Verification of Proofs of Unsatisfiability for CNF Formulas

Evgueni Goldberg
Cadence Berkeley Labs (USA),
Email: egold@cadence.com

Yakov Novikov
The United Institute of Informatics Problems,
National Academy of Sciences (Belarus),
Email: nov@newman.bas-net.by

Abstract

As SAT-algorithms become more and more complex, there is little chance of writing a SAT-solver that is free of bugs. So it is of great importance to be able to verify the information returned by a SAT-solver. If the CNF formula to be tested is satisfiable, solution verification is trivial and can be easily done by the user. However, in the case of unsatisfiability, the user has to rely on the reputation of the SAT-solver. We describe an efficient procedure for checking the correctness of unsatisfiability proofs. As a by-product, the proposed procedure finds an unsatisfiable core of the initial CNF formula. The efficiency of the proposed procedure was tested on a representative set of large "real-life" CNF formulas from the formal verification domain.

1. Introduction

Many problems such as ATPG [10], logic synthesis [3], equivalence checking [4,8], and bounded model checking [2] reduce to the satisfiability problem. In the last decade substantial progress has been made in the development of practical SAT algorithms [1,5,9,10, 13,14,16]. As a result, SAT-solvers are becoming commercially viable. However, due to the growing complexity of the state-of-the art algorithms it is unlikely that a SAT-solver will be free of bugs. Hence it is important to run an independent check of the information returned by a SAT-solver so that the latter can be used even if it is buggy.

When testing the satisfiability of a CNF formula a SAT-solver either 1) returns an assignment of values that satisfies all the clauses of the formula or 2) reports that such an assignment does not exist. In the first case, it is trivial to check whether the returned solution is correct. To verify the second kind of an answer, one needs much more information about SAT-solver's work.

One way to verify a proof of unsatisfiability is to build a resolution directed acyclic graph (DAG) G [7,12]. We will refer to such a DAG as a *resolution graph proof*. The sources of G are assigned clauses of the initial CNF formula. Each internal (i.e. different from a source) node v has two incoming edges going out of two (parent) nodes v_1 and v_2. The proof verification procedure consists of gradual assigning clauses to the internal nodes of G. As soon as the two parent nodes v_1 and v_2 of node v are assigned clauses (denote them by $C(v_1)$ and $C(v_2)$ respectively), the child clause $C(v)$ is produced by resolving $C(v_1)$ and $C(v_2)$. The proof specified by DAG G is correct if 1) for any parent nodes v_1 and v_2 clauses $C(v_1)$ and $C(v_2)$ have the opposite literals of exactly one variable (and so they can be resolved in this variable); 2) A sink node of the DAG G will be eventually assigned the empty clause.

In [7] it is explained how a resolution graph can be built by a SAT-solver based on the DPLL procedure and the state-of-the-art techniques. The advantage of the approach is that the procedure of proof verification is very simple. However, there are at least two potential drawbacks. First, generation of a resolution proof may take a substantial rewriting of some parts of the SAT-solver. Second, the size of the resolution graph to be stored may get prohibitively large. Addressing the last issue is the main motivation of the paper.

We present a simple verification procedure that is, in a sense, complementary to building a resolution graph. This procedure can be applied to all state-of-the-art SAT-solvers based on conflict clause recording, for example [1,9,13,14,16]. We used the proposed procedure to verify proofs obtained by our SAT-solver BerkMin [9]. The idea of the verification procedure is to represent the proof as a chronologically ordered set of the conflict clauses. (Here, we assume that any observed conflict assignment was accompanied by recording a conflict clause. In practice, as soon as the SAT-solver hits a conflict, the corresponding conflict clause is output to disk.). We will refer to such a proof as a *conflict clause proof*. To verify a conflict clause proof one just checks whether each conflict clause was deduced correctly.

Let F^* be a conflict clause proof i.e. the set of all deduced conflict clauses. To prove the correctness of a conflict clause C we form the CNF formula G obtained by adding to the initial CNF F all the clauses of F^* deduced before C. Then we falsify C by making the assignment of values A setting the literals of C to 0. The key point is that

if C was deduced "correctly" then the Boolean Constraint Propagation (BCP) procedure triggered by A in G *will lead to a conflict*. Obtaining such a conflict means that C is implied by G (and hence by F). If for all the deduced clauses of the proof their correctness has been established, then the whole proof is correct. Otherwise, one can point to a clause of the proof whose deduction is questionable.

Conflict clause proofs have the following advantages. First, such proofs are, in general, shorter than resolution graph ones (see Section 5). Second, it takes only a slight modification of the SAT-solver, to generate a conflict clause proof. Besides, SAT-solver's performance does not change much. In our experiments, outputting all the conflict clauses to disk took about 10% of the total runtime of the SAT-solver. A potential disadvantage of conflict clause proofs is that the proof verification procedure is more time consuming and a bit more complex than for resolution graph proofs. However, the BCP procedure (the only procedure one needs to implement to verify a conflict clause proof) is well established and it should not be a problem for a user to implement it. Besides, our experiments showed that verification of a conflict clause proof can be done in a reasonable time. (Unfortunately, we could not directly compare our results with data on resolution graph proof verification because no experimental results have been ever reported on resolution proofs to the best of our knowledge).

In the procedure sketched above we have to verify the correctness of each conflict clause of F^*. (The order in which clauses are checked does not matter.) At the same time, some conflict clauses may not contribute to the deduction of the empty clause and so checking their correctness is a waste of time. This problem is easily solved by checking conflict clauses in the order that is reverse to chronological (i.e. we start with clauses deduced last). Then we can limit verification checks only to clauses that have been actually used in deducing the empty clause. This is done by marking the clauses of F^* that were used at least once in a BCP procedure invoked when verifying a conflict clause. Initially, only the last two deduced unit clauses of F^* are marked. If during verification one reaches a conflict clause C of F^* that has not been marked, it can be skipped (because C has never been used in deducing the empty clause).

In some applications the user may want to know what subset of clauses of the original CNF formula is responsible for its unsatisfiability. The extraction of an *unsatisfiable core* of the formula can help to understand the "cause" of unsatisfiability. Such a core can be identified as a "by-product" of the procedure above. The idea is that during BCP procedures one marks not only the used conflict clauses of F^* but also the used clauses of the original CNF formula. After completing proof verification, the subset of clauses of the initial formula F that turned out to be marked after completing proof verification forms an unsatisfiabile core of F.

The paper is organized as follows. In Section 2 we introduce basic notions. Section 3 describes a conflict clause proof verification procedure. In Section 4 a more efficient version of the procedure of Section 3 is described that can also identify an unsatisfiable core of the initial formula. Conflict clause and resolution graph proofs and their verification procedures are compared in Section 5. In Section 6 experimental results are given. In Section 7 we make some conclusions.

2. Basic notions

Given a conjunctive normal form (CNF) F specified on a set of variables $\{x_1,\ldots,x_n\}$, *the satisfiability problem* (SAT) is to satisfy (set to 1) all the disjunctions of F by some assignment of values to variables from $\{x_1,\ldots,x_n\}$. If such an assignment does not exist, CNF F is said to be *unsatisfiable*. A disjunction of literals is also called *a clause*. A clause containing one literal is called *unit*. Two unit clauses consisting of the opposite literals of a variable are called *a conflicting pair*. The Boolean Constraint Propagation procedure (BCP) [5] is as follows.

```
Conflicting_pair, Assignments ← BCP(CNF F)
    {While (there is no conflicting pair and
            there is a unit clause l )
        {Assignment = satisfy( l );
         Assignments = Assignments ∪ Assignment
         F= simplify( F, Assignment);
        }
     return(conflicting_pair(F),Assignments);
    }
```

The **satisfy** procedure returns the assignment satisfying the unit clause l. Such an assignment is called *deduced*. For instance, assignment x=0 is deduced from the unit clause $\sim x$. The **simplify** procedure removes the clauses satisfied by *Assignment* from formula F. For instance, if *Assignment* is x=0, then all the clauses containing $\sim x$ are removed from F. Besides, literal x is removed from all the clauses of CNF F containing it. The situation when the BCP procedure produces a conflicting pair is called a *conflict*. The ***BCP***(F) procedure returns either the found conflicting pair or the set of assignments accumulated during the procedure.

Note that in the implementation of the BCP procedure literals and clauses are not physically removed. Instead, some variables are marked as assigned (and the assigned values are stored) and some clauses are marked as satisfied. This way one can easily restore the right CNF formula after undoing some assignments. To speed up the BCP procedure one can use the idea of watched literals [16], the description of which we omit.

Let R be a set of assignments. Each assignment from R specifies the literal which is set to 0 by this assignment. Denote by $C(R)$ the disjunction of literals specified by R. (Obviously, clause $C(R)$ is falsified by assignment R.) We

will say that the clause $C(R)$ *encodes* the *assignment* R. In its turn, the assignment R is said to *specify* the *clause* $C(R)$.

Denote by $F|R$ the CNF that is obtained from F by making the assignments from R and removing all the satisfied clauses and all literals set to 0. (Henceforth, we assume that no clause of F is falsified by R.) If **BCP**($F|R$) leads to a conflict due to appearance of a conflicting pair (l, $\sim l$) we will say that R is *responsible* for that conflict. Suppose that R is responsible for a conflict in CNF F. Then clause $C(R)$ is called a *conflict clause* for F. It can be shown that a conflict clause is implied by F (because it can be obtained by resolving clauses of F). Due to this property one can always add a conflict clause to the current CNF and this is exactly what SAT-solvers based on conflict clause recording do.

Proof of the unsatisfiability of a CNF performed by such SAT-solvers can be considered as a sequence of steps. At each step a conflict clause is added to the current CNF formula F (at the first step F is equal to the initial CNF formula). Once in a while, some clauses are removed from the current formula. The proof terminates if after obtaining a unit conflict clause (say clause $\sim x$) we prove that the unit clause x is also a conflict one. The pair of unit clauses $\sim x$ and x is called *the final conflicting pair* (in contrast to a conflicting pair obtained in each BCP procedure leading to a conflict.)

We consider a proof to be correct if each step of the proof is correct. That is, each clause C added to the current CNF F is indeed a conflict one. This means that **BCP**($F|R$) where R is the set of assignments encoded by C returns a conflicting pair.

3. Conflict clause proof verification

Let F be an unsatisfiable CNF formula and F^* be the set of all deduced clauses. For the sake of clarity, we will view F^* as a chronologically ordered stack of clauses where the last (first) conflict clause is located at the top (at the bottom). We assume that the two topmost clauses of F^* form the final conflicting pair. The verification procedure is as follows.

```
bool Proof_verification1(CNF F, CNF F*) {
  {While(F* is not empty)
     {C = pop_clause_off ( F* );
     R=assignments_encoded_by( C );
      if (no_confl_pair== BCP( (F∪ F*)|R)
          return proof_is_not_correct;
      }
   return proof_is_correct;
  }
```

As it was mentioned before, if one checks the correctness of all the clauses of F^*, the order in which clauses are processed does not matter. In the procedure above, clauses are verified in the order that is opposite to chronological (i.e. we start verification with the two clauses of the final conflicting pair). The reason for such a choice will be explained in the next section.

Note that in the procedure above, one runs BCP for $(F\cup F^*)|R$. On the other hand, at the point of obtaining the conflict clause $C(R)$ (when testing the satisfiability of F) the current CNF was equal to $F\cup F'$ where $F'\subseteq F^*$ because some conflict clauses may have been dropped. However, if **BCP**($(F\cup F^*)|R$) does not find a conflict, **BCP**($(F\cup F')|R$) would not find a conflict either. So if the **Proof_verification1** procedure returns *proof_is_not_correct*, the SAT-solver contains a bug. On the other hand, if the procedure returns *proof_is_correct* it may validate a correct proof produced by a buggy SAT-solver. (It is possible that **BCP**($(F\cup F^*)|R$) produces a conflict while **BCP**($(F\cup F')|R$) did not and the conflict clause $C(R)$ was deduced "by mistake".)

4. Extraction of unsatisfiable core

In this section, we describe a more efficient proof verification procedure that also extracts an unsatisfiable subset of clauses of the original formula as a "by-product" of verification. The idea is that some conflict clauses are "redundant" from the viewpoint of the proof. A proof of unsatisfiability F^* has to contain two conflict clauses (l, $\sim l$) forming the final conflicting pair. By redundancy of a conflict clause C we mean that no descendent of C has ever been used for deducing l and $\sim l$. Obviously, checking the correctness of deducing C is a waste of time.

The identification of redundant conflict clauses can be done by modifying the **Proof_verification1** procedure. The modification is that when checking the correctness of deducing a clause C of F^* one marks all the clauses that are involved in producing the conflict found by **BCP**($(F\cup F^*)|R$). Initially, only the two clauses of F^* forming the final conflicting pair (l, $\sim l$) are marked.

```
bool Proof_verfication2(CNF F, CNF F*)
    {While(F* is not empty)
        {C = pop_clause_off ( F* );
        if (C is not marked)
            continue;
        R=assignments_encoded_by( C );
        (Assgns, confl_pair) = BCP((F∪ F*)|R)
        if (confl_pair == ∅)
            return proof_is_not_correct;
        Conflict_analysis(Assgns,confl_pair,F ,F*);
        }
    return proof_is_correct;
    }
```

In contrast to **Proof_verification1**, in the **Proof_verification2** procedure described above, clause C is checked for correctness only if it has been marked. Since the clauses of F^* are processed in the chronologically reverse order, the fact that a clause C is not marked means that none of its descendents has contributed

to deducing l or $\sim l$. The marking is done by the **Conflict_analysis** procedure. This procedure just marks all the clauses of F and F^* that are responsible for the conflict produced by **BCP**$((F \cup F^*)|R)$. This is done by processing deduced assignments in the reverse order starting with the conflicting pair of literals produced by BCP. Suppose for example that x and $\sim x$ is the conflicting pair produced by the BCP procedure and those literals where deduced from clauses $C'= x+v+\sim w$ and $C''=\sim x+y+z$. Then C' and C'' get marked. After that, for each of the literals of the set $S=\{v,\sim w, y,z\}$ the following procedure is applied. If a literal $p \in S$ is in the clause C whose deduction is tested for correctness, then nothing happens. However, if p was deduced from a clause of F or F^*, that clause gets marked. The same procedure repeats recursively for the literals of each new marked clause.

Note that the **Conflict_analysis** procedure described above marks clauses of both F^* and the original formula F. If a clause of F is left unmarked after applying the **Proof_verification2** procedure it means that this clause has never been employed in deducing a "useful" clause of F^*. So it can be removed from F without affecting the unsatisfiability of the latter. Hence the set of marked clauses of F forms an unsatisfiable core.

5. Resolution graph proof verification versus conflict clause proof verification

In this section, we compare conflict clause and resolution graph proofs in more detail. Let G be a resolution graph. A source node of G is labeled with a clause of the initial formula. Each internal node of G has two parent nodes. Verification check consists of assigning clauses to internal nodes of G. As soon as the two parent nodes are assigned clauses, the clause corresponding to the child node can be produced by resolving the two parent clauses. The proof is correct if the resolution of each pair of parent clauses produces a non-tautologous (i.e. not having opposite literals of the same variable) clause and the empty clause is deduced at a sink node of G .

Let F^* be the set of conflict clauses produced by a SAT-solver when proving that F is unsatisfiable. Let G be the resolution graph corresponding to the same run of the SAT-solver that produced the proof F^*. In the general case, for each node of G one needs to store at least three numbers (the label of the node and the labels of the parents). However in the case of SAT-solvers based on conflict clause recording, it is sufficient to store only one label per node using a special representation of the resolution graph [12].

In the worst case, the size of a resolution graph is $O(|F^*|^2)$ (because for deducing each of the F^* clauses one may need to resolve $O(F^*)$ clauses.) The size of a conflict clause proof is $O(n \cdot |F^*|)$. A substantial difference between the two kinds of proofs is that the size of a conflict clause proof does not change during proof verification. On the other hand, when verifying a resolution graph proof, one has to assign clauses to internal nodes of the graph. So, in a sense, the size of a conflict clause proof gives a lower bound on the maximum size the corresponding resolution graph proof may grow to during proof verification. (*Each conflict clause will be eventually assigned to an internal node of the resolution graph*.)

To analyze the factors affecting the size of resolution graph and conflict clause proofs one needs to introduce the notion of "local" and "global" conflict clauses. Informally, a conflict clause C is *local* if it is obtained by resolving a small number of clauses. In the corresponding resolution, graph the deduction of the clause C is represented by a small set A of internal nodes. On the other hand, in a conflict clause proof, one has to store the conflict clause itself. In the case C is long, storing its literals may turn out to be more space consuming than storing the nodes of A. (Of course during proof verification, C has to be deduced and assigned to a node of the resolution graph.). Informally, a conflict clause C is called *global* if it is obtained by resolving many clauses of the current CNF formula. In this case, especially if C is a short clause, storing the literals of C in a conflict clause proof is much more space efficient than storing the nodes specifying the deduction of C in a resolution graph proof.

One way to deduce a global conflict clause is to represent it in terms of literals of decision variables. In this case one keeps resolving the clause falsified in the conflict and its descendents until a clause containing only literals of decision variables is obtained. (More detailed description of different conflict driven learning schemes can be found in [17]). Such a way of constructing a conflict clause is used in the SAT-solver Relsat [1]. The reason why conflict clauses specified in terms of decision variables are global is that to obtain such a clause one has to resolve many clauses of the current formula. On the other hand, the SAT-solver Chaff [13] deduces local conflict clauses because it uses the 1 UIP learning scheme (in terms of paper [17]). In this case a conflict clause usually contains a lot of literals of deduced variables and is typically obtained by a small number of resolutions.

It is not hard to see that conflict clause and resolution graph proofs are complementary from the viewpoint of size. If a SAT-solver proved the unsatisfiability of a formula deducing only local conflict clauses it makes sense to represent the obtained proof as a resolution graph. However, if a substantial number of deduced clauses are global, then representing the proof as a set of conflict clauses is the best (and perhaps the only) choice.

6. Experimental results

In the experiments we used our SAT-solver BerkMin [9]. The experiments were carried out on a PC with clock frequency of 500 MHz and 640 Mbytes of memory running Windows. The objective of experiments was a) to estimate

the practicality of the proposed approach; b) to identify unsatisfiable cores of some known benchmarks; c) to show that resolution graph proofs may grow very large. BerkMin is well suited for proving the third point. The reason is that once in a while BerkMin deduces clauses in terms of decision variables (i.e. "global" clauses). This is a new feature of BerkMin not described in [9]. We found out that for some instances, combining the deduction of local and global clauses gives a noticeable speed-up.

In the implementation of **Proof_verfication2** we used an optimized version of the BCP procedure that employs the machinery of watched literals [16]. A conflict clause proof F^* contains a large number of long clauses, which is exactly the case when using watched literals is especially effective. Of course, implementing the technique of watched literals makes the verification program more complex and so more prone to bugs. On the other hand, the machinery of watched literals has been well studied in the state-of-the-art SAT-solvers [9,13,16]. Besides, the code of a verification program is "stable" (in contrast to SAT-solvers whose code keeps changing).

Name	All conflict clauses	Tested %	Number of clauses in the initial CNF	Unsatisfiable core %
verification of pipelined microprocessors [15]				
5pipe	20,137	44.6	195,452	21.5
5pipe_1	43,597	56.2	187,545	36.7
5pipe_5	34,209	50.3	240,892	30.4
6pipe	213,923	47.0	394,739	48.9
6pipe_6	110,161	37.5	545,612	39.6
7pipe	365,245	34.2	751,118	44.7
9vliw	88,975	49.1	179,492	51.1
verification of PicoJava II™ microprocessor [21]				
exmp72	30,036	75.5	148,536	31.7
exmp73	54,014	61.0	219,972	38.7
exmp74	46,557	63.1	141,432	43.3
exmp75	29,761	72.0	284,446	22.6
bounded model checking [20]				
Barrel7	44,024	83.0	13,765	66.1
Barrel8	123,712	94.1	20,083	66.7
Barrel9	46,423	56.1	36,606	70.7
Longmult12	113,698	89.7	18,645	71.2
Longmult13	111,421	88.2	20,487	72.7
Longmult14	117,215	88.0	22,389	72.6
Longmult15	110,074	90.3	24,351	71.1
equivalence checking [19]				
c3540	15,433	67.2	9,326	98.6
c5315	16,132	75.0	15,024	95.3
c7572	22,307	77.9	20,423	97.3
bounded model checking, SAT-2002 [18]				
w10_45	4,285	84.34	51,803	26.3
w10_60	14,489	78.65	83,538	33.0
w10_70	32,847	81.44	103,556	41.5

Table 1. Unsatisfiable core extraction

In Tables 1,2,3 we give experimental results on hard instances from the verification domain. Table 1 gives the data on unsatisfiable core extraction. Name of the instances are given in the first column. The "All conflict clauses" column gives the cardinality of the set F^*. The "Tested" column shows the percentage of clauses of F^* that got marked and hence were tested for correctness. On the one hand, these numbers show that **proof_verfication2** is, indeed, more efficient than **proof_verification1**. On the other hand, the percentage of tested clauses allows one to estimate "the coefficient of efficiency" of the used SAT-solver that is the share of deduced conflict clauses actually used in the proof of unsatisfiability. The "Unsatisfiable core" column shows the percentage of clauses of the initial CNF formula that formed the found unsatisfiable core.

Name	Verification time (sec.)	Resolution graph size (in thousands of nodes)	Confl. clause proof size (in thousands of lit.)	(Confl. clause proof size)/ (res. proof size) %
5pipe	25.7	1,128	1,234	109.5
5pipe_1	110.8	10,592	2,747	25.9
5pipe_5	69.6	6,319	2,575	40.8
6pipe	747.6	105,506	24,947	23.7
6pipe_6	446.2	55,406	9,797	17.7
7pipe	1,902.	435,726	60,312	13.8
9vliw	433.4	8,756	5,877	67.1
exmp72	340.5	6,875	1,768	25.7
exmp73	536.6	9,705	5,039	51.9
exmp74	307.3	5,828	2,973	51.0
exmp75	519.3	4,394	1,237	28.2
Barrel7	138.0	3,915	4,691	119.8
Barrel8	1579.2	21,955	26,844	122.3
Barrel9	63.9	2,919	2,959	101.4
Longmult12	1366.1	30,138	8,487	28.2
Longmult13	1306.3	32,124	8,939	27.8
Longmult14	1417.6	35,734	9,592	26.8
Longmult15	1251.7	26,945	8,346	31.0
c3540	16.5	623	724	116.2
c5315	7.0	441	416	94.4
c7572	17.3	761	726	95.4
w10_45	20.5	532	89	16.7
w10_60	104.4	1,844	440	23.9
w10_70	354.6	6,723	1,303	19.4

Table 2. Proof verification

Table 2 gives data about proof verification. The "Verification time" column shows the time taken by **Proof_verification2**. (*Typically, verifying a proof that a formula F was unsatisfiabile took 2-3 times the time one needed to generate the proof i.e. to test the satisfiability of F.*) The "Resolution graph size" column shows a lower bound on the number of *nodes* in the resolution graph (in thousands). For instance, the size of the graph for 5pipe would be greater or equal to 1 million and 128 thousand

nodes. The reason for computing only a lower bound of the resolution graph size is that some conflict clauses are built by our SAT-solver using an involved procedure. To avoid writing too much extra code, if a conflict clause was obtained using that procedure, we computed only a lower bound on the number of resolutions one has to apply to produce the clause. (For the rest of the conflict clauses we computed the number of resolutions exactly. So we believe the lower bounds shown in Table2 are close to the real sizes.) The "Confl. clause proof size" column contains the total number of *literals* in the clauses of *F** (in thousands). The last column gives the ratio (per cent) of conflict clause and resolution proof sizes.

It is not hard to see that with the exception of a few instances conflict clause proofs are smaller than resolution graph ones. (In Table 2 we estimate only the initial size of a resolution graph. That is we do not take into account that, as it was mentioned in Section 5, the size of the resolution proof grows during proof verification.)

Name	Resol. proof size (in thousands of nodes)	Confl. cl. proof size (in thousands of literals)	Ratio %
bounded model checking, SAT-2002 [18]			
fifo8_200	379,992	71,971	18
fifo8_300	987,840	118,132	11
fifo8_400	4,581,450	335,752	7

Table 3. Growth of resolution proof size

The size of the largest proof of Table 2 (formula 7pipe) was 257 Mbyte and so we were able to verify the proof on the computer with 640 Mbytes of memory. On the other hand, the corresponding resolution graph proof contained 435 million nodes and so the resolution graph would take more than 2 Gbytes of memory (assuming that on average one needs 5 digits to label a node of the resolution graph). Instances of the pipe family show that the gap between conflict clause proofs and resolution graph ones may widen as the size of instances grows. One more example of this trend is shown in Table 3 where the ratio of sizes of conflict clause and resolution graph proofs decreases from 18% to 7% as the size of intstances grows.

7. Conclusion

We introduced a simple procedure for the verification of proofs of unsatisfiability for CNF formulas where a proof is represented as a chronologically ordered set of conflict clauses. Conflict clause proofs are complementary to resolution graph proofs and should be used when the size of the resolution graph proof grows too large. Experiments show that conflict clause proofs can be generated even for large real-life formulas and the verification of a conflict clause proof can be completed in a reasonable time.

References

1. R.J.J.Bayardo, R.C. Schrag. *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*, in: Proceeding of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), Providence,Rhode Island, 1997, pp. 203-208.
2. A. Biere et.al. *Symbolic model checking using SAT procedures instead of BDDs.* Proceedings of Design Automation Conference, DAC'99. -1999.
3. R.K.Brayton et. al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, -1984.
4. J.Burch, V.Singhal. *Tight Integration of Combinational Verification Methods.* Proceedings of International. Conf. on Computer-Aided Design.-1998.
5. M.Davis, G.Longemann, D.Loveland. *A Machine program for theorem proving.* Communications of the ACM. -1962. -V.5. -P.394-397.
6. O.Dubois, P.Andre, Y. Boufkhad, J.Carlier. *SAT versus UNSAT*. In: Johnson and Trick, Second DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996, pp.415-436.
7. A. Van Gelder: *Extracting (easily) checkable proofs from a satisfiability solver*. In Proceedings of the 7-th International Symposium on Artificial Intelligence and Mathematics, January 2002.
8. E. Goldberg, M.Prasad. *Using Sat for combinational equivalence checking.* Proceedings of Design, Automation and Test in Europe Conference. –2001. -pp.114-121.
9. E.Goldberg,Y.Novikov. *BerkMin:A fast and robust SAT_solver*. Proceedings of Design, Automation and Test in Europe Conference, DATE-2002, pp.142-149.
10. J.W. Freeman. *Improvements to propositional satisfiability search algorithms.* Ph.D. thesis, University of Pennsylvania, Philadelphia , 1995.
11. T.Larrabee. *Test pattern generation using Boolean satisfiability*. IEEE transactions on computer-aided design, vol. 11, pp.4-15, January 1992.
12. K.McMillan. *Private communication.*
13. M.W.Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. *Chaff: Engineering an Efficient SAT Solver*. In: Proceeding of the 38th Design Automation Conference (DAC'01), 2001.
14. J.P.M.Silva, K.A.Sakallah. *GRASP: A Search Algorithm for Propositional Satisfiability.* IEEE Transactions of Computers. -1999. -V. 48. -P. 506-521.
15. M.Velev. *CMU benchmark suite*. Available from http://www.ece.cmu.edu/~mvelev.
16. H.Zhang. *SATO: An efficient propositional prover*. Proceedings of the International Conference on Automated Deduction. -July 1997. -P.272-275.
17. L.Zhang, C.F.Madigan, M.H.Moskewicz, S.Malik. *Efficient Conflict Driven Learning in a Boolean Satisfiability Solver*. International Conference on Computer-Aided Design (ICCAD '01), November 2001, pp. 279-285.
18. http://www.ececs.uc.edu/sat2002/scripts/menu_choix2.php3
19. http://sat.inesc.pt/benchmarks/cnf/equiv-checking
20. http://www.mrg.dist.unige.it/star/sim/
21. http://www-cad.eecs.berkeley.edu/ ~kenmcmil