Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula

Lintao Zhang and Sharad Malik

Department of Electrical Engineering,
Princeton University, Princeton NJ 08544, USA
{lintaoz,sharad}@ee.princeton.edu

Abstract. Given an unsatisfiable Boolean propositional formula in Conjunctive Normal Form, we investigate the problem of extracting a subset of the clauses of the formula such that the conjunction of these clauses are still unsatisfiable. We call the unsatisfiable subformula the unsatisfiable core of the original formula. We propose an efficient procedure to extract an unsatisfiable core from an unsatisfiability proof of the formula provide by a Boolean Satisfiability (SAT) solver. Our procedure for unsatisfiable core extraction is fully automatic and scales well on very large (and hard) CNF instances generated from real world applications. Extensive experimental results are provided to validate our proposed procedure.

1 Introduction and Previous Work

A propositional Boolean formula is said to be in Conjunctive Normal Form (CNF) if it consists of a conjunction (logic *and*) of clauses C_i, each is a disjunction (logic *or*) of literals. A literal is the occurrence of a variable in either positive or negative *phase*. A CNF formula can be written as:

$$F = C_1 C_2 \dots C_n$$

where $C_i (i = 1 \dots n)$ are clauses. Given such an CNF formula that is unsatisfiable, there exists a subset of clauses $\phi \subseteq \{C_i | i = 1 \dots n\}$ such that the formula formed by conjunct the clauses in ϕ is unsatisfiable. We call the formula obtained by conjunct the clauses in ϕ an *unsatisfiable core* of the original formula. Such an unsatisfiable core may (but not necessarily need to) contain much smaller number of clauses than the original formula.

There are many applications that can benefit from being able to obtain a *small* unsatisfiable core from an unsatisfiable Boolean formula. In application such as planning [7]. A satisfiable assignment for the SAT instance implies that there exists a viable scheduling. Therefore, if a planning is proven unfeasible, a small unsatisfiable core can help locating the problem more quickly. In FPGA routing [10], an unsatisfiable instance implies that the channel is unroutable. By localizing the reasons for the unsatisfiability to the unsatisfiable core, it would be easier to determine the reasons for the failure of routing.

The unsatisfiable core problem has been studied by Bruni and Sassano in [2]. In that paper, the authors discussed a method to find *minimal unsatisfiable subformula (MUS)* of an unsatisfiable CNF instance. In their approach, an adaptive search procedure is used to heuristically estimate the *hardness* of the clauses and try to find a small subformula consisted of clauses that are deemed *hard* in the original formula. The procedure performs SAT checking on the subformula iteratively, begins with a very small subformula consists of a small number of the hardest clauses. If the subformula is satisfiable, more clauses are added into the subformula. If the search procedure cannot determine the satisfiability of the subformula after certain number of branches, some clauses are removed from it. The process goes on until the subformula can be proven to be unsatisfiable. In that case the subformula will be the resulting unsatisfiable core. The search is adaptive in the sense that the hardness of clauses change during the SAT checking. The experimental results show that the procedure is successful for finding small unsatisfiable cores for the instances tested. However, the benchmarks used are very small instances that are trivial for current state-of-the-art SAT solvers. Moreover, the proposed procedure needs careful tuning of run time parameters (i.e. initial size of the subformula, cutoff for SAT checking, number of clauses added/removed for each iteration) for each formula being processed. Therefore, the procedure is not fully automatic. The procedure may not scale well on difficult SAT instances generated from real world applications because such iterative procedure may take too much time to converge. Therefore, the procedure may not be practical for real world applications.

In this paper, we propose another procedure that can extract a small unsatisfiable core from an unsatisfiable Boolean formula. The procedure is based on a SAT solver validation procedure proposed in [15]. In that paper, we show how to check an unsatisfiable proof provided by a SAT solver. As a by product, the checking procedure can produce the unsatisfiable core of the instance being validated. However, in that approach the procedure may fail on large proofs due to memory overflow. In this paper, we improve the procedure proposed in [15] so that we can extract the unsatisfiable cores from arbitrarily large unsatisfiable proofs.

The procedure we propose use the resolution graph produced by a SAT solver to determine an unsatisfiable core of the original formula. The idea of regarding a DLL search as a resolution process is not new and have been studied by various authors. Using the information provide by the resolution graph for a certification of the SAT proof has also been discussed before [12, 15]. The contribution of this paper is to use the same resolution graph for unsatisfiable core extraction. Since in this work we assume the SAT solver (and the proof) to be valid, we can avoid the overhead of checking the integrity of the resolution graph itself. Instead, we provide an efficient algorithm to traverse the graph on hard disk so that the procedure is not memory limited. Our procedure for unsatisfiable core extraction is fully automated and scales well on very large (and hard) CNF instances generated from real world applications. Extensive experimental results are provided to validate our proposed procedure.

2 Extracting Unsatisfiable Core

In this section, we describe the theory behind our procedure for extracting unsatisfiable core from an unsatisfiable Boolean formula. The idea of the procedure is as follow. Given a Boolean formula, if a SAT solver based on Davis Logemann Loveland (DLL) procedure [4] can prove that it is unsatisfiable, then we are able to extract a resolution sequence from the proof such that an empty clause can be generated by the sequence of resolutions on the original clauses in the formula. The set of original clauses that are involved in the resolution sequence can be regarded as a *minimal* set of clauses that are necessary for the unsatisfiability of the original formula *for that particular proof*. This subset of clauses could be much smaller than the original formula, and can be regarded as a good unsatisfiable core. In order to further improve the size of the core, such process can be carried on iteratively on the unsatisfiable cores extracted from the previous iteration. Depending on the resource limitation and requirements for the quality of the core, we can either run the procedure for a fixed number of iterations or continue until it converge (i.e. the extracted core cannot be shrunk further by the procedure).

In Section 2.1, we discuss how to extract a resolution sequence from a DLL SAT solver's solving process. In Section 2.2, we explain how to analyze the resolution sequence to find the minimal unsatisfiable core with regard to that resolution sequence. In Section 2.3, we discuss some implementation issues.

2.1 Learning in DLL Search as a Resolution Process

In this subsection, we briefly discuss how we can regard a proof of unsatisfiability produced by SAT solvers (e.g. [8] [9]) based on DLL search with learning as a resolution process. Here we will briefly overview the learning process in such SAT solvers. We assume the readers are familiar with modern DLL SAT solving algorithms.

DLL procedure is a search and backtracking algorithm. When a conflict occurs during the search, a conflict analysis procedure will be invoked to analyze the conflict and bring the search to a new space. During the conflict analysis, the knowledge of failure may be recorded as clauses to prevent the solver making the same mistake again. This learning process can be formulated as a resolution process [15]. The pseudo code for conflict analysis is shown in Figure 1. At the beginning, the function checks if the current decision level is already 0. In that case, the function will return -1, indicating that there is no way to resolve the conflict and the formula is unsatisfiable. We will call the conflicting clause encountered at decision level 0 the *final conflicting clause*. The final conflicting clauses only contain value 0 literals assigned at decision level 0. If current decision level is not 0, iteratively, the solver resolves the conflicting clause with the *antecedent* clause of a variable in the clause. The antecedent clause of a variable is the unit clause that implies the variable. Function `choose_literal()` will choose a literal in the clause that is assigned *last*. Function `resolve(cl1,cl2,var)` will return a clause that has all the literals appearing in $cl1$ and $cl2$ except for the

literals corresponding to *var*. Notice that the conflicting clause has all literals evaluating to 0, and the antecedent clause of a variable has all but one literal evaluating to 0 (since it is a unit clause) and the remaining literal evaluates to 1. Therefore the resulting resolvent clause is still a conflicting clause because all its literals evaluate to 0, and the resolution process can continue iteratively.

```
analyze_conflict()
{
    if (current_dlevel()==0)
        return -1;
    cl = find_conflicting_clause();
    do
    {
        lit = choose_literal(cl);
        var = variable_of_literal( lit );
        ante = antecedent( var );
        cl = resolve(cl, ante, var);
    } while (!stop_criterion_met(cl));
    add_clause_to_database(cl);
    back_dl = clause_asserting_level(cl);
    return back_dl;
}
```

Fig. 1. Pseudo Code for Conflict Analysis

The iterative resolution will stop if the resulting clause is an *asserting* clause. An asserting clause is a clause with all 0 literals, among them only one literal is at the current decision level and all the others are assigned at a decision level less than the current. After backtracking, the clause will be a unit clause and this literal will be forced to assume the opposite value, thus bringing the search to a new space. This flipped variable will assume the highest decision level of the rest of the literals in the asserting clause. We call this decision level the *asserting level*. The solver will backtrack to the asserting level determined by function `clause_asserting_level()` and continue search. This process continues until the solver finds a solution (if the instance is satisfiable) or a final conflicting clause (if the instance is unsatisfiable).

If the instance is unsatisfiable, we will encounter a final conflicting clause in the solving process. We can generate an empty clause by resolve the final conflicting clause with antecedent of the variables that appear in it. Because all variables appearing in the final conflicting clause are assigned at decision level 0, and in turn all their antecedent clauses contain variables assigned at decision level 0, therefore, an empty clause can always be generated if we choose the clauses to be resolved in proper order. Due to space limit, we will not describe the procedure here. For a detailed description of the resolution process, we refer the readers to [15].

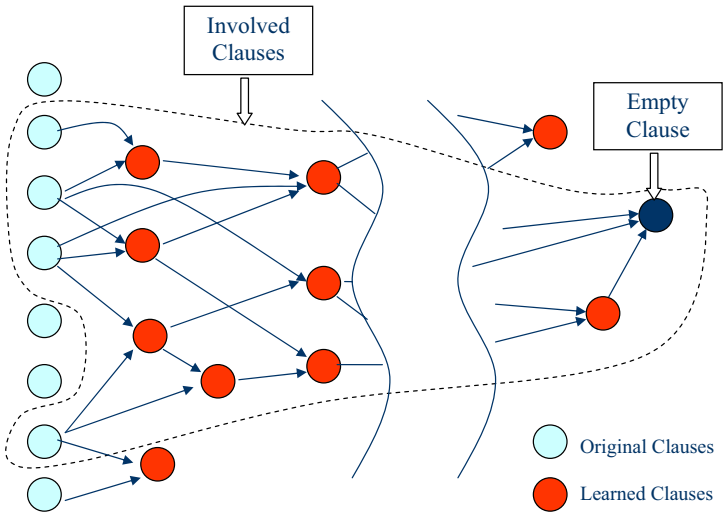


Fig. 2. The Resolution Graph

2.2 Resolution Graph

In last subsection, we describe how a DLL SAT solver's solving process can be regarded as a resolution process to generate an empty clause. We formalize the idea in the concept of a *resolution graph*. An example of resolution graph is shown in Figure 2. A resolution graph is a directed acyclic graph (DAG) that represents a SAT solving process of a Boolean formula. Each node in the graph represents a clause. The root nodes are the original clauses in the Boolean formula. The internal nodes represent the learned clauses generated during the solving process. The edges in the resolution graph represents resolution. If an edge from node b to node a appears in the resolution graph, we say that node b (or the clauses represented by b) is a *resolve source* of node a (or the clause represented by a). A clause represented by an internal node is obtained by resolving all its resolve sources. In conflict analysis procedure depicted in Figure 1, the resolve sources of the learned clause (the clause added to the database by `add_clause_to_database(cl)`) are all the clauses that corresponds to `ante` in the pseudo code together with the conflicting clause obtained by `find_conflicting_clause()`.

As we mentioned in the last subsection, if the Boolean formula is proven to be unsatisfiable, there will be a resolution process to generate an empty clause. The empty clause is shown in Figure 2. The resolve sources of the empty clause are the final conflicting clause together with the antecedent clauses of variables assigned at decision level 0. This resolution graph represents a proof of the Boolean formula being unsatisfiable. All clauses that are not in the transitive fan-in cone of the empty clause in the resolution graph are not needed to construct this particular proof. Deleting them will not invalidate the proof of unsatisfiability. More precisely, the root nodes (original clauses) in the transitive fan-in cone of

the empty clause is a *minimal* subset of the original clauses that are needed for this particular proof. This subset of clauses is a unsatisfiable core of the original formula.

The resolution graph can be easily recorded as a trace file during the SAT solving process, and a simple traversal of the graph can tell what clauses are in the transitive fan-in cone of the empty clause. Therefore, if a Boolean formula is proven to be unsatisfiable by a SAT solver, we can generate an unsatisfiable core from the unsatisfiable proof. The unsatisfiable core of a formula is by itself an unsatisfiable Boolean formula. Therefore, we can run a SAT solver on it and produce another unsatisfiable proof, and we can generate an unsatisfiable core from the proof. This process can be carried out iteratively to improve the size of the final result.

We want to point out that the process discussed here is not affected by the restart technique [6] often employed by modern SAT solvers. Restart throws away the current search tree and begins a fresh search periodically. This operation will not invalidate the facts that learned clauses are resolved from original clauses and/or a formula is proven unsatisfiable only if the solver finds a final conflicting clause. Therefore, all the discussions in this paper are valid regardless whether the SAT solver employ restart or not. However, we do require that the solver's reasoning procedure can be regarded as a resolution process in order for this procedure to work. If the solver employs reasoning techniques other than unit implication (e.g. BDD [3] or Stålmarck's algorithms [11]), the procedure discussed in this paper may not work.

2.3 Implementation Issues

The implementation of the ideas discussed in the previous subsection is relatively straight forward. It is a simple graph traversal algorithm to calculate the fan-in cone of a directed acyclic graph. However, there are a couple of points we want to mention here that are important for efficient implementation. The resolution graph is dumped to the hard disk as a trace file by the SAT solver during the solving process. The graph can be very large for hard SAT instances and may not fit in the main memory of the computer. Therefore, the graph traversal has to be carried out on hard disk. Fortunately, the graph stored on the hard disk is already topologically ordered because when we generate a clause by resolution, all its resolve sources must have already been generated. Therefore, if we dump the resolve sources of clauses to the trace file in the same order as we generate them, the nodes in the file will be topologically ordered. However, to find the transitive fan-in of a node in a graph, we need to traverse the graph in *reversed* topological order. Access a hard disk file in the reverse order as it is stored is very inefficient. Therefore, we need to reverse it first before the traversal. This can be achieved efficiently by using a memory buffer.

A possible failure for marking the transitive fan-ins of a node in a DAG is that during the traversal, we need a storage to indicate whether a yet-to-be-visited node in the graph is in the transitive fan-in or not. We use a hash table for this storage. In theory, the hash table may has the same number of entries as

Instance Name	Num. Clauses	Num. Vars	Proof Time(s)	Trace Overhead	Trace Size(KB)	Num. Nodes	Num. Edges
2dlx_cc_mc_ex_bp_f	41704	4524	3.3	11.89%	1261	9554	184123
bw_large.d	122412	5886	5.9	9.12%	1367	7140	182127
c5315	15024	5399	22	10.45%	11337	50298	1951816
too_largefs3w8v262	50216	2946	40.6	7.68%	8866	91691	1227840
c7552	20423	7651	64.4	8.76%	24327	100487	4068238
5pipe_5_ooo	240892	10113	118.8	4.51%	17466	79770	2434924
barrel9	36606	8903	238.2	4.51%	19656	121071	3058428
longmult12	18645	5974	296.7	6.17%	102397	131649	18340965
9vliw_bp_mc	179492	19148	376	4.26%	39538	255603	5497009
6pipe_6_ooo	545612	17064	1252.4	3.39%	151858	462135	21497545
6pipe	394739	15469	4106.7	2.77%	493655	1327373	69102904
7pipe	751118	23910	13672.8	1.68%	736053	2613927	93903489

Table 1. Statistics of SAT Instances and Resolution Graphs

the number of nodes in the unvisited graph, which in turn can be as large as the whole graph. Therefore, theoretically the algorithm may cause memory overflow for extremely large graphs because the hash table may not fit in memory. In practice, we found that the maximum number of hash table entries encountered during the traversal is rarely much larger than the number of clauses in the final unsatisfiable cores. Therefore, our procedure in practice can find unsatisfiable cores for arbitrarily large proofs generated by a SAT solver.

3 Experimental Results

We implemented the idea discussed in this paper as an unsatisfiable core extractor. We modified the SAT solver zchaff [14] to produce trace file representing resolution graph of an unsatisfiable proof. We want to point out that similar modification can be implemented on many modern SAT solvers based on DLL (e.g. GRASP [8] and BerkMin [5]) with little effort. Therefore, as long as we can prove a Boolean formula to be unsatisfiable using these SAT solvers, we can extract an unsatisfiable core from the formula. The experiments are carried out on a PIII 1133Mhz machine with 1G memory. The benchmarks we use are some relatively large unsatisfiable instances that are commonly used for SAT solver benchmarking. They include instances from applications such as microprocessor verification [13] (9vliw, 2dlx, 5pipe, 6pipe, 7pipe), bounded model checking [1] (longmult, barrel), combinational equivalence checking (c7225, c5135), FPGA routing [10] (too_large) as well as AI planning [7] (unsat/bw_large).

The statistics of the instances and the resolution graph generated are shown in Table 1. Besides the number of variables and clauses for the formula, we also show the run time for (unmodified) SAT solver zchaff to prove the formula unsatisfiable. In the fifth column of Table 1 we show the overhead to record the

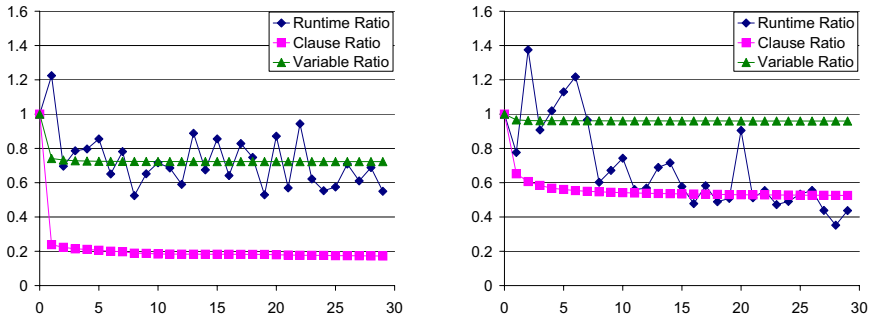
Instance Name	First Iteration				30 Iterations/Fixed Point		
	Num. Clauses	Num. Variables	Hash Size	Extract Time	Num. Clauses	Num. Variables	Num. Itrs.
2dlx_cc_mc_ex_bp_f	11169	3145	11169	0.85	8038	3070	26
bw_large.d	8151	3107	8151	1.54	1364	769	30
c5315	14336	5399	14611	2.64	14289	5399	3
too_largefs3w8v262	10060	2946	11971	2.65	4473	645	30
c7552	19912	7651	23651	5.37	19798	7651	9
5pipe_5_ooo	57515	7494	57521	6.62	41499	7312	30
barrel9	23870	8604	24039	4.66	19238	8543	30
longmult12	10727	4532	26894	21.31	9524	4252	7
9vliw_bp_mc	66458	16737	66458	10.68	36840	16099	30
6pipe_6_ooo	180559	12975	180561	37.14	109369	12308	30
6pipe	126469	13156	126469	99.96	73681	13065	30
7pipe	221070	20188	221079	152.71	133211	20076	30

Table 2. Statistics of Unsatisfiable Core Extracted

resolution graph on file as a percentage of the original SAT solving time. We also show the file size of the resolution graph as well as the number of nodes and edges stored in the resolution graph file (node counts exclude the original clauses and the empty clause, edge counts exclude the incoming edges to the empty clause). As we can see from the table, the resolution graph is quite big for difficult SAT instances. We also notice that generating trace file is inexpensive, especially so for hard benchmarks.

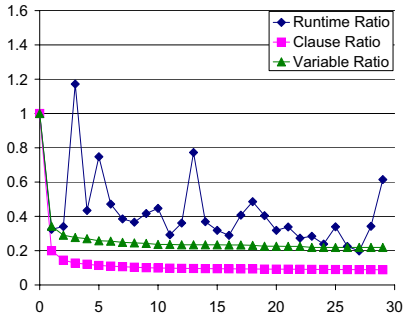
In Table 2, we show the statistics of extracting the unsatisfiable cores from the trace files. The "First Iteration" section show the resulting unsatisfiable core extracted in the first iteration of the core extraction procedure. We show the size of the extracted core as well as run time for the core extraction. Moreover, we show the maximum hash table size for storing unvisited nodes during the traversal, as described in Section 2.3. Under the column "30 Iterations/Fixed Point". We measured up to 30 iterations of core extraction, and report the data. If the "iteration" number is smaller than 30, it means that after certain iterations, our procedure cannot reduce the size of the core further.

Comparing Table 2 with Table 1, we can see that the core extracted can be much smaller than the original formula in the case of bw_large.d or almost the same size as in the case of c7552 and c5315. In the case of bw_large.d, the instance is obtained from SAT planning. A small core means that the reason for unfeasible scheduling is localized. In the case of c7552 and c5315, the instances are obtained from combinational equivalence checking. A large core means that the two circuits being checked for equivalence do not contain much redundancy. We also find that the core extraction is very fast compared with the SAT solving. As we mentioned in Section 2.3, our experiment shows that the maximum hash table sizes are barely larger than the number of clauses in the unsatisfiable cores

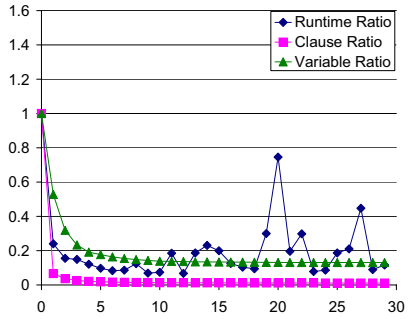


(a) 5pipe_5_ooo

(b) barrel9



(c) too_largefs3w8v262



(d) bw_large.d

Fig. 3. Statistics of Extracted Core as a Ratio of the Original Instances

in all test cases. Comparing the 30 iteration number with the first iteration number, we see that the core size is often reduced if we perform the procedure iteratively, but usually the gains for core size reduction are not as large as the first iteration.

In Figure 3, we show the statistics of the core extractions during the iterations for four representative instances. The three lines represent the zchaff proofing time for the instance, the number of clauses and the number of variables of the extracted cores as a ratio of the original Boolean formula. The X-axis are the number of iterations, the Y-axis are the ratios. From the figure, we find that the size of the instance reduce dramatically in the first iteration, but not by much during the following iterations. The run times for proving the cores to be unsatisfiable are often reduced compared with the original formula with a trend similar to the reduction in size.

Benchmark Instance	Original #Cls.	Extracted #Cls.	#Extract Iterations	Minimal #Cls.	Clause Ratio
2dlx_cc_mc_ex_bp_f	41704	8036	26	7882	1.020
bw_large.d	122412	1352	35	1225	1.104
too_largefs3w8v262	50416	4464	32	3765	1.186

Table 3. The Quality of Unsatisfiable Cores Extracted

As shown by previous results, the procedure described can often extract much smaller unsatisfiable cores from an unsatisfiable formula. However, the cores extracted are minimal only with regard to the particular proofs produced by the SAT solver. There is no guarantee of *minimal* (i.e. deleting any clause from the core will render the core satisfiable) or *minimum* (i.e. the core is the smallest among all unsatisfiable cores for the Boolean formula). To estimate the quality of the cores extracted by the proposed procedure, we choose several easier instances from our benchmarks for the following experiment. Start from the smallest unsatisfiable core the procedure can produce (i.e. further iteration can not reduce the core size), we delete clauses one by one from it until deleting any remaining clause will make the instance satisfiable. Now the resulting formula is a *minimal* unsatisfiable core for the original formula. We compare the minimal formula with the extracted formula by our procedure in Table 3. It shows the original sizes of the instance, number of clauses of the cores our procedure can extract, number of iterations our procedure takes to converge, and the minimal core that can be found by a iterative clause deletion procedure. It also shows the ratio of the minimal core to the extracted core.

As we can see from Table 3, for the three benchmarks tested, the cores extracted by the procedure are about 2% to 20% larger than the minimal cores found by an iterative process. This shows that the proposed procedure is reasonably good for extracting small cores. Even though the iterative method can find minimal cores, it takes much longer to finish (essentially needs to run a SAT solver on the instance for iterations equal to the number of clauses in the formula). We do not have a computationally efficient way to find the *minimum* unsatisfiable core since enumerating all combinations of the clauses in the formula is clearly intractable. Therefore, we cannot compare the size of the extracted cores with the size of the minimum cores of the instances.

4 Conclusion and Future Work

In this paper, we discuss a practical method to extract a small unsatisfiable core from an unsatisfiable Boolean formula. The procedure we proposed is fully automatic and efficient enough for SAT instances generated from real world applications. We implemented and experimentally evaluate the procedure and the results are encouraging.

As for future work, we are still not able to test the absolute quality of the procedure. For example, we are not able to evaluate the closeness of the ex-

tracted core to the *minimum* sized core of the formula. Even test the *minimality* of the resulting core when the iterative procedure converge is computationally expensive for difficult instances. In our procedure the SAT solver is unaware of the core extraction process. It is an open questions whether it is possible to tune the SAT solver to generate proofs that involve small number of original clauses.

References

1. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS*, 1999.
2. R. Bruni and A. Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, June 2001.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 8(35):677–691, 1986.
4. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
5. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.
6. Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.
7. H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.
8. João P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
10. G. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Routing complex fpgas via search-based boolean sat. In *Proceedings of International Symposium on FPGAs*, Feburary 1999.
11. Mary Sheeran and Gunnar Stålmark. A tutorial on stålmark's proof procedure for propositional logic. In *Volume 152 of Lecture Notes in Computer Science*, pages 82–99. Springer Verlag, 1998.
12. Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002.
13. M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 226–231, June 2001.
14. Lintao Zhang. Zchaff sat solver. http://www.ee.princeton.edu/ chaff/zchaff.php, 2000.
15. Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of Design and Test Europe(DATE'03)*, 2003.