6], lazy data structures and low-overhead heuristics [9]. As a result, CQuest performance is comparable to the most efficient SAT solvers [9, 4]. For all experimental

Table 1: Computing Unsatisfiable Cores

Bench	#Vars	#Cls	Algorithm 1				Algorithm 2			
			Search	S+Core	%V	%C	Search	S+Core	%V	%C
bench1	108714	314854	3.8	3.8	8	2	4.3	22.4	17	6
bench2	196925	849233	10.9	11.0	9	2	10.0	120.3	8	2
bench3	265445	1144213	22.3	22.5	9	2	20.7	229.4	9	3
bench4	310602	1166891	30.1	30.4	13	3	15.3	91.7	11	3
bench5	333965	1439193	36.2	36.5	11	2	39.7	364.3	10	3
bench6	9365	22996	61,53	63.2	28	29	74.4	188.7	28	31
bench7	186730	755064	67.6	68.6	32	9	77.6	380.0	32	10
bench8	217046	870680	613,4	620.4	43	14	607.5	1687.5	41	14

results a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used.

Table 1 gives results for a set of industrial benchmarks from the formal verification domain [1]. For each instance, the table presents the number of variables (#Vars) and clauses (#Cls), the CPU time (in seconds) for performing the search (Search), the total CPU time (the search time *plus* the time for computing the unsatisfiable core, S+Core) and the percentage of variables (%V) and clauses (%C) in the unsatisfiable core. Moreover, data for computing the unsatisfiable cores is given for the two approaches being compared.

From the results shown, the following conclusions can be drawn:

1. Zhang and Malik's algorithm for computing unsatisfiable cores is much more effective than Goldberg and Novikov's algorithm. This means that the overhead of maintaining a file with the clauses used for deriving each recorded clause is lower than the overhead of forcing the repetition of many search conflicts after the search.

2. Both the percentage of variables and clauses involved in the unsatisfiable have small values. (These values are quite similar for the two approaches.) This result is indeed promising for the utilization of unsatisfiable cores in model checking or other industrial domains.

4 Computing the Minimum Unsatisfiable Core

The algorithms described in the previous section do not offer any guarantees of optimality regarding the size of the computed unsatisfiable core. The work in [10] proposes an iterative solution for computing a *minimal* unsatisfiable core, by iteratively invoking the SAT solver on each computed sub-formula. This solution, albeit capable of reducing the size of computed unsatisfiable cores, does

[1] These instances have industrial origin and are not publicly available.

not provide any guarantees regarding the size of the unsatisfiable core with the least number of clauses. The objective of this section is to propose a first solution to this problem.

4.1 Minimum Unsatisfiable Cores

We assume that each formula φ is defined over n variables, $X = \{x_1, \ldots, x_n\}$, and that the formula has m clauses, $\Omega = \{\omega_1, \ldots, \omega_m\}$. We start by defining a set S of new variables, $\{s_1, \ldots, s_m\}$, and create a new formula φ' defined on $n+m$ variables, $X \cup S$, with m clauses, where each clause $\omega'_i \in \varphi'$ is defined from a corresponding clause $\omega_i \in \varphi$, and from a variable s_i as follows:

$$\omega'_i = \{\neg s_i\} \cup \omega_i$$

Observe that the variables s_i can be interpreted as *clause selectors* which allow considering or not each clause ω_i. Clearly, φ' is readily satisfiable by setting all s_i variables to 0. Now, for each assignment to the S variables, the resulting sub-formula may be satisfiable or unsatisfiable. For each unsatisfiable sub-formula, the number of S variables assigned value 1 indicate how many clauses are contained in the unsatisfiable core (since the other clauses are satisfied by the S variables assigned value 0). The *minimum unsatisfiable core* is obtained for the unsatisfiable sub-formula with the *least* number of S variables assigned value 1.

4.2 Computing Minimum Unsatisfiable Cores with SAT

One can adapt a state-of-the-art SAT solver to implement the proposed model. The problem instance variables are organized into two disjoint sets: the S variables and the X variables. Decisions are first made on the S variables and afterwards on the X variables; hence each assignment to the S variables defines a potential core. If for a given assignment all clauses become satisfied, then the search simply backtracks to the most recently untoggled S variable. Otherwise, each time the search backtracks from a decision level associated with an X variable to a decision level associated with a S variable, we have identified an unsatisfiable core, defined by the S variables assigned value 1. After all assignments to the S variables have been (implicitly) evaluated, the unsatisfiable core with the least number of utilized clauses corresponds to the minimum unsatisfiable core.

4.3 Challenges & Optimizations

The key challenge of the proposed model is the search space. For the original problem instance the search space is 2^n, whereas for the transformed problem instance the search space becomes 2^{n+m}, where m is the number of clauses. Nevertheless, a few optimizations can be applied.

First, the SAT-based algorithm can start with an upper bound on the size of the minimum unsatisfiable core. For this purpose, one of the algorithms of the previous section can be used [10, 5]. Hence, when searching for the minimum unsatisfiable core, we just need to consider assignments to the S variables which yield smaller unsatisfiable cores. This additional constraint can be modeled with

a cardinality constraint. In addition, each computed unsatisfiable core can be used for backtracking non-chronologically on the S variables, thus further potentially reducing the search space.

For the final version of the paper, preliminary results of the proposed model will be presented and analyzed.

5 Conclusions

This paper overviews algorithms for computing unsatisfiable cores, analyzes the actual practical performance of these algorithms, and proposes a model for computing the *minimum* unsatisfiable core. The proposed model represents a complex optimization problem, and a SAT-based algorithm has been proposed. Future research work entails the experimental evaluation of the proposed model and algorithm, both for computing minimum-size unsatisfiable cores and for evaluating the actual merit of existing algorithms for computing *minimal* unsatisfiable cores.

References

[1] R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, July 1997.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.

[4] E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.

[5] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of the Design and Test in Europe Conference*, pages 10886–10891, March 2003.

[6] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[7] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of Computer Aided Verification*, 2003.

[8] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems*, April 2003.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

[10] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Design and Test in Europe Conference*, pages 10880–10885, March 2003.