Tracking Unsatisfiable Subformulas from Reduced Refutation Proof

Jianmin Zhang
Computer School, National University of Defense Technology, ChangSha, China
Email: jmzhang@nudt.edu.cn

Shengyu Shen and Sikun Li
Computer School, National University of Defense Technology, ChangSha, China
Email: {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 finding unsatisfiable subformulas has been addressed frequently by research works, mostly based on the SAT solvers with DPLL backtrack-search algorithm. However little attention has been concentrated on extraction of unsatisfiable subformulas using incomplete methods. In this paper, we present the definitions of refutation proof and refutation parsing graph, and then propose a resolution-based local search algorithm to track unsatisfiable subformulas according to the reduced refutation proof of a formula. This approach directly constructs the resolution sequences for proving unsatisfiability with a local search procedure, and then recursively derives unsatisfiable subformulas from the resolve traces. We report and analyze the experimental results on well-known and randomly generated benchmarks.**

Index Terms—**Boolean satisfiabiltiy, unsatisfiability subformula, refutation proof, local search**

I. INTRODUCTION

Many real-world problems, arising in formal verification, electronic design, equivalence checking and Auto Test Pattern Generation (ATPG), can be formulated as constraint satisfaction problems, which are translated into Boolean formulas in conjunctive normal form (CNF). Boolean satisfiability (SAT) solvers, such as Chaff[1] and MiniSAT[2], are generally able to determine whether a large formula is satisfiable or not. When a formula is unsatisfiable, it is often required to find an unsatisfiable subset of the original formula, because we are interested in a small explanation of infeasibility that excludes irrelevant information. 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 applications. A paradigmatic example is SAT-based model checking on predicate abstraction[3], where analysis of unsatisfiability is an essential step for ensuring completeness of bounded model checking. Additional examples include fixing wire routing in FPGAs[4], counterexample explanation[5], and repairing inconsistent knowledge from a knowledge base[6].

There have been many different contributions to research on unsatisfiable subformulas extraction in the last few years, owing to the increasing importance in practical applications. Experimental works can be grouped into complete search algorithms and incomplete search algorithms. Most of previous works are complete search approaches[7-17], mostly on the basis of the SAT solvers with DPLL backtrack-search algorithm. In the recent past, a few researches have considered the problem of finding the unsatisfiable subformulas by incomplete methods. Gregoire et al.[18] present an algorithm which derives unsatisfiable subformulas from the trace of a failed local search run for consistency checking. However, this approach is essentially based on a typical local search procedure for giving the formula a satisfiable interpretation. Two independent algorithms proposed in [19] and [20] are the first known works on using local search method for proving unsatisfiability of a formula. Whereas to the best of our knowledge, there is no published work in the literature devoted to the unsatisfiable subformulas extraction from the proof of infeasibility utilizing a randomized local search procedure.

In this paper, we propose the definitions of refutation proof and refutation parsing graph, and tackle the problem of extracting unsatisfiable subformulas from refutation proof 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 the empty clause is combined with some Boolean reasoning techniques, such as unit clause propagation, binary clause resolution and equality reduction. While the resolvent is added to the formula, the subsumption elimination procedure is used to guarantee the CNF subsumption-free. Then each resolve trace is constructed as a refutation parsing graph, and an effective method called refutation proof pruning is applied to the graph 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 an unsatisfiable subformula is obtained, because the original clauses involved in the derivation of the empty clause are referred to as the unsatisfiable subformula. Finally, we report and analyze the experimental results on well-known pigeon hole and FPGA routing benchmarks, and randomly generated 2-SAT and 3-SAT problem instances.

The paper is organized as follows. The next section introduces the basic definitions and theorems used throughout the paper. Section 3 proposes the local search algorithm for finding unsatisfiable subformulas. Section 4 presents some reasoning techniques to improve the efficiency of our algorithm. Section 5 provides the depiction of the subsumption elimination procedure. Section 6 describes the refutation proof pruning technique. Section 7 shows and analyzes experimental results on well-known and randomly generated problem instances. Finally, Section 8 concludes the paper and outlines future research work

II. Related Work

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. In [7], a method of adaptive core search guided by clauses hardness is employed to extract small unsatisfiable subformulas. zCore[8] is an algorithm for deriving small unsatisfiable subformulas based on the ability of DPLL-based SAT solvers to produce resolution refutations. In [9], an algorithm of enumerating all possible subsets is suggested to compute a minimum unsatisfiable subformula. Another approach is called AMUSE[10], in which selector variables are added to each clause and the unsatisfiable subformula is derived by a branch-and-bound algorithm on the updated formula. A different algorithm existing in the current literature that guarantees minimality is MUP[11]. MUP is mainly a prover of minimal unsatisfiability, as opposed to an unsatisfiable subformula extractor.

Some research works, based on a strong relationship between maximal satisfiability and minimal unsatisfiability, have developed some sound techniques for finding all minimal unsatisfiable subformulas[12] or a minimum unsatisfiable subformula[13,14]. CoreTimmer[15] iterates over each internal node that consumes a large number of clauses and attempts to prove them without these clauses. A scalable algorithm[16], adopting a deeper exploration of resolution refutation, is proposed for minimal unsatisfiable subformulas extraction. A novel algorithm[17] to find minimal unsatisfiable subformula is based on Brouwer's fixed point approximation theorem to satisfiability. In [18], the authors present an algorithm which tracks minimal unsatisfiable subformulas according to the trace of a failed local search run for satisfiability checking.

II. 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 $(\perp)$. Each application of the inference rule is called a resolution step. The above resolution step is represented as $((A \vee x) \wedge (B \vee \neg x)) \models (A \vee B)$.

Definition 1. (Boolean Satisfiability) Given a CNF formula $\varphi(X)$, where X is the set of variables, and a Boolean function $F(X)$: $\{0,1\}^n \rightarrow \{0,1\}$, the Boolean satisfiability problem consists of identifying a set of assignments X' to the variables, such that $F(X')=1$, or proving that no such assignment exists.

Definition 2. (Unsatisfiable Subformula) Given a formula φ, ψ is an unsatisfiable subformula for φ if and only if ψ is an unsatisfiable formula and $\psi \subseteq \varphi$.

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

It is well-known that a formula is unsatisfiable if it is possible to generate the empty clause by resolution from the original clauses. A sequence of resolution steps, each one uses the result of the previous step or the clauses of the original formula as the resolving clauses of the current step, is called a resolution sequence.

Definition 3. (Refutation Proof) Given an unsatisfiable formula φ, a refutation proof R of φ is defined as a resolution sequence in which the final resolvent is the empty clause.

From the definition, it is concluded that a refutation proof contains the explanation of infeasibility of the formula. In other words, the causes of unsatisfiability can be derived from the refutation proofs in the sense that removing them will correct the infeasibility.

Theorem 1. Consider an unsatisfiable formula φ, then φ contains at least one refutation proof R. Given a set S: $S=C(\varphi) \cap C(R)$, where $C(\varphi)$ and $C(R)$ respectively denote the set of clauses in φ and R, then $\psi = \bigwedge_{c \in S} c \models \perp$, and ψ is an unsatisfiable subformula of φ.

Proof. The first conclusion can be directly proved by the definition of refutation proof and Lemma 1. We next try to prove the second conclusion.

The final resolvent of a refutation proof R is the empty clause $\perp$. Suppose that there are n resolution steps in R. Proof by the principle of mathematical induction involving the integral variable n.

When n=1, the formula φ must contain two opposite polarity unit clauses, and then the refutation proof can be denoted as $(x \wedge \neg x) \models \perp$, thus the conclusion is verified.

When $n \leq m$, suppose the conclusion is verified.

When $n=m+1$, there are two kinds of condition:

First condition: the first two steps of the refutation proof R can be represented as $C_1 \wedge C_2 \models C_4$ and $C_3 \wedge C_4 \models C_5$, where $\{C_1,C_2,C_3\}\subseteq\varphi$. Then the resolvent C_4 is replaced by $C_1\wedge C_2$. According to the above supposition, the remain m resolution steps can arrive at the empty clause $\perp$.

Second condition: the first three steps of R can be represented as $C_1\wedge C_2|{=}C_5$, $C_3\wedge C_4|{=}C_6$ and $C_5\wedge C_6|{=}C_7$, where $\{C_1,C_2,C_3,C_4\}\subseteq\varphi$. Then the resolvent C_5 is replaced by $C_1\wedge C_2$, and the resolvent C_6 is replaced by $C_3\wedge C_4$. According to the supposition, the remain $m-1$ resolution steps can arrive at $\perp$.

In conclusion, the original clauses included in a refutation proof constitute an unsatisfiable subformula.

□

Definition 4. (Refutation Parsing Graph, RPG) A refutation parsing graph corresponding to an unsatisfiability proof by resolution, is a directed acyclic graph $G(V,E,s)$ with a single sink node $s\in V$, in which the nodes denote CNF clauses: the leaf nodes represent original clauses, the inner nodes represent clauses derived by resolution, and the sink node represents the empty clause $\perp$. Each node can be inferred from its parent nodes according to a resolution step.

Given a refutation proof R of the Boolean formula φ, we build a refutation parsing graph $G(V,E,s)$ corresponding to R by the following rules: the only sink node represents the empty clause $C_s=\perp$; We suppose there is a resolution step in R: $C_p\wedge C_q|{=}C_r$, then the clauses C_p, C_q and C_r respectively denote three nodes $\{v_p,v_q,v_r\}\subseteq V$, and $e_{pr}\in E$ or $e_{qr}\in E$ represents the directed edge which is from the parent node p or q to the child node r. Complying with these rules, we can create a refutation parsing graph step by step, starting from the empty clause and backtracking to the original clauses.

Theorem 2. Consider a refutation proof R of an unsatisfiable CNF formula φ, and a refutation parsing graph $G(V,E,s)$ for R. Then all of clauses corresponding to the leaf nodes in an RPG compose an unsatisfiable subformula of φ.

Proof. Proof by contradiction. Suppose that there is a clause $C\notin\varphi$ which corresponds to a leaf node $v\in V$.

According to Definition 3, there exist two clauses C_1 and C_2 in φ, which satisfy $C_1\wedge C_2\,|{=}C$.

Then according to Definition 4, there must be two nodes v_1 and v_2, which respectively correspond to C_1 and C_2. That is to say, the node v for the clause C is not a leaf node.

However, the above supposition says that $v\in V$ is a leaf node. Therefore, it results in a contradiction, and the supposition is false.

That is, all of the clauses corresponding to the leaf nodes belong to the original formula φ. Then from Theorem 1, we can draw the conclusion that all of the leaf clauses compose an unsatisfiable subformula of φ.

□

From the theorems, the set of original clauses involved in the derivation of the empty clause is referred to as the unsatisfiable subformula. In other words, the clauses, contained in the intersection of a refutation proof and the original formula, constitute an unsatisfiable subformula. Then we illustrate the process of extracting unsatisfiable subformulas from a formula according to the theorems. For example, a CNF formula is

$$\varphi=(x_1)\wedge(\neg x_2)\wedge(\neg x_1\vee x_2)\wedge(\neg x_2\vee x_3)\wedge(\neg x_3) \quad (2)$$

The above formula is refuted by a series of resolution steps ending with the empty clause. Two refutation proofs to affirm the infeasibility of the formula φ are shown as follows:

$$R_1=\frac{(x_1)(\neg x_1\vee x_2)}{x_2}\to\frac{(x_2)(\neg x_2)}{(\perp)}, \quad (3)$$

$$R_2=\frac{(x_1)(\neg x_1\vee x_2)}{x_2}\to\frac{(x_2)(\neg x_2\vee x_3)}{(x_3)}\to\frac{(x_3)(\neg x_3)}{(\perp)} \quad (4)$$

From R_1, 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 the empty clause. Similarly, the other sequence of resolution steps also arrives at the empty clause. According to Theorem 1, the original clauses included in the proof of infeasibility belong to the unsatisfiable subformula. More specifically, two unsatisfiable subformulas respectively corresponding to the refutation proofs R_1 and R_2 are

$$\psi_1=(x_1)\wedge(\neg x_1\vee x_2)\wedge(\neg x_2), \quad (5)$$

$$\psi_2=(x_1)\wedge(\neg x_1\vee x_2)\wedge(\neg x_2\vee x_3)\wedge(\neg x_3) \quad (6)$$

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

III. Algorithm Overview

In recent years, the complete methods have made great progress in solving many real life problems including Boolean satisfiability, 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 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. 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 Theorem 1 and Theorem 2.

The pseudo code of the local search algorithm, detailed in the later, is given in Fig.1. The algorithm begins with an input formula in CNF format. The objective function of this algorithm is to derive the 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 either of the following conditions is achieved: one is that the Boolean formula is refuted, or the other is that the upper limit of iterations is reached.

In Fig.1, the function called unit clause propagation can determine whether the formula is unsatisfiable, because the formula is refuted if and only if the 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 including two opposite polarity literals. 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 in accordance with the inference rule shown in Equation 1. When the resolvent is added into the formula, the subsumption elimination procedure is employed to remove the subsumed clauses. However, too many resolving clauses increase the overhead of the search process, thus a clause deletion scheme called refutation proof pruning is proposed. When the updated formula exceeds the maximum size, a clause is chosen and removed at random, and then some redundant clauses on the source trace of this clause are also deleted. The longer a clause is, the greater is the probability that the clause is selected.

```
RbLSA (formula)
1  refuted = false
2  iteration = 0
3  while ((iteration < MAXITER) and !refuted) do
4    if (Unit_Clause_Propagation() return UNSAT)
5      refuted = true
6    else if (there exist binary clauses) then
7      Binary_Clause_Resolution()
8      Non_Tautology()
9      Equality_Reduction()
10     No_Same_Clause()
11   else
12     Randomly choose two clauses to resolve
13   Subsumption_Elimination(resolvent)
14   Trace_Updating(resolvent)
15   if (formula.size > MAXSIZE) then
16     Remove a clause C at random
17     Trace_Pruning(C)
18   iteration++
19 if (refuted == true) then
20   print "unsatisfiable"
21   SmallUS = Compute_US(sequence)
22 else
23   print "unresolved"
24 return SmallUS
```

Figure 1. Overview of the resolution-based local search algorithm.

When the algorithm proceeds, we record the sequences of the clauses engaged in resolving the empty clause. Then a refutation parsing graph is created with respect to each refutation proof. If the formula is refuted, a recursive function, called Compute_US, shown in Fig.2, is employed to track an unsatisfiable subformula from the formation of a treelike arrangement. According to Theorem 2, we can conclude that all leaf nodes of an RPG are actually referred to as the unsatisfiable subformula.

```
Compute US (sequence)
1 C = last element of sequence
2 if (C is not the empty clause) then
3   print error and return
4 else
5   Traverse_graph(sequence.size–1)
6   Traverse_graph(sequence.size–2)
7 return SmallUS
Traverse_graph (ClausePos)
1 C = sequence[ClausePos]
2 if (C ∈ original formula) then
3   Push C into SmallUS
4 else
5   pos = Find position of C in the sequence
6   Traverse_graph(pos–1)
7   Traverse_graph(pos–2)
```

Figure 2. Procedure of tracking the unsatisfiable subformulas from a refutation parsing graph

Fig.3 illustrates the process of deriving unsatisfiable subformulas from the formula denoted by Equation 2. As depicted in Fig.3, there are two refutation parsing graphs corresponding to two refutation proofs, which are respectively represented as Equation 3 and 4. The original clauses located on the leaves of an RPG can be extracted by a recursive algorithm to form the unsatisfiable subformula. For example, in Fig.3(a), the sink node, namely the 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 sink node, the inner and leaf nodes with the sink node also constitute an RPG, and then the recursive function can be applied to this subgraph. The clause (x_2) is resolved by two leaf nodes (x_1) and ($\neg x_1 \vee x_2$). Thus an unsatisfiable subformula is composed of the three leaf clauses belonging to the original formula. Similarly, in Fig.3 (b), the unsatisfiable subformula consists of the four leaf nodes ($\neg x_3$), ($\neg x_2 \vee x_3$), (x_1), and ($\neg x_1 \vee x_2$).

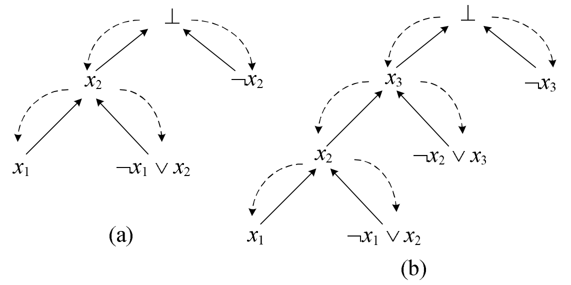


Figure 3. Refutation Parsing Graphs(RPGs) of φ. The solid edges belong to the refutation parsing graph, and the dashed edges denote the backtracking procedure to compute unsatisfiable subformulas.

IV. Boolean Reasoning Techniques

To improve the efficiency of the local search algorithm, we implement some reasoning techniques. One of the techniques 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, then this literal is deleted from that clause; Secondly, we eliminate the clauses which include the literal of the unit clause. For example, consider the formula shown in Equation 2. The clause ($\neg x_2$) is a unit clause, and is propagated to the whole formula. According to the reduction rule, 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

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

After applying unit clause propagation, it is observed that the formula may be 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 the empty clause is reached or no more unit clauses in the remain formula. The order in which the unit clause reductions occur is not important to the correctness of the 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 there exists one pair of opposite polarity literals, and abides by the resolution rule. For instance, a Boolean formula in CNF is given as follows:

$$\varphi_1 = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee x_3)\,. \tag{8}$$

This formula contains three binary clauses, which can be resolved by the inference rule. Then the process of resolution between the binary clauses is

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

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 technique is equality reduction, which is also a 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 ($x \vee y$), 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 the clauses. For example, a Boolean formula in CNF is

$$\begin{aligned} \varphi_2 = {} & (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \wedge \\ & (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3) \end{aligned} \tag{11}$$

Obviously, one may conclude that x_1 is equivalent to 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:

$$\varphi_2' = (x_1 \vee x_3) \wedge (\neg x_1 \vee x_3) \tag{12}$$

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 the empty clause is resolved or no new clause is added.

V. SUBSUMPTION ELIMINATION METHOD

Given a clause C_1 in a CNF formula φ, if a subset of its literals constitutes another clause C_2 in φ, that is, $L(C_1) \supseteq L(C_2)$, where $L(C)$ denotes the set of literals of a clause C, we say C_1 is subsumed by C_2 and C_2 subsumes C_1. Furthermore, if the resolvent of two clauses subsumes either of these two resolving clauses, this is called self-subsumption. For example, let $C_1=(x_1 \vee x_2 \vee x_3)$ and $C_2=(\neg x_1 \vee x_2)$, then resolving on x_1 will produce the resolvent $C=(x_2 \vee x_3)$, which subsumes C_1. Thus after adding C into the formula, C_1 can be deleted. For a formula in CNF, the clauses that are subsumed by other clauses not only slow down the resolution process, but also do not help to prune the search space. Particularly, a subsumed clause never needs to be part of a proof of infeasibility and may be removed without negative effect on resolution. Consequently, we introduce the subsumption detection and removal to shrink the formula and decrease runtime and memory consumption of the local search algorithm substantially. The pseudo code of subsumption elimination procedure is shown in Fig.4.

```
Subsumption_Elimination(resolvent)
1  if (resolvent.size < one of resolving clauses C.size)
2    if ($L(resolvent) \subseteq L(C)$)) then
3      Substitute resolvent for C in the formula
4  else
5    for (each clause $C \in$ formula)
6      if ($L(C) \subseteq L(resolvent)$) then
7        return
8      else if ($L(resolvent) \subseteq L(C)$) then
9        Remove C from the formula
10 Add resolvent into the formula
```

Figure 4. Subsumption Elimination Procedure

The process of subsumption elimination is: when a resolvent is generated by performing resolution on a variable, we first check whether the resolvent subsumes either of the two resolving clauses. If self-subsumption is detected, the subsumed clause is replaced by the resolvent. Then the resolvent is checked against existing clauses in the formula to see if it is subsumed by any of them. If there is a clause which subsumes the resolvent, the resolvent is redundant and should be discarded. This process is called forward subsumption elimination. While a resolvent is added to the current formula, existing clauses in the formula are also checked against the resolvent to see whether they are subsumed. If such a clause exists, it can be removed from the formula. This process is called backward subsumption elimination. To keep the CNF formula subsumption-free, both subsumption checks and self-subsumption detection are employed in the local search algorithm.

VI. Refutation Proof Pruning

During the process of tracking unsatisfiable subformulas from refutation proofs, many redundant clauses bring a degradation of runtime performance and memory consumption. To reduce the search space, we propose a technique called refutation proof pruning, which on-the-fly filters out the clauses not belonging to any infeasible proof of a formula. We keep two fields for each resolvent: one is the list of source trace of this clause, and the other is a counter that tracks the number of offspring of this clause which still have a chance to involve in the refutation proof. Refutation proof pruning contains two functions: the first function named Trace_Updating is to establish or update the two fields of trace information when a new clause is added into the sequence, and the second function called Trace_Pruning is to remove the clauses which are redundant for proof of unsatisfiability. Fig.5 shows the pseudo code of two functions.

```
Trace_Updating(C)
1 C.trace = parent_clauses.trace
2 C.offspring_count = 0
3 for (each clause C1 in C.trace) do
4     C1.offspring_count++
Trace_Pruning(C)
1 if ((C.offspring_count == 0) and (C.trace != ∅))
2     for (each clause C1 in C.trace) do
3         C1.offspring_count—
4         Trace_Pruning(C1)
5     Delete the resolution steps of C
6     Free the space of C.trace
```

Figure 5. Refutation Proof Pruning Procedure

In Fig.5 when a clause is created, the counter of its offspring should be zero. A newly generated clause can potentially take part in the proof, thus the offspring counter of each clause on its resolution trace is incremented. When a clause is removed and its offspring counter does not equal to zero, we keep its list of source trace because we cannot know whether any of its descendants is included in the proof or not. If this clause has no descendant, then its trace list is deleted and the offspring counter for each clause on its source proof is decremented. These counters might become zero, so a recursive call to Trace_Pruning tries to remove each of the resolution sources.

VII. Experimental Results

To experimentally evaluate the effectiveness of our algorithm, we select 10 problem instances from the well-known pigeon hole family and 10 FPGA routing problem instances, and then compare our algorithm with the greedy genetic algorithm[14], which extracts a nearly minimum unsatisfiable subformula. The pigeon hole problem "pigeon_hole*n*" asks whether it is possible to place $n+1$ pigeons in n holes without two pigeons being in the same hole. Another benchmark suite[4,10] is derived from the problem of Boolean-based FPGA detailed routing formulation on island-style FPGA architecture, which is one of the typical applications for unsatisfiable subformulas. The Boolean-based router expresses the routing constraints as a CNF formula which is unsatisfiable if and only if the layout is unroutable.

Our algorithm to find 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 variables (vars) and the number of clauses (clas) for each of the 20 problem instances. Table 1 also gives the total number of minimal unsatisfiable subformulas contained in every formula (MUSes). For generating all minimal unsatisfiable subsets we use the CAMUS algorithms[12]. However there are five instances which fail to obtain all MUSes within 2 hours, and we mark them with *TO* in the table. Table 1 provides the runtime in seconds of the greedy genetic algorithm (GGA time) and the number of clauses that the unsatisfiable subformula contains (GGA size). Furthermore, Table 1 reports the runtime in seconds (time), memory consumption in MB (mem) and size of the derived unsatisfiable subformula (size) for the local search algorithm excluding the refutation proof pruning and subsumption elimination procedure (Basic RbLSA). The last three columns present the CPU time in seconds (time), memory consumption in MB (mem) and size of the resulting unsatisfiable subformula (size) for the whole local search algorithm (RbLSA+RRP+SE). In Table 1, the numbers in bold are denoted as the minimum value among the same parameter of three algorithms.

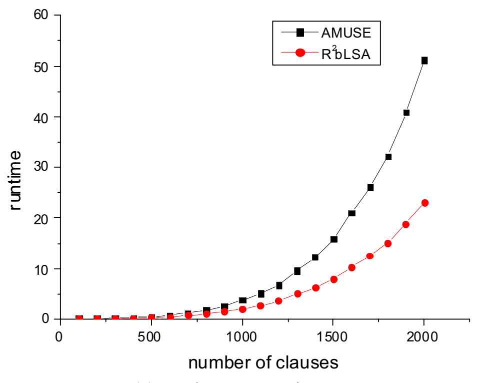
From Table 1, we may observe the following. Both the basic and the whole resolution-based local search algorithm outperform the greedy genetic algorithm for most formulas, except for the instance of pigeon_hole6. For the instances of fpga_routing6 through fpga_routing10 and pigeon_hole10, the greedy genetic algorithm failed to extract the unsatisfiable subformula within the timeout, but our algorithms succeeded in obtaining it. Moreover, the local search algorithms find the minimum unsatisfiable subformula for each formula of the FPGA routing benchmark suite. Further, as compare with the basic algorithm, the whole local search algorithm decrease the runtime by about 10% and the memory consumption by about 20% for most larger instances, and with the size of the formulas increasing, the whole algorithm seems more efficient, mainly owing to the capabilities of the refutation proof pruning and subsumption elimination techniques. However, for a few small and simple problem instances, the experimental results are opposite, because the two pruning procedures bring more overheads than their gains.

TABLE I.
PERFORMANCE RESULTS ON WELL-KNOWN BENCHMARK

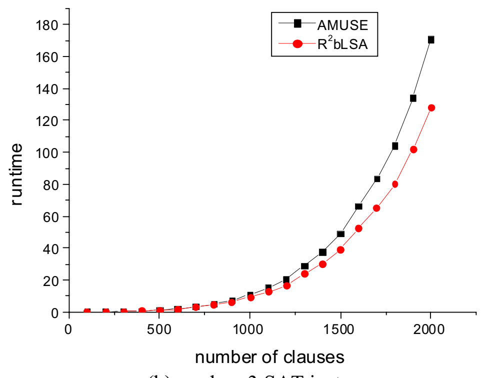
Benchmarks	vars	clas	MUSes	GGA		Basic RbLSA			RbLSA+RRP+SE		
				time	size	time	mem	size	time	mem	size
fpga_routing1	10	17	4	0	11	0	**0.12**	9	0	0.2	9
fpga_routing2	14	25	11	0.02	12	**0**	**0.29**	9	0	0.35	9
fpga_routing3	18	33	26	0.13	10	**0.08**	**0.63**	9	0.1	0.7	9
fpga_routing4	22	41	57	2.16	14	**1.2**	**1.15**	9	1.28	1.2	9
fpga_routing5	26	49	120	51.9	15	27.6	3.25	9	**25.2**	**2.9**	9
fpga_routing6	30	57	*TO*	time out		182.5	10.6	9	**171.0**	**8.75**	9
fpga_routing7	34	65	*TO*	time out		358.0	17.5	9	**321.0**	**14.1**	9
fpga_routing8	38	73	*TO*	time out		617.0	23.4	9	**566.1**	**18.5**	9
fpga_routing9	42	81	*TO*	time out		1040.1	28.0	9	**941.0**	**21.6**	9
fpga_routing10	46	89	*TO*	time out		1690.0	36.5	9	**1507.0**	**27.1**	9
pigeon_hole1	2	3	1	0	3	0	0	3	0	0	3
pigeon_hole2	6	9	1	0	9	0	**0.1**	9	0	0.11	9
pigeon_hole3	12	22	1	0	22	0	**0.38**	22	0	0.46	22
pigeon_hole4	20	45	1	0	45	0	**0.52**	45	0	0.55	45
pigeon_hole5	30	81	1	0.02	81	**0**	**0.64**	81	0.1	0.65	81
pigeon_hole6	42	133	1	**0.08**	133	0.1	0.78	133	0.1	**0.77**	133
pigeon_hole7	56	204	1	0.9	204	0.5	1.12	204	**0.44**	**1.0**	204
pigeon_hole8	72	297	1	51.9	297	22.8	10.8	297	**20.6**	**8.9**	297
pigeon_hole9	90	415	1	1304.0	415	682.6	25.3	415	**605.8**	**21.0**	415
pigeon_hole10	110	561	1	time out		1850.0	58.8	561	**1689.0**	**46.5**	561

To experimentally test the local search algorithms on benchmark with different structures, we used the randomly generated problem instances. The *k*-SAT generator uses as input the number of variables *N*, the number of clauses *C*, the number of literals per clause *k*, and the probability *p*, *q*. Each clause is generated by randomly choosing *m* out of *N* variables, and $m=k$ with probability *q*, and $1\le m\le k$ probability $1-q$, and by determining the sign of each literal (positive or negative) with probability *p*. Fig. 6 shows the performance results of local search algorithm vs. AMUSE[10] on randomly generated 2-SAT and 3-SAT problem instances. Fig.6(a) is the 2-SAT benchmark, and Fig.6(b) is 3-SAT benchmark. In our experiments $p=q=0.5$, and $N=200$. The number of clauses in these instances ranges from 100 to 2000, and increases 100 clauses for each set of instances.

Form the experimental results, our local search algorithm outperforms AMUSE, especially for 2-SAT problem instances. When the number of clauses in the instance is increasing, the performance of our local search algorithm is much better. The causes include three aspects: The first is that the function of deriving unsatisfiable subformula is coupled tightly with the satisfiability checking procedure of the formula. While the resolution is proceeding, the refutation is recorded, and the parsing tree is constructed simultaneously, then the unsatisfiable subformula is computed very efficiently. The second reason is that the decision of satisfiability is implemented simply and performs many more moves per second. The third cause is there are many powerful heuristics in the local search algorithm, especially for unit clauses and binary clauses.



(a) random 2-SAT instances



(b) random 3-SAT instaces

Figure 6. Performance Results on Randomly Generated Benchmarks

VIII. CONCLUSION

In this paper, we present a resolution-based local search algorithm to track small unsatisfiable subformulas from reduced refutation proof. The algorithm is combined with some reasoning and pruning techniques. The experimental results illustrate that our algorithm outperforms the greedy genetic algorithm on well-known benchmarks, and is faster than AMUSE on randomly generated 2-SAT and 3-SAT problem instances. However extensive experimental studies show that this 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, mainly because it makes the decisions on resolution of two long clauses in a stochastic way, and lacks of the effective heuristics for selecting the appropriate clauses. Therefore one of the future works is to explore more aggressive methods for efficient resolution of long clauses.

REFERENCES

[1] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: engineering an efficient SAT solver. Proc. of the 38th Design Automation Conf., Las Vegas: ACM Press, 2001. 530-535.

[2] N. Een, N. Sorensson. An extensible SAT-solver. Proc. of the 6th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 2919, Heidelberg: Springer-Verlag, 2003. 502-518.

[3] K.L. McMillan, N. Amla. Automatic abstraction without counterexamples. Proc. of the 9th International Conf. on Tools and Algorithms for the Construction and Analysis of Systems, *LNCS* 2619, Heidelberg: Springer-Verlag, 2003. 2-17.

[4] G.J. Nam, F. Aloul, K. Sakallah, R. Rutenbar. A comparative study of two Boolean formulations of FPGA detailed routing constraints. Proc. of the 2001 International Symposium on Physical Design, Sonoma County: ACM Press, 2001. 222-227.

[5] S.Y. Shen, Y. Qin, S.K. Li. A faster counterexample minimization algorithm based on refutation analysis. Proceedings of 2005 Design Automation and Test in Europe Conference, Munich: IEEE Press, 2005. 672-677.

[6] B. Mazure, L. Sais, and E. Gregoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22(3-4):319-331, 1998.

[7] R. Bruni. Approximating minimal unsatisfiable subformulas by means of adaptive core search. *Discrete Applied Mathematics*, 2003, 130(2): 85~100.

[8] L. Zhang, S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. Proc. of the 6th International Conf. on Theory and Applications of Satisfiability Testing, 2003.

[9] I. Lynce, J. Marques-Silva. On computing minimum unsatisfiable cores. Proc. of the 7th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 3542, Heidelberg: Springer-Verlag 2004. 305-310.

[10] Y. Oh, M.N. Mneimneh, Z.S. Andraus, K.A. Sakallah, I.L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. Proc. of the 41st Design Automation Conf., San Diego: ACM Press, 2004. 518-523.

[11] J. Huang. MUP: A minimal unsatisfiability prover. Proc. of the 10th Asia and South Pacific Design Automation Conf., Shanghai: ACM Press, 2005. 432-437.

[12] M.H. Liffiton, K.A. Sakallah. On finding all minimally unsatisfiable subformulas. Proc. of the 8th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 3569, Heidelberg: Springer-Verlag, 2005. 173-186.

[13] M.N. Mneimneh, I. Lynce, Z.S. Andraus, J.P. Marques-Silva, K.A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. Proc. of the 8th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 3569, Heidelberg: Springer-Verlag, 2005. 393-399.

[14] J.M. Zhang, S.K. Li, S.Y. Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. Proc. of the 19th Australian Joint Conference on Artificial Intelligence, *LNCS* 4304, Heidelberg: Springer-Verlag, 2006. 847-856.

[15] R. Gershman, M. Koifman, O. Strichman, Deriving small unsatisfiable cores with dominator. Proc. of the 18th International. Conf. on Computer Aided Verification, *LNCS* 4144, Heidelberg: Springer-Verlag, 2006. 109-122.

[16] N. Dershowitz, Z. Hanna, A. Nadel, A scalable algorithm for minimal unsatisfiable core extraction. Proc. of the 9th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 4121, Heidelberg: Springer-Verlag, 2006. 36-41.

[17] Maaren H, Wieringa S. Finding guaranteed MUSes fast. In: Buning HK, Zhao X, eds. Proc. of 11th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 4996, Berlin: Springer-Verlag, 2008. 291-304.

[18] E. Gregoire, B. Mazuer, C. Piette. Local-search extraction of MUSes. *Constraints*, 2007, 12(3): 325-344.

[19] S. Prestwich, I. Lynce. Local search for unsatisfiability. Proc. of the 9th International Conf. on Theory and Applications of Satisfiability Testing, *LNCS* 4121, Heidelberg: Springer-Verlag, 2006. 283-296.

[20] G. Audemard, L. Simon. GUNSAT: A greedy local search algorithm for unsatisfiability. Proc. of the Twentieth International Joint Conf. of Artificial Intelligence, Hyderabad, India, 2007. 2256-2261.

Jianmin Zhang born in China, 1979. Ph. D. in computer science, earned in National University of Defense Technology, Changsha, China, 2008. His major fields of study include formal verification of hardware.

Shengyu Shen born in China, 1975. Ph. D. in computer science, earned in National University of Defense Technology, Changsha, China, 2005. His major fields of study include formal verification of hardware.

Sikun Li born in China, 1941. Professor in computer science. His major fields of study include VLSI designing and verification.