Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications

Lintao Zhang
Department of Electrical Engineering
Princeton University
lintaoz@ee.princeton.edu

Sharad Malik
Department of Electrical Engineering
Princeton University
sharad@ee.princeton.edu

ABSTRACT

As the use of SAT solvers as core engines in EDA applications grows, it becomes increasingly important to validate their correctness. In this paper, we describe the implementation of an independent resolution-based checking procedure that can check the validity of unsatisfiable claims produced by the SAT solver zchaff. We examine the practical implementation issues of such a checker and describe two implementations with different pros and cons. Experimental results show low overhead for the checking process. Our checker can work with many other modern SAT solvers with minor modifications, and it can provide information for debugging when checking fails. Finally we describe additional results that can be obtained by the validation process and briefly discuss their applications.

1. Introduction and Previous Work

Given a propositional logic formula, the question whether it is satisfiable (i.e. exists a variable assignment that makes the formula evaluates to *true*) is called the Boolean Satisfiability problem, or SAT. SAT is a well-known NP-Complete problem. Many practical EDA problems such as test pattern generation, combinational equivalence checking, microprocessor verification [1], bounded model checking [2] and FPGA routing [3] can be formulated as SAT instances and solved by SAT solvers.

Recent advances in SAT solving algorithms (learning and non-chronological backtracking e.g. [4], random restart [5]) and efficient implementation techniques (e.g. fast implication engine [6]) have dramatically improved the efficiency and capacity of state-of-the-art SAT solving algorithms. SAT solvers have quickly become a viable deduction engine for industrial strength applications (e.g. [7]). As these applications are often mission critical, it is very important to ensure that the results provided by their SAT engines are correct.

Typically, a SAT solver takes a formula as input and produces an output that claims the formula to be either satisfiable or unsatisfiable. A formula is satisfiable if there exists a variable assignment that makes the formula evaluate to *true*; and is unsatisfiable if no such assignment exists. When the solver claims satisfiability it is usually possible for the solver to provide the satisfying solution with very little overhead. An independent program can take this and verify that it indeed satisfies the formula. The NP-Completeness of SAT guarantees that such a check takes polynomial time in the size of the SAT instance – in fact linear time for Conjunctive Normal Form (CNF) representations used in practice. On the other hand, when a SAT solver claims unsatisfiability, it is usually not trivial for an independent checker to verify the correctness of that claim.

It can be shown that to prove a formula in CNF to be unsatisfiable, we only need to show that an empty clause can be generated from a sequence of resolutions among the original clauses. The classic Davis-Putnam (DP) [8] algorithm is based on this. However, this algorithm is hard to use in practice due to prohibitive space requirements, and over the years has given way to search algorithms based on the Davis-Logemann-Loveland (DLL) algorithm [9]. In this paper, we describe how a checker for a DLL based solver can use the resolution based proof, even though the solver itself is not resolution based in the DP sense. During the solution process, the SAT solver produces a trace. The checker can take the trace together with the original CNF formula as inputs and verify that there indeed exists a resolution sequence to generate an empty clause from the original clauses. If the solver claims that the instance is unsatisfiable but the checker cannot construct an empty clause, then a bug exists in the solver.

The ability to produce checkable proofs in automated reasoning tools is not new. Some theorem provers (e.g. [10]) and model checkers (e.g. [11]) have this ability. Verifying SAT solvers has also been addressed before. In [12], the author mentions that one of the requirements for Stålmarck's SAT solver is the ability to provide a trace that can be independently checked. However, no details are provided for the checking procedure. That SAT solver is based on proprietary algorithms, so the checking procedure may not be applicable to contemporary DLL based solvers. In [13], the authors provide a SAT procedure that can produce easily checkable proofs. However, it is not obvious that the method employed can be *easily* adapted to current state-of-the-art solvers. Moreover, in that work an independent checker is not addressed, thus no evaluation of the actual feasibility of checking is provided. To the best of our knowledge, the checker described in this paper is the first for this purpose and that can easily check the correctness of many existing state-of-the-art SAT solvers with little modification.

The checker imposes some requirements on the solver for the generated trace to be sufficient. In particular, we require that the solver be DLL based, and use *assertion* for backtracking. These requirements will be discussed in later sections. Many widely used state-of-the-art SAT solvers such as GRASP [4], Chaff [6], Berkmin [14] satisfy the requirements and can be easily modified to produce the trace to be checked. In this paper, we focus our discussion and experimentation on the SAT solver zchaff, which is an implementation of the Chaff [6] algorithm, but the same discussion is applicable to other similar SAT solvers.

2. The Boolean SAT solver Chaff

In this section, we describe the algorithms used by Chaff and prove its correctness. This forms the foundation of our checking procedure described in the next sections.

2.1 The Algorithm

Like most SAT solvers, Chaff requires that the input formula be in Conjunctive Normal Form (CNF). To make a CNF formula satisfiable, each clause should be satisfied. If a clause contains only one *free* (i.e. not assigned a value) literal and all the other literals are assigned the value *false*, then the free literal must be assigned the value *true*. Such clauses are called *unit clauses* and the free literal is called the *unit literal*. Since a unit literal is forced to be true, we say that it is *implied*. The unit clause is called the *antecedent* of the variable corresponding to the implied literal. The process of iteratively assigning all unit literals with the value *true* till no unit clause left is called *Boolean Constraint Propagation (BCP)*.

```
if (preprocess()==CONFLICT)
  return UNSATISFIABLE;
while(1) {
  if (decide_next_branch()) {       //Branching
    while(deduce()==CONFLICT) {     //Deducing
      blevel = analyze_conflict();//Learning
      if (blevel < 0)
        return UNSATISFIABLE;
      else back_track(blevel); //Backtracking
    }
  }
  else                   //no free variables left
    return SATISFIABLE;
}
```

Fig. 1. Top-level algorithm of SAT solver Chaff

The top-level algorithm of Chaff is shown in Figure 1. It begins with all variables being unassigned. Function `preprocess()` makes some deductions before any decision is made. The preprocessor will perform BCP, and if during BCP a clause has all its literals assigned the value *false*, we say that a *conflict* occurred and this clause is said to be *conflicting*. In that case, the preprocessor returns with value `CONFLICT`. Otherwise, the main loop begins. The solver chooses a free variable to branch on by assigning it a value. This is called a *decision* and is performed by `decide_next_branch()` in Figure 1. In Chaff (as well as most solvers based on DLL), each decision has a *decision level* associated with it, with the first decision at level 1. The variables that are assigned during preprocessing (before any decision) will have decision level 0. After a decision variable is assigned a value, some clauses may become unit and function `deduce()` performs BCP. Variables that are assigned after a branch during BCP assume the same decision level as the decision variable. An important invariant is that a non-free, non-decision variable will always have an antecedent, and its decision level will always equal the highest decision level of the other variables in its antecedent clause. If after BCP no conflict occurs, then the solver will choose another free variable to branch on and increment the decision level. Function `decide_next_branch()` will return true if it is possible to find a variable to branch on. Otherwise, all variables must have been assigned a value. As no conflicting clause exists, the formula must be satisfiable.

If there exists a conflicting clause during BCP, the function `analyze_conflict()` is called to analyze the reason for the conflict, perform learning, and find out the decision level to backtrack to. If conflict analysis reports that the solver needs to backtrack to decision level -1 to resolve the conflict, it implies that the formula has conflicts even without any branching, so the solver will claim the problem to be unsatisfiable.

Conflict analysis and determining the backtracking level is crucial to the completeness of the solver, so we will discuss it here in a little more detail. The conflict analysis is based on an operation called *resolution*. Two clauses can be resolved to generate a *resolvent* clause as long as there is one and only one variable that appears in both clauses in different phases. The resolvent clause takes the disjunction of the remaining literals in both clauses. An example of resolution is:

$$(x + y)\ (y' + z) \Rightarrow (x+z)$$

with the third clause being the resolvent of the first two. The resolvent is redundant with respect to the original clauses. Therefore, we can generate clauses from original clause database by resolution and add them back to the original clause database without changing the satisfiability of the formula.

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

Fig. 2. Generating Learned Clause by Resolution

The pseudo-code for function `analyze_conflict()` is given in Figure 2. At the beginning, the function checks if the current decision level is already 0. In that case, the function will return -1, declaring that there is no way to resolve the conflict and the formula is unsatisfiable. Otherwise, iteratively, the procedure resolves the conflicting clause with the antecedent clause of a variable in it. Function `choose_literal()` will choose a literal in reverse chronological order, meaning that it will choose the 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 *false*, and the antecedent clause has all but one literal evaluating to *false* and the remaining one literal evaluates to *true* (since it is a unit clause). Therefore, one and only one variable appears in both clauses with different phases and the two clauses can be resolved. The resolvent clause is still a conflicting clause because all its literals evaluate to *false*, and the resolution process can continue iteratively.

The iterative resolution will stop if the resulting clause meets a stop criterion. The stop criterion is that the resulting clause be an *asserting clause*. An asserting clause is a clause with all *false* literals, among them only one literal is at the current decision level and all the others are assigned at decision levels less than current. After backtrack to a level less than the current one, the clause will become 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*. Function `backtrack()`

will undo all the variable assignments between the current decision level and the asserting level and reduce the current decision level to the asserting level. We call such a backtracking scheme *assertion based backtracking*. As we will see in the following sections, this backtracking scheme is key to the checker we describe in this paper.

The resulting clauses from the resolution can be optionally added to the clause database. This mechanism is called *learning*. Learning is not required for the correctness or completeness of the algorithm. Experiments show that learning can often help prune future search space and reduce solution time, therefore it is always employed in modern SAT solvers (e.g. [4] [6] [14]). Learned clauses can also be deleted in the future if necessary (e.g. [14]). Regardless of whether the solver employs learning or not, the clauses that are antecedents of currently assigned variables should always be kept by the solver because they may be used in the future resolution process.

2.2 The Correctness of the Chaff Algorithm

The algorithm is correct if given enough run time the solver can always determine the correct satisfiability of the input formula. Our proof uses the following Lemma.

Lemma. A Boolean propositional formula in CNF is unsatisfiable if it is possible to generate an empty clause by resolution from the original clauses.

Proof. Clauses generated from resolution are redundant and can be added back to the original clause database. If a CNF formula has an empty clause in it, then it is unsatisfiable. ■

In addition we use the following three propositions.

Proposition 1. Given enough run time, the algorithm described in Section 2.1 will always terminate.

Proof. We use $k(l)$ to denote the number of variables assigned at decision level l in a certain state during the search. Suppose the input formula contains n variables. Because at least one variable is assigned at each decision level except decision level 0, $\forall l,\ 0 \le l \le n,\ k(l) \le n;$ and $\Sigma_l\, k(l) \le n$. Consider function f:

$$f = \sum_{l=0}^{n} \frac{k(l)}{(n+1)^l}$$

The function is constructed such that for two sets of different variable assignment states α and β of the same propositional formula, the value $f_\alpha > f_\beta$ if and only if

$\exists d,\ 0 \le d < n,\ k_\alpha(d) > k_\beta(d);$ *For any* l *such that* $0 \le l < d,\ k_\alpha(l) = k_\beta(l)$

Essentially this function biases the sum towards the number of variables assigned at lower decision levels. The value of f monotonically increases as the search progresses. This is obvious when no conflict exists, because the solver will keep assigning variables at the highest decision level. When a conflict occurs, assertion based backtracking moves an assigned variable at the current decision level to the asserting level, which is smaller than the current decision level. Therefore, the value of f still increases compared with its value before conflict analysis.

Because function f can only take a finite number of different values for a given SAT instance, the algorithm is guaranteed to terminate after sufficient run time. ■

Notice that in our proof for termination, we do not require the solver to keep all learned clauses; the value of function f will monotonically increase regardless whether any learned clauses are added or deleted. Therefore, contrary to common belief, deleting learned clauses cannot cause the solving process to loop indefinitely. On the other hand, with restarts [5] it is actually possible for the solver to loop indefinitely because after each restart, all the variables are unassigned, and the value of f decreases. Therefore, it is important for the solver to increase the restart period as the solving progresses to make sure that the algorithm will terminate.

Proposition 2. If the algorithm described in Section 2.1 returns `SATISFIABLE`, then the formula is satisfiable.

Proof. If all variables are assigned a value and no conflict exists, the formula is obviously satisfied. ■

Proposition 3. If the algorithm described in Section 2.1 returns `UNSATISFIABLE`, then the formula is unsatisfiable.

Proof. We prove this proposition using the Lemma. We will show how to construct an empty clause from the original clauses by resolution.

When the solver returns `UNSATISFIABLE`, the last `analyze_conflict()` function must have encountered a conflicting clause while the current decision level is 0. Starting from that clause, we iteratively do a resolution similar to the `while` loop shown in Figure 2. The only difference is that the stop criterion is changed so that the iteration will stop only if the resulting clause is an empty clause. Because there is no decision variable at decision level 0, every assigned variable must have an antecedent. Since function `choose_literal()` chooses literals in reverse chronological order, no literal can be chosen twice in the iteration and the process will stop after at most n calls to `resolve()`. At that moment, an empty clause is generated. All the clauses involved in the resolution process are either original clauses or learned clauses that are generated by resolution from the original clauses; therefore by the Lemma, the original formula is indeed unsatisfiable. ■

3. A Resolution Based Checker

In Section 2, we prove that the algorithm of Chaff is sound and complete. However, a SAT solver is a complex piece of code, and consequently the implementation may have bugs in it. In fact, during the recent SAT 2002 solver competition [15], quite a few submitted SAT solvers were found to be buggy. Thus, a rigorous checker is needed to validate the solvers. In this section, we describe the checker that is derived from the results of last section. The checker can prove a formula to be unsatisfiable when a SAT solver claims so.

3.1 Producing the Trace for the Checker

For the checker to be able to verify that a SAT instance is indeed unsatisfiable, it only needs to verify that there exists a sequence of resolution among the original clauses that can generate an empty clause. To achieve this, we need to modify the SAT solver to generate a trace that can be used for the resolution checking process of the checker. We assign each clause encountered in the solution process a unique ID. The original clauses have IDs that are agreed to by both the solver and the checker (e.g. the order of appearance in the formula).

The trace is produced by modifying the solver in the following ways:

1. Each time a learned clause is generated, the clause's ID is recorded, together with the clauses that are involved in generating this clause. In Figure 2, these include the clause returned by `find_conflicting_clause()`, as well as all the clauses that correspond to parameter `ante` in the calls to `resolve()`. We will call these clauses the *resolve sources* of the generated clause.
2. When `analyze_conflict()` is called and the current decision level is 0, the solver will record the IDs of the

clauses that are conflicting at that time before returning value -1. We call these clauses the *final conflicting clauses*. In the proof process, we only need one of these clauses to construct a proof, so it is sufficient to record only one ID.

3. When `analyze_conflict()` returns with value -1, before returning with value `UNSATISFIABLE`, the solver will record all the variables that are assigned at decision level 0 together with their values and the IDs of their antecedent clauses in the trace.

These modifications total less than twenty lines of C++ code, and should be very easy for other SAT solvers that are based on the same assertion based backtracking technique such as GRASP [4] and BerkMin [14].

The checker will take the trace together with the original formula and try to generate an empty clause in a manner similar to the description of the proof for Proposition 3 in Section 2.2. However, initially the checker does not have the actual literals of the learned clauses. Instead, from the trace file the checker only knows the resolve sources' IDs that are used to generate each of these learned clauses. Therefore, to do the resolution, the checker needs to use resolution to produce the needed learned clauses first. Essentially the checker creates and traverses the resolution graph, which is a directed acyclic-graph that describes the sequence of resolutions starting from the original clauses at the leaves and ending with the empty clause at the root. There are two different approaches for this traversal: depth first (Section 3.2) and breadth first (Section 3.3).

3.2 The Depth First Approach

The depth first approach for building the learned clauses begins from any one of the final conflicting clauses. It builds the needed clauses by resolution recursively as described in Figure 3 in function `check_depth_first()`.

```
check_depth_first()
{
  id = get_one_final_conf_clause_id();
  cl = recursive_build(id);
  while(!is_empty_clause(cl)){
    lit = choose_literal(cl);
    var = variable_of_literal(lit);
    ante_id = antecedent_id(var);
    ante_cl = recursive_build(ante_id);
    cl = resolve(cl,ante_cl);
  }
  if (error_exist())
    printf("Check Failed");
  else
    printf("Check Succeeded");
}
recursive_build(cl_id) {
  if (is_built(cl_id))
    return actual_clause(cl_id);
  id = get_first_resolve_source(cl_id);
  cl = recursive_build(id);
  id = get_next_resolve_source(cl_id);
  do {
    cl1 = recursive_build(id);
    cl = resolve(cl,cl1);
    id = get_next_resolve_source(cl_id);
  }while(id != NULL);
  associate_clause_with_id(cl, cl_id);
}
```

Fig. 3. A depth-first approach for the checker

If a clause is needed, it will be built on the fly using function `recursive_build()`. The checking for the checker is built in at each of the functions it employs. For example, when `resolve(cl, cl1)` is called, the function should check whether there is one and only one variable appearing in both clauses with different phases; when a variable's antecedent clause is needed, the function should check whether the clause is really the antecedent of the variable (i.e. whether it is a unit clause and whether the unit literal corresponds to the variable). If such checks fail, the solver (or its trace generation) is buggy. The checker can also provide as much information as possible about the failure to help debug the solver.

The advantage of the depth-first approach is that the checker builds only the clauses that are involved in the empty clause generation process. Also, as a by-product, the checker can tell what clauses are needed for this proof of unsatisfiability. This gives us a minimal set of clauses in the original propositional formula that are needed *for this particular proof*. There are several applications that need small unsatisfiable subsets of an unsatisfiable instance (e.g. [16]), where this can be used.

The disadvantage of the depth-first approach is that in order to make the recursive function efficient, the checker needs to read in the entire trace file into main memory. This can be quite large and possibly not fit in memory. Therefore, for some long proofs, the checker may not be able to complete the checking.

3.3 The Breadth-First Approach

A breadth-first approach starting from the leaves can avoid keeping the entire trace file in memory as the depth-first solver does. The checker traverses the learned clauses in the order that they are generated in the solving process, which is the same order as they appear in the trace file. When generating the final empty clause, all its resolution sources should have all been constructed and available in memory.

However, we will still have the memory blowup problem because the checker will have all the learned clauses in memory before the final empty clause construction begins. It is well known that storing all the learned clauses in memory is often not feasible. Note however that because of the breadth first nature, a clause can be deleted once its use as a resolve source is complete. A first pass through the trace can determine the number of times a clause is used as a resolve source. During the resolution process, the checker tracks the number of times the clause has been used as a resolve source and when its use is complete, the clause can be deleted safely from main memory.

In the actual implementation, the clause's total use count is stored in a temporary file because there is a possibility that even keeping just one counter for each learned clause in main memory is still not feasible. Due to the same reason, we may also need to break the first pass into several passes so that we can count the number of usages of the clauses in one range at a time. The checker is guaranteed to be able to check any proof produced by a SAT solver without danger of running out of memory because during the resolution process, the checker will never keep more clauses in the memory than the SAT solver did when producing the trace. Because the SAT solver did not run out of memory (it finished and claimed the instance to be unsatisfiable), the checker will not run out of memory during the checking process either; assuming we run both programs with the same amount of memory.

4. Experimental Results

In this section we report some experimental results. The experiments are carried out on a PIII 1133Mhz machine with 1G memory. We set the memory limit to 800MB for the checkers. The benchmarks we use are some relatively large unsatisfiable instances that are commonly used for SAT solver

Instance Name	Num. Variables	Orig. Num. Clauses	Num. Learned Clauses	Runtime Trace Off (s)	Runtime Trace On(s)	Trace Gen. Overhead
2dlx_cc_mc_ex_bp_f	4583	41704	9554	3.3	3.7	11.89%
bw_large.d	5886	122412	7140	5.9	6.5	9.12%
c5315	5399	15024	50298	22.0	24.3	10.45%
too_largefs3w8v262	2946	50216	91691	40.6	43.8	7.68%
c7552	7652	20423	100487	64.4	70.0	8.76%
5pipe_5_ooo	10113	240892	79770	118.8	124.2	4.51%
barrel9	8903	36606	121071	238.2	249.0	4.51%
longmult12	5974	18645	131649	296.7	315.1	6.17%
9vliw_bp_mc	20093	179492	255603	376.0	392.0	4.26%
6pipe_6_ooo	17064	545612	462135	1252.4	1294.8	3.39%
6pipe	15800	394739	1327373	4106.7	4220.6	2.77%
7pipe	23910	751118	2613927	13672.8	13902.4	1.68%

Table 1. Statistics of zchaff with trace generation turned on and off

Instance Name	Trace Size (KB)	Depth First				Breath First	
		Num. Cls	Built%	Runtime(s)	Peak Mem (KB)	Runtime(s)	Peak Mem(KB)
2dlx_cc_mc_ex_bp_f	1261	8105	84.83%	0.84	7860	1.30	4652
bw_large.d	1367	5513	77.21%	1.48	8720	2.44	9920
c5315	11337	27516	54.71%	2.80	18108	5.19	3732
too_largefs3w8v262	8866	56193	61.28%	3.79	26752	5.47	6164
c7552	24327	56603	56.33%	6.16	41420	11.44	5976
5pipe_5_ooo	17466	24910	31.23%	6.60	50044	13.29	17936
barrel9	19656	34624	28.60%	4.85	31456	10.46	6752
longmult12	102397	118615	90.10%	25.87	154288	41.22	7488
9vliw_bp_mc	39538	77296	30.24%	12.78	126752	33.81	17724
6pipe_6_ooo	151858	89695	19.41%	38.52	249468	102.67	40136
6pipe	493655	*	*	*	*	301.98	40248
7pipe	736053	*	*	*	*	645.33	62620

Table 2. Statistics for two different checking strategies (* Indicates Memory Out)

benchmarking. They include instances generated from EDA applications such as microprocessor verification [1] (9vliw, 2dlx, 5pipe, 6pipe, 7pipe), bounded model checking [2] (longmult, barrel), FPGA routing [3] (too_largefs3w8v262) combinational equivalence checking (c7225, c5135) as well as a benchmark from AI planning community (bw_large).

We modified the SAT solver zchaff to produce the trace files that are needed for the checker. In all experiments zchaff uses default parameters. Table 1 shows some statistics of the benchmarks and also of the solution process, arranged in increasing order of the run times. It also shows the run times of the solver with trace generating turned off (i.e. exactly the same as the original zchaff) and turned on. From Table 1 we find that the overhead for producing the trace file (1.7% to 12%) is not significant. Also, we notice that the overhead percentage tends to be small for difficult instances.

We show the statistics for the checkers to check the validity of the proofs in Table 2. The column "File Size" shows the actual size of the trace files that are generated by the SAT solver. For the depth-first approach, the column "Num. Cls Built" are the number of clauses that have its literals constructed by function `recursive_build()` in Figure 3. The column "Built%" shows the ratio of the clauses that are built to the total number of learned clauses, which is listed under column "Num. Learned Clauses" in Table 1. As we can see from Table 2, we only need to construct between 19% to 90% learned clauses to check the proof. Moreover, we see that the instances that take a long time to finish often need a smaller percentage of clauses built to check the proofs. A notable exception to this is longmult12, which is derived from a multiplier. The original circuit contains many xor gates. It is well known that xor gates often require long proofs by resolution. For both the depth-first and breadth-first approaches, we show the actual run time to check the proof as well as the peak memory usage. Comparing these two approaches, we find that in all test cases, the depth-first approach is faster by a factor of around 2. However, it requires much more memory than the breadth-first approach and fails on the two hardest instances because it runs out of memory. In contrast, even though the breadth-first approach is slower compared with the depth-first approach, it is able to finish all benchmarks with a relatively small memory footprint.

From Table 2 we find that the actual time needed to check a proof is always significantly smaller compared with the time needed to perform the actual proof. We also find that the trace files produced by the SAT solvers are quite large for hard benchmarks. We want to point out that the format of the trace file we use is not very space-efficient in order to make the trace human readable. It is quite easy to modify the format to emphasize space efficiency and get a 2-3x compaction (e.g. use binary encoding instead of ASCII). By doing so, we also expect the efficiency of the checker to improve as profiling shows that a significant amount of run time for the checker is spent on parsing and translating the trace files.

In Table 3, we show the number of original clauses and variables that are involved in the actual proof by the depth-first approach. The "original" columns show the statistics of the original benchmark. The variable numbers may be a little different from the ones reported in Table 1. Table 1 shows the variable numbers declared by the header of the CNF file, but some of the variables are actually not used in the formula. The "First iteration" column shows the number of original clauses and variables needed for the checker to prove the unsatisfiability. Essentially these numbers show that not all the clauses in the original CNF formula are needed for it to be unsatisfiable. We can use these involved clauses as a new SAT instance. The new instance should still be unsatisfiable. We can feed it back to the SAT solver and iteratively perform the depth-first checking again to determine the number of clauses needed

Benchmark Instance	Original		First Iteration		30 Iterations (or reach fixed point)		
	Num Cls	Num. Vars	Num. Cls	Num. Vars	Num. Cls	Num. Vars	Iteration
2dlx_cc_mc_ex_bp_f	41704	4524	11169	3145	8038	3070	26
bw_large.d	122412	5886	8151	3107	1364	769	30
c5315	15024	5399	14336	5399	14289	5399	3
too_largefs3w8v262	50216	2946	10060	2946	4473	645	30
c7552	20423	7651	19912	7651	19798	7651	9
5pipe_5_ooo	240892	10113	57515	7494	41499	7312	30
barrel9	36606	8903	23870	8604	19238	8543	30
longmult12	18645	5974	10727	4532	9524	4252	7
9vliw_bp_mc	179492	19148	66458	16737	36840	16099	30
6pipe_6_ooo	545612	17064	180559	12975	109369	12308	30

Table. 3. Number of Clauses and Variables involved in the proof

for an unsatisfiability proof for it. After several iterations, the number may reach a fixed point, so that all the clauses are needed for the proof. We measured up to 30 iterations of such a process, and the data is reported under the column "30 iterations (or reach fixed point)". If the "iteration" number is smaller than 30, it means that after a certain number of iterations, all clauses are needed in the proof.

The process described above can be used to determine a small sub-formula (an *unsatisfiable core*) of a CNF formula that is unsatisfiable. This unsatisfiable core is useful in some applications such as debugging software models in the Alloy system [17], FPGA routing [2] and AI planning. For example, in AI planning, a satisfiable solution corresponds to a feasible scheduling. The unsatisfiable core gives the information about why no scheduling is feasible. In FPGA routing, an unsatisfiable instances means that the channels are un-routable. The unsatisfiable core can help the designers concentrate on the reasons (constraints) that are responsible for the routing failure. Notice from Table 3, the instances from both AI planning (bw_large.d) and FPGA routing (too_largefs3w8v262) have small unsatisfiable cores compared with the original formulas.

5. Conclusions and Future Work

In this paper, we describe a checker for Boolean Satisfiability solvers. When a SAT solver proves that a SAT instance is unsatisfiable, the checker can take a trace produced by the SAT solver and check whether the proof is actually valid. The checker uses resolution to generate an empty clause from the original clause database with the information provided by the trace. We discuss two approaches for implementing the checker. Experiments show that both approaches have their relative merits. In all cases, the time needed to check a proof is much less than the SAT solving time.

It is desirable to have a checker that has the advantage of both the depth-first and breadth-first approaches without suffering from their respective shortcomings. Potentially a depth-first algorithm for the graph on disk [18] can provide this.

6. References

[1] M. Velev, and R. Bryant, Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors, *Proc. of the Design Automation Conference,* 2001.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs, *Tools and Algorithms for the Analysis and Construction of Systems (TACAS),* 1999

[3] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar, Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT, *Proc. of Int'l Symposium on Field-Programmable Gate Arrays (FPGA'99)*, Monterey, California, 1999.

[4] J. P. Marques-Silva and K. A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transactions on Computers*, vol. 48, 506-521, 1999.

[5] C. P. Gomes, B. Selman, and H. Kautz, Boosting Combinatorial Search Through Randomization, *Proc. of AAAI-98*, Madison, WI, 1998.

[6] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT Solver, *Proc. of the Design Automation Conference,* July 2001.

[7] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M.Y. Vardi, Benefits of Bounded Model Checking at an Industrial Setting, *Proc. 13th Conf. on Computer Aided Verification,* 2001

[8] M. Davis and H. Putnam, A computing procedure for quantification theory, *Journal of the ACM*, vol. 7, pp. 201-215, 1960.

[9] M. Davis, G. Logemann, and D. Loveland, A machine program for theorem proving, *Communications of ACM*, vol. 5, pp. 394-397, 1962.

[10] A. Stump and D. Dill, Faster Proof Checking in the Edinburgh Logical Framework, *Proc. 18th International Conference on Automated Deduction (CADE-18),* 2002

[11] K. Namjoshi, Certifying Model Checkers*, Proc. 13th Conf. on Computer Aided Verification,* 2001

[12] J. Harrison, Stålmarck's algorithm as a HOL Derived Rule, *Proc. International Conf. on Theorem Proving in Higher Order Logics,* 1996

[13] A. Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution, *7th Int'l Symposium on AI and Mathematics,* 2002

[14] E. Goldberg and Y. Novikov, BerkMin: a Fast and Robust SAT-Solver, *Proc. of Design Automation & Test in Europe,* 2002

[15] SAT 2002 Solver Competition results are available at http://www.satlive.org/SATCompetition/index.jsp

[16] R. Bruni and A. Sassano**,** Restoring Satisfiability or Maintaining Unsatisfiability by finding small Unsatisfiable Subformulae, *Proc. of Theory and Applications of Satisfiability Testing (SAT2001)*, 2001

[17] Daniel Jackson, Alloy: A Lightweight Object Modelling Notation, *ACM Transactions on Software Engineering and Methodology.* April 2002, Vol. 11, No. 2.1

[18] A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian and J. Westbrook, On External Memory Graph Traversal, *11th ACM-SIAM Symposium on Discrete Algorithms,* January 2000