Finding Unsatisfiable Subformulas with Stochastic Method*

Jianmin Zhang, Shengyu Shen, and Sikun Li

School of Computer Science, National University of Defense Technology
410073 Changsha, China
{jmzhang,syshen,skli}@nudt.edu.cn

Abstract. Explaining the causes of infeasibility of Boolean formulas has many practical applications in various fields. A small unsatisfiable subformula provides a succinct explanation of infeasibility and is valuable for applications. In recent years the problem of finding unsatisfiable subformulas has been addressed frequently by research works, which are mostly based on the SAT solvers with DPLL backtrack-search algorithm. However little attention has been concentrated on extraction of unsatisfiable subformulas using stochastic methods. In this paper, we propose a resolution-based stochastic local search algorithm to derive unsatisfiable subformulas. This approach directly constructs the resolution sequences for proving unsatisfiability with a local search procedure, and then extracts small unsatisfiable subformulas from the refutation traces. We report and analyze the experimental results on benchmarks.

Key words: Unsatisfiable subformula, Stochastic method, Local search, Resolution sequence, Refutation trace

1 Introduction

Many real-world problems, arising in artificial intelligence, formal verification and electronic design, can be formulated as constraint satisfaction problems, which are translated into Boolean formulas in conjunctive normal form (CNF). Boolean satisfiability (SAT) solvers are generally able to determine whether a large formula is satisfiable or not. We are usually interested in a small explanation of infeasibility that excludes irrelevant information. Therefore when a formula is unsatisfiable, it is often required to find an unsatisfiable subformula, that is, a small unsatisfiable subset of the original formula. Localizing an unsatisfiable subformula is necessary to determine the underlying reasons for the failure. Explaining the causes of unsatisfiability of Boolean formulas is an essential requirement in many fields. A paradigmatic example is repairing inconsistent knowledge from a knowledge base [1]. Additional examples include SAT-based model checking on predicate abstraction [2], counterexample minimization and explanation [3], and fixing wire routing in FPGAs [4].

* This work is supported by the National Natural Science Foundation of China under grant No. 60603088.

There have been many different contributions to research on unsatisfiable subformulas extraction in the last few years, owing to the increasing importance in numerous practical applications. Experimental works on computing unsatisfiable subformulas can be grouped into complete search algorithms and incomplete search algorithms. Most of previous works are complete search approaches [5–13], essentially on the basis of enhanced versions of the Davis-Putnam-Logemann-Loveland (DPLL) backtrack-search algorithm. In the recent past, a few researches have considered the problem of finding the unsatisfiable subformulas by incomplete methods. In [14], the authors present an algorithm which tracks minimal unsatisfiable subformulas according to the trace of a failed local search run for consistency checking. However, this method is essentially based on a typical local search procedure for giving the formula a satisfiable interpretation. Two distinct algorithms proposed in [15] are the first known works on using stochastic method for proving unsatisfiability of a formula. Whereas up till now, no existing research work has concentrated on the unsatisfiable subformulas extraction from the proof of infeasibility utilizing a stochastic local search procedure.

In this paper, we tackle the problem of extracting unsatisfiable subformulas from refutation traces of Boolean formulas by a stochastic local search algorithm. This approach is the first work we are aware of to adopt resolution-based local search method to find unsatisfiable subformulas. Firstly, a local search procedure is employed to compute the resolution sequences for proving unsatisfiability of a formula. The process of resolving an empty clause is combined with some reasoning heuristics, such as unit clause propagation, binary clause resolution and equality reduction. Then each refutation trace is constructed as a tree, and an effective technique called refutation trace pruning is applied to the tree on-the-fly to reduce the search space. Finally, a recursive function is used to find all of the leaves which correspond to the original clauses, and then a small unsatisfiable subformula is obtained, because the original clauses involved in the derivation of an empty clause are referred to as the unsatisfiable subformula.

The paper is organized as follows. The next section introduces the basic definitions used throughout the paper. Section 3 proposes the local search algorithm for finding small unsatisfiable subformulas. Section 4 presents some heuristics and technique to improve the efficiency of our algorithm. Section 5 shows and analyzes experimental results on well-known pigeon hole problem instances. Finally, Section 6 concludes the paper and outlines future research work.

2 Preliminaries

Resolution is a proof system for CNF formulas with the following inference rule:

$$\frac{(A \vee x)(B \vee \neg x)}{(A \vee B)}\ , \tag{1}$$

where A and B denote the disjunctions of literals. The clauses $(A \vee x)$ and $(B \vee \neg x)$ are the resolving clauses, and $(A \vee B)$ is the resolvent. The resolvent of the clauses (x) and $(\neg x)$ is the empty clause $(\perp)$. Every application of the inference

rule is called a resolution step. A resolution sequence, namely a sequence of resolution steps, is that each one uses the result of the previous step or the clauses of the original formula as the resolving clauses of the current step.

Lemma 1. *A CNF formula φ is unsatisfiable iff there exists a finite sequence of resolution steps ending with the empty clause.*

It is well-known that a Boolean formula in CNF is unsatisfiable if it is possible to generate an empty clause by a resolution sequence from the original clauses. A refutation trace of an unsatisfiable formula is defined as a resolution sequence in which the final resolvent is an empty clause.

Definition 1. ***(Unsatisfiable Subformula).*** *Given a formula φ, ψ is an unsatisfiable subformula for φ iff ψ is an unsatisfiable formula and $\psi \subseteq \varphi$.*

It is obvious that there may exist many different unsatisfiable subformulas with different numbers of clauses for the same problem instance, such that some of these subformulas are the subsets of others.

Lemma 2. *The set of original clauses involved in the derivation of an empty clause is referred to as the unsatisfiable subformula.*

That is to say, the clauses, contained in the intersection of a refutation trace and the original formula, belong to some unsatisfiable subformula. Then we illustrate the process of extracting unsatisfiable subformulas from a Boolean formula according to Lemma 1 and Lemma 2. For example, the CNF formula is

$$\varphi = (x_1) \wedge (\neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3) \ . \tag{2}$$

The formula is refuted by a series of resolution steps ending with an empty clause. There are two resolution sequences to prove the infeasibility of this formula. One of the refutation traces is

$$\frac{(x_1)(\neg x_1 \vee x_2)}{(x_2)} \longrightarrow \frac{(x_2)(\neg x_2)}{(\bot)} \ . \tag{3}$$

From the sequence, the resolvent (x_2) of the first resolution step serves as one of the resolving clauses of the second step, and the result of the second resolution step is an empty clause. According to Lemma 1, this formula is unsatisfiable. Therefore, the original clauses included in the proof of infeasibility belong to an unsatisfiable subformula. More specifically, the unsatisfiable subformula corresponding to the above resolution sequence is

$$\psi_1 = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2) \ . \tag{4}$$

Moreover, the other refutation trace is

$$\frac{(x_1)(\neg x_1 \vee x_2)}{(x_2)} \longrightarrow \frac{(x_2)(\neg x_2 \vee x_3)}{(x_3)} \longrightarrow \frac{(x_3)(\neg x_3)}{(\bot)} \ . \tag{5}$$

This finite sequence of resolution steps arrives at an empty clause, and the unsatisfiable subformula consists of the original clauses involved in the process of refutation. Consequently, another unsatisfiable subformula is

$$\psi_2 = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3) \ . \tag{6}$$

In conclusion, this simple example demonstrates that our stochastic local search algorithm to find the small unsatisfiable subformulas is essentially based on Lemma 1 and Lemma 2.

3 Local Search for Finding Unsatisfiable Subformulas

In recent years, the complete methods have made great progress in solving many real life problems including constraint satisfaction problem, but they usually cannot scale well owing to the extreme size of the search space. One way to solve the combinatorial explosion problem is to sacrifice completeness, thus some of the best known methods using this incomplete strategy are stochastic local search algorithms. In general, the local search strategy starts from an initial solution, which may be randomly or heuristically generated. Then the search moves to a better neighbor according to the objective function, and terminates if the goal is achieved or no better solution can be found. Stochastic local search methods are underlying some of the best-performing algorithms for certain types of problem instances, both from an empirical as well as from a theoretical point of view. Consequently, this stochastic strategy is adopted to tackle the problem of finding unsatisfiable subformulas. We propose a resolution-based local search algorithm based on Lemma 1 and Lemma 2. The algorithm, detailed in the later, is given as follows:

```
SLSAtoFindUS (formula)
   refuted = false
   iteration = 0
   while ((iteration < MAXITER) && !refuted) do
      if (Unit_Clause_Propagation() return UNSAT) then
         refuted = true
      else if (there exist binary clauses) then
         Binary_Clause_Resolution()
         Non_Tautology()
         Equality_Reduction()
         No_Same_Clause()
      else  Randomly choose two clauses to resolve
      for (each clause c1 added into sequence)
         Trace_Updating(c1)
      if (formula.size > MAXSIZE) then
         Remove a clause c1 at random
         Trace_Pruning(c1)
      iteration++
```

```
if (refuted == true) then
    print unsatisfiable
    SmallUS = Compute_Unsatisfiable_Subformula(sequence)
else  print unresolved
return SmallUS
```

The stochastic local search algorithm begins with an input formula in CNF format. The objective function of this algorithm is to derive an empty clause, and a necessary condition for this to occur is that the formula contains at least some short clauses. We perform resolution of two clauses heuristically or randomly, until the formula is refuted or the upper limit of iterations is reached. In this algorithm, the function called unit clause propagation can determine whether the formula is infeasible, because the formula is refuted if and only if an empty clause can be resolved by two unit clauses. If the current formula contains binary clauses, some reasoning strategies are employed in this algorithm, such as binary clause resolution and equality reduction. The function named Non_Tautology deletes the clauses which contain two opposite literals such as $(x_1 \vee \neg x_1 \vee x_3)$. The function of No_Same_Clause is to remove the duplicate clauses from the formula. If there is no binary clause in the formula, two clauses will be randomly chosen to resolve according to the inference rule shown in Equation 1, and then the resolvent is added into the formula. However, too many resolving clauses increase the overhead of the search process, thus a clause deletion scheme called refutation trace pruning is employed. When the updated formula exceeds the maximum size constant, a clause is removed at random, and some redundant clauses on the source trace of this clause are also deleted.

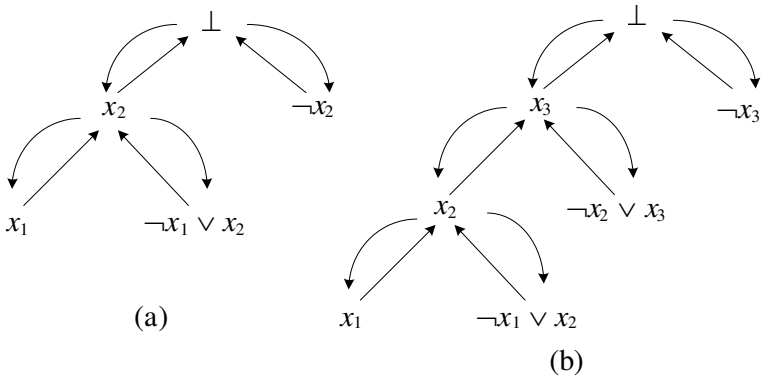

Fig. 1. The Process of Finding Unsatisfiable Subformulas

When the algorithm proceeds, we record the sequences of clauses engaged in the process of resolving an empty clause. Then a tree is constructed with respect to each refutation trace. If the formula is refuted, a recursive function

called Compute_Unsatisfiable_Subformula is employed to extract a small unsatisfiable subformula from the formation of a treelike arrangement. According to Lemma 2, we can conclude that all leaf nodes of a tree are actually referred to as the unsatisfiable subformula. Fig. 1 illustrates the process of deriving small unsatisfiable subformulas from the formula denoted by Equation 2. As shown in Fig. 1, there are two trees respectively corresponding to two refutation traces, which are represented as Equation 3 and Equation 5. The original clauses located on the leaves of a tree can be extracted by a recursive algorithm to form the unsatisfiable subformula. For example, in Fig. 1 (a), the root node, namely an empty clause, is resolved by an interim result (x_2) and a leaf node $(\neg x_2)$. If we treat the clause (x_2) as a root node, the branches and leaves with the root also constitute a tree, and then the recursive function can be used on this subtree. The clause (x_2) is resolved by two leaf nodes (x_1) and $(\neg x_1 \vee x_2)$. Consequently, an unsatisfiable subformula is composed of the three leaf clauses belonging to the original formula. Similarly, in Fig. 1 (b), another unsatisfiable subformula consists of the four leaf clauses $(\neg x_3)$, $(\neg x_2 \vee x_3)$, (x_1), and $(\neg x_1 \vee x_2)$.

4 Heuristics and Pruning Technique

To improve the efficiency of the local search algorithm, we implement some reasoning heuristics. One of the heuristics is unit clause propagation. A so-called unit clause is the clause only containing one literal. Unit clause propagation selects a unit clause from the original formula, and then performs the reduction on the formula by this unit clause. We achieve this reduction in two kinds of situation: Firstly, if some clause contains a literal which is negative of the literal in the unit clause, the corresponding literal is deleted from that clause; Secondly, we eliminate the clauses which include the literal of the unit clause. Considered the formula shown in Equation 2, the clause $(\neg x_2)$ is a unit clause, and is propagated to the whole formula. In accordance with the reduction rule of unit clause propagation, the literal (x_2) is removed from the third clause $(\neg x_1 \vee x_2)$, and the fourth clause $(\neg x_2 \vee x_3)$ is deleted. Consequently, the formula is turned into

$$\varphi' = (x_1) \wedge (\neg x_2) \wedge (\neg x_1) \wedge (\neg x_3) \ . \tag{7}$$

After applying the unit clause propagation, one can observe that the formula is strongly simplified and easily refuted. Furthermore, because unit clause propagation might generate new unit clauses, it is an iterative process of executing reductions by unit clauses until either of the following conditions is reached: one is that an empty clause is resolved, or the other is that there are no more unit clauses in the remain formula. The order in which the unit clause reductions occur is not important to the correctness of our local search algorithm.

In general, a Boolean formula might also have many binary clauses, which are defined as the clauses including two literals. Then it is possible to do a lot of reductions on the original formula by reasoning with these binary clauses as well. The resolution of two binary clauses arises if and only if they contain one

pair of opposite literals, and abides by the inference rule depicted in Equation 1. For instance, a Boolean formula in CNF is given as follows:

$$\phi_1 = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee x_3) \ . \tag{8}$$

There are three binary clauses, which can be resolved by the inference rule, in this formula. Then the process of resolution between the binary clauses is

$$\frac{(\neg x_1 \vee x_2)(\neg x_2 \vee x_3)}{(\neg x_1 \vee x_3)}, \frac{(\neg x_1 \vee x_2)(x_1 \vee x_3)}{(x_2 \vee x_3)}, \frac{(\neg x_1 \vee x_3)(x_1 \vee x_3)}{(x_3)} \ . \tag{9}$$

Resolving these clauses produces two new binary clauses $(\neg x_1 \vee x_3)$, $(x_2 \vee x_3)$ and one new unit clause (x_3). More generally, performing all possible resolutions of pairs of binary clauses may generate new binary clauses or new unit clauses. Therefore, binary clause resolution can be done in conjunction with unit clause propagation in a repeated procedure.

The third heuristic is equality reduction, which is also a type of useful binary clause reasoning mechanism. Equality reduction is essentially based on the following equation:

$$(x \Leftrightarrow y) = (x \Rightarrow y) \wedge (y \Rightarrow x) = (\neg x \vee y) \wedge (\neg y \vee x) \ . \tag{10}$$

If a formula contains two correlated clauses such as $(x \vee \neg y)$ and $(\neg x \vee y)$, we can form an updated formula by equality reduction. Equality reduction is a three-step procedure: Firstly, all instances of y in the formula are replaced by the literal x or vice versa; Secondly, all clauses containing both x and $\neg x$ are deleted; Finally, all duplicate instances of x or $\neg x$ are removed from all of the clauses. For example, a Boolean formula in CNF is

$$\phi_2 = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3) \ . \tag{11}$$

Obviously, one can conclude that x_1 is equivalent to x_2, because there exist $(x_1 \vee \neg x_2)$ and $(\neg x_1 \vee x_2)$. We substitute x_1 for x_2 throughout the formula, and perform reductions on the new clauses. Then the reduced formula is obtained:

$$\phi_2' = (x_1 \vee x_3) \wedge (\neg x_1 \vee x_3) \ . \tag{12}$$

Similar to binary clause resolution, such clause reasoning approach might yield new binary clauses. Consequently, equality reduction combined with unit clause propagation and binary clause resolution can run iteratively, until an empty clause is resolved or no new clause is added.

During the process of derivation, many redundant clauses bring a degradation of runtime performance and memory consumption. To reduce the search space, we propose a technique called refutation trace pruning, which filters out the clauses not belonging to any refutation proof of the formula. We keep two fields for each interim clause: one is the list of resolving source trace of this

clause (*clause.tarce*), the other is a counter that tracks the number of descendants of this clause which still have a chance to involve in the refutation proof (*clause.offspring_count*). This technique contains two functions: one is to establish or update the two fields of trace information when a new clause is added into the sequence, the other is to remove the clauses which are redundant for proof of unsatisfiability. Firstly, the function called Trace_Updating is introduced:

```
Trace_Updating(c)
    c.trace = resolution_clauses
    c.offspring_count = 0
    for (each clause c1 in c.trace)
        c1.offspring_count++
```

While a clause c is created its *offspring_count* is zero. A newly generated clause can potentially take part in the proof, thus the *offspring_count* of each clause on its resolution trace is incremented. Another function called Trace_Pruning is presented as follows:

```
Trace_Pruning(c)
    if ((c.offspring_count == 0) && (c.trace is not empty))
        for (each clause c1 in c.trace)
            c1.offspring_count--
            Trace_Pruning(c1)
        delete c.trace
```

When a clause c is removed and $c.offpsring\_count \geq 1$, we keep *c.trace* because we cannot know whether a descendant of c is included in the proof or not. If c has no descendant, the *c.trace* is deleted and the *offspring_count* for each clause on its resolution source is decremented. These counters may become zero, so a recursive call to the function of Trace_Pruning tries to remove each of the resolution sources.

5 Experimental Results

To experimentally evaluate the effectiveness of our algorithm, we select 9 problem instances from the well-known pigeon hole family, and compare our algorithm with the greedy genetic algorithm [11] on this benchmark. The pigeon hole problem "hole$n$" asks whether it is possible to place $n+1$ pigeons in n holes without two pigeons being in the same hole. We choose these instances because each of them has only one unsatisfiable subformula. Consequently, the greedy genetic algorithm which derives a minimum unsatisfiable subformula and our algorithm can obtain the same unsatisfiable subformula for one problem instance.

Our algorithm to find small unsatisfiable subformulas is implemented in C++ using STL. The experiments were conducted on a 1.6 GHz Athlon machine having 1 GB memory and running the Linux operating system. The limit time was 3600 seconds. The experimental results are listed in Table 1. Table 1 shows the

number of clauses (clas) and the number of variables (vars) for each of the 9 problem instances. Table 1 also gives the number of clauses in the unique unsatisfiable subformula (US size) for every instance. Furthermore, Table 1 provides the runtime in seconds of the greedy genetic algorithm (GGA time) to extract the unsatisfiable subformula. The last column presents the mean runtime of ten launches in seconds for the stochastic local search algorithm (SLSA time).

Table 1. Performance Results on Benchmark

Instances	**clas**	**vars**	**US size**	**GGA time**	**SLSA time**
hole2	9	6	9	0	0
hole3	22	12	22	0	0
hole4	45	20	45	0	0
hole5	81	30	81	0.02	0
hole6	133	42	133	0.08	0.1
hole7	204	56	204	0.90	0.5
hole8	297	72	297	51.90	22.8
hole9	415	90	415	1304.00	682.6
hole10	561	110	561	time out	1850.0

From this table, we may observe the following. The stochastic local search algorithm outperforms the greedy genetic algorithm for most formulas, except for the instance of *hole6*. Our algorithm is able to successfully find the unsatisfiable subformula at each launch. For the instance of *hole10*, the greedy genetic algorithm failed to extract the unsatisfiable subformula within the timeout, but our algorithm succeeded in obtaining it. The resolution-based local search algorithm can efficiently solve these instances, partially because the heuristics brings the great capabilities of reasoning short clauses, such as unit clause propagation, binary clause resolution and equality reduction, and the pigeon hole problem instances contain many binary clauses.

6 Conclusion

Finding the unsatisfiable subformulas of problem instances has practical applications in many fields. In this paper, we present a stochastic local search algorithm to derive small unsatisfiable subformulas from CNF formulas. The algorithm is combined with some reasoning heuristics and pruning technique. The experimental results illustrate that our algorithm outperforms the greedy genetic algorithm. The results also show that this local search algorithm can efficiently tackle the certain type of problem instances with many short clauses, and cannot work very well for the formulas with most long clauses, largely because it makes the decisions on resolution of two long clauses in a stochastic way, and lacks of

the effective heuristics for selecting the right clauses. Therefore one of the future works is to add more aggressive methods for resolution of long clauses.

References

1. Mazure, B., Sais, L., Gregoire, E.: Boosting complete techniques thanks to local search methods. Annals of Mathematics and Artificial Intelligence 22(3–4) 319–331 (1998)
2. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
3. Shen, S., Qin, Y., Li, S.: Minimizing counterexample with unit core extraction and incremental SAT. In: Cousot, R. (eds.) VMCAI 2005. LNCS, vol. 3385, pp. 298–312. Springer, Heidelberg (2005)
4. Nam, G-J., Aloul, F., Sakallah, K., Rutenbar, R.: A comparative study of two Boolean formulations of FPGA detailed routing constraints. In: Proceedings of the 2001 International Symposium on Physical Design (ISPD'01), pp. 222–227 (2001)
5. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, Springer, Heidelberg (2004)
6. Lynce, I., Marques-Silva, J.P.: On computing minimum unsatisfiable cores. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 305–310. Springer, Heidelberg (2005)
7. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: a minimally-unsatisfiable subformula extractor. In: Proceedings of the 41st Design Automation Conference (DAC'04), pp. 518–523 (2004)
8. Liffiton, M.H., Sakallah, K.A.: On finding all minimally unsatisfiable subformulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 173–186. Springer, Heidelberg (2005)
9. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005)
10. Mneimneh, M.N., Lynce, I., Andraus, Z.S., Marques-Silva, J.P., Sakallah, K.A.: A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 393–399. Springer, Heidelberg (2005)
11. Zhang, J., Li, S., Shen, S.: Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In: Sattar, A., Kang, B.H. (eds.) AI 2006, LNCS(LNAI), vol. 4304, pp. 847–856. Springer, Heidelberg (2006)
12. Gershman, R., Koifman, M., Strichman, O.: Deriving small unsatisfiable cores with dominator. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 109–122. Springer, Heidelberg (2006)
13. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
14. Gregoire, E., Mazuer, B., Piette, C.: Tracking MUSes and strict inconsistent covers. In: Proceedings of the the Sixth Conference on Formal Methods in Computer-Aided Design (FMCAD'06), pp. 39–46 (2006)
15. Prestwich, S., Lynce, I.: Local search for unsatisfiability. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 283–296. Springer, Heidelberg (2006)