Verifying Propositional Unsatisfiability: Have Your Cake and Eat It Too

Allen Van Gelder

University of California, Santa Cruz CA 95060, USA,
WWW home page: `http://www.cse.ucsc.edu/~avg`

Abstract. The importance of producing a certificate of unsatisfiability is increasingly recognized for high performance propositional satisfiability solvers. This paper reports progress on several fronts. The leading solvers develop a conflict graph as the basis for deriving (or "learning") new clauses. Extracting a resolution derivation from the conflict graph is theoretically straightforward, but it turns out to have some surprising practical pitfalls (as well as the unsurprising problem that resolution proofs can be extremely long). These pitfalls are exposed, solutions are presented, and analyzed for worse cases. Dramatic improvements on industrial benchmarks are demonstrated. Several other certificate formats have been proposed and studied. They are much more reasonable in their space requirements than a full resolution proof, but the verifiers for these formats are beyond any hope of automated verification in their own rights. A compact format called RUP (for Reverse Unit Propagation), an outgrowth of "conflict clause proofs" proposed by Goldberg and Novikov, is described and its relation to resolution proofs is analyzed. Its practical attraction is ease of implementation. Several systems for verification of propositional unsatisfiability are evaluated experimentally. Systems for formal verification of more compact certificate formats are described and experimental comparisons are presented. Future work to improve the support for RUP is outlined.

1 Introduction

With the explosive growth of Sat Modulo Theories (SMT) in the last few years, the focus in propositional SAT solvers is shifting to unsatisfiable formulas, because these are the negated theorems to be proved in many applications. Producing proofs and independently checking them has received limited attention. Two ground-breaking efforts are Goldberg and Novikov [GN03], who built on BerkMin [GN02], and Zhang and Malik [ZM03b,ZM03a], who built on Chaff [MMZ$^+$01]. It is important to get our propositional house in order to provide an adequate foundation for the more sophisticated challenge of producing independently checkable proofs for SMT.

The author has argued elsewhere [VG02] that solvers should be able to produce *easily verifiable* certificates to support claims of unsatisfiability. The gold standard proposed is that the language of certificates should be recognizable in *deterministic log space*, a very low complexity class. Intuitively, an algorithm to recognize a log space language may re-read the input as often as desired, but can only write into working storage consisting of a fixed number of registers, each able to store $O(\log L)$ bits, for inputs of length L.

The rationale for such a stringent requirement is that the buck has to stop somewhere. How are we to trust a "verifier" that is far too complex to be subjected to an automated verification system? And how are we to trust that automated verification system? Eventually, there has to be a verifier that is so elementary that we are satisfied with human inspection.

An explicit resolution proof is one in which each derived clause is stated explicitly, along with the two earlier clauses that were resolved to get the current clause. It is not hard to see that an explicit resolution proof can be recognized with a fixed number of working-storage registers, provided they can store indexes to any point in the input. Thus this language is in deterministic log space.

A detailed specification for an explicit resolution derivation (`%RES`) was used for the verification track of the SAT05 solver competition. We have a verifier for this format (named `checker1`) that is able to handle proofs up to 12 GB on certain available compute servers (the limitation is that 16 GB of swap space plus real memory is needed to process a 12 GB proof). The verifier accepts two binary formats and one ascii format. Specification documents and software are available at: `http://www.cse.ucsc.edu/~avg/ProofChecker/`. Notice that a much more compact format, called *resolution proof trace* (**%RPT**), states the two operands needed for each resolution operation, but does not materialize the clause. This language has little hope for log-space recognition because there is not enough working storage for the verifier to materialize a clause.

For the remainder of this paper, Section 2 discusses pitfalls that were discovered in one existing solver and greatly limited its capability to produce proofs; Section 3 presents an attractive proof format named RUP for Reverse Unit Propagation that is easy-to-implement, general, and compact; Section 4 experimentally compares several approaches for proof generation and checking; and Section 5 shows some additional properties of RUP and outlines how to use them to reap the benefits of RUP without paying the major performance penalty observed in Section 4 (have your cake and eat it too).

2 Pitfalls in Extracting Resolution from a Conflict Graph

Most, if not all, leading SAT solvers use a *conflict graph* data structure to infer *conflict clauses*. Figure 1 illustrates a conflict graph. The notation varies from other papers [ZMMM01,BKS04] to better reflect the actual data structures used by the programs. Each graph vertex is associated with a different literal, no complementary literals appear, and the conflict vertex is associated with the constant *false*, denoted by "$\perp$." The vertex for each implied literal, including *false*, is labeled with an "input" clause, called the *antecedent clause*. This notation agrees more closely with the original presentation . Assumed (guessed) literals, commonly called "decision literals," do not have an antecedent clause.

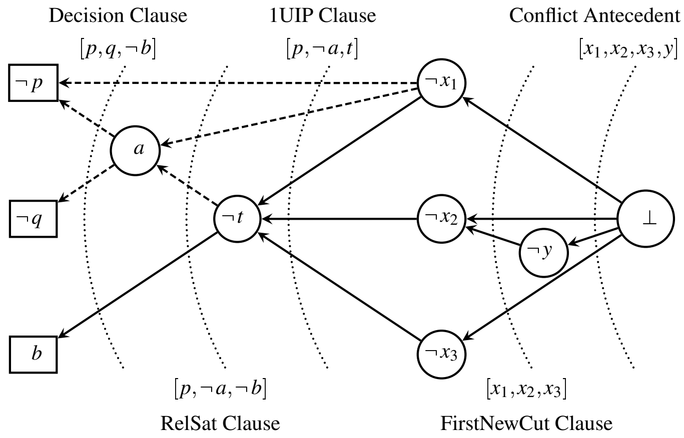

Fig. 1. Conflict graph with several cuts shown.

```
1.     cl = final_conflicting_clause;
2.     while (!is_empty_clause(cl)) {
3.         lit = choose_literal(cl);
4.         var = variable_of_literal(lit);
5.         ante_cl = antecedent(var);
6.         cl = resolve(cl, ante_cl);
       }
```

Fig. 2. Pseudocode to generate resolution proof from conflict graph.

Recent papers have observed the connection between conflict graphs and resolution [GN03,ZM03b,BKS04,VG05]. Of course, given a cut, the antecedent clauses on the conflict side logically imply the conflict clause, which consists of the negations of the reason-side literals adjacent to some vertex on the conflict side (i.e., one or more edges cross the cut to such literals). By the completeness of resolution there must be a resolution derivation of the conflict clause from the antecedent clauses.

The question is how to exploit the structure of the conflict graph to obtain a resolution derivation of the conflict clause. This question is not as simple as it might appear, in view of the fact that the algorithm published by Zhang and Malik, and actually implemented in `zverify_df` (in the `zchaff` distributions), has an exponential worst case. Their algorithm is based on Figure 3 of their paper [ZM03b], the crucial part of which appears in Figure 2. It is important to note on line 1 that `final_conflicting_clause` is *not* a conflict clause. Rather, it denotes the antecedent of $\perp$, the "input" clause that became empty during the unit propagation that generated the conflict graph (see "Conflict Antecedent" in Figure 1). The *conflict clause* is the final value of `cl`, and is empty if the solver was correct. An invariant is that `cl` contains the *negations of* some literals in the conflict graph.

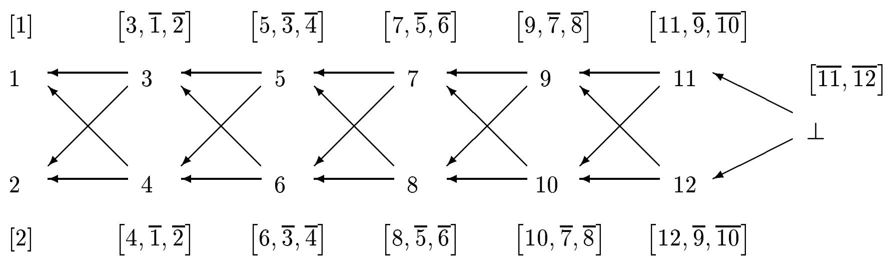

Fig. 3. DAG family with exponential worst case for `zverify_df` as published and distributed. Antecedent clauses are shown in brackets. This graph is for $h = 6$.

Line 3 is implemented two ways in different versions. In one version, the literal chosen is one with minimum *DAG height*, which is the maximum path length to a vertex whose antecedent is a unit clause. In another version, the literal chosen is simply the one with the lowest variable number, essentially an arbitrary choice. The conflict graph family that generates exponential behavior in the size of the conflict graph is the same for both versions. The parameter of this family is h, the DAG height of the *false* vertex. A member with $h = 6$ is shown in Figure 3. The resolution developed by Figure 2 begins as follows, where "(11)" denotes resolution with clashing literal 11:

$$[\overline{11},\overline{12}]\ (11)\ [11,\overline{9},\overline{10}]\ (9)\ [9,\overline{7},\overline{8}]\ (7)\ [7,\overline{5},\overline{6}]\ (5)\ [5,\overline{3},\overline{4}]\ (3)\ [3,\overline{1},\overline{2}]$$
$$(1)\ [1]\ (2)\ [2]\ (4)\ [4,\overline{1},\overline{2}]\ (1)\ [1]\ (2)\ [2]\ (6)\ [6,\overline{3},\overline{4}]\ (3)\ [3,\overline{1},\overline{2}]\ \ldots$$

Literals 1 and 2 will be resolved upon 2^{h-1} times each; literals 3 and 4 will be resolved upon 2^{h-2} times each, etc.; the total number of resolutions is $2\,(2^h - 1)$.

This is a case where theory translates into practice, at least in the case of the arbitrary choice of literal in the version dated 2004.11.15. Table 1 shows the sizes of resolution proofs on some smaller industrial benchmarks, for the original program and a fixed version that uses a better order, which will be described shortly. (Although `zverify_df` does not output the resolution proof, it materializes all the clauses.) More details on the GN03 benchmarks are given in Section 4. Those labeled "`IBM_FV`" are from the "industrial" category of the SAT 2005 Competition (see `http://www.satcompetition.org/2005`).

The Princeton `zchaff` team has been very cooperative, both in providing source codes and answering questions. The purpose of showing this data is not to criticize them in any way. Rather, the purpose is to show that extracting a resolution proof from the conflict graph has the potential for very bad performance. This provides an incentive for developers to look for a verification method that is simple to implement and less disaster-prone.

We now introduce some terminology and notation to study efficient methods of extraction, with worst-case guarantees. We shall use the notation that n is the number of vertices in the conflict graph, m is the number of edges, d is the

Table 1. Resolution proof length inefficiencies. Sizes are mega-literals. "+?" indicates job was killed when size exceeded stated number.

Small GN03		
benchmark	2004.11.15	with fix
5pipe	153	15
5pipe_1_ooo	986	92
5pipe_5_ooo	2320	94
6pipe	169	97
6pipe_6_ooo	3072+?	486
7pipe	358	319
9vliw_bp_mc	308	58
barrel7	2754	5
barrel8	3072+?	73
barrel9	3072+?	80
c3540	393	78
c5315	162	9
c7552	2650	22

IBM_FV SAT 2005		
benchmark	2004.11.15	with fix
01_SAT_dat.k10	3	0.134
07_SAT_dat.k30	3	2.582
07_SAT_dat.k35	3	2.859
18_SAT_dat.k10	174	0.511
18_SAT_dat.k15	102520+?	14.410
1_11_SAT_dat.k10	13	0.338
1_11_SAT_dat.k15	128	2.077
20_SAT_dat.k10	13	0.208
23_SAT_dat.k10	3	0.062
23_SAT_dat.k15	15259+?	3.737
26_SAT_dat.k10	0.036	0.036
2_14_SAT_dat.k10	70	0.857
2_14_SAT_dat.k15	3220	7.188

maximum out-degree of any vertex in the conflict graph, and w is the number of literals in the conflict clause to be derived. Note that the sum of the lengths of the antecedent clauses is m. Thus $(m + w)$ is the combined length of the input and output.

A *linear resolution derivation* is one in which the first clause is an "input" clause, called the *top clause*, and each resolution operation has the previous clause in the derivation as one operand; the second operand may be an "input" clause or an earlier-derived clause of the linear derivation. (We shorten "resolution derivation" to "derivation" when there is no ambiguity.)

An *input derivation* is a linear derivation with the further restriction that the second operand must be an "input" clause; earlier-derived clauses of the same linear derivation are not acceptable. For our purposes, an "input" clause is either a clause of the original formula or a clause that was derived *before* the current linear derivation began—hence the quotes around "input." Note that Figure 2 provides a framework for input derivations using antecedent clauses of the conflict graph as "input" clauses.

Beame *et al.* [BKS04] define a *trivial resolution derivation* (**TVR**) to be an input derivation with the further restriction that no clashing variable occurs in more than one resolution operation. They show (their Proposition 4) that the conflict clause can be derived by a trivial resolution derivation using the antecedent clauses as the "input" clauses, and using the antecedent of the *false* vertex as the top clause. The successful trivial resolution derivation has exactly n resolution operations, but using a correct order is crucial.

To get a worst-case bound on total derivation size of TVR, measured in number of literals, we note that it is possible for the current resolvent to grow by up to $(d - 1)$ literals per resolution for the first n/d steps to a size of $(w + 1 + n\,(d - 1)/d)$, even if the final conflict clause is fairly narrow. Thereafter, it can shrink only one literal per resolution, so the sum of the sizes of all resolvents in the derivation is bounded by $(w + n\,(d-1)/2\,d)\,n$, which can be quadratic in the size of the conflict graph.

The trouble with TVR scheme is that the correct order is not readily accessible from the data structure of the conflict graph. To obtain a valid TVR order, the rule for `choose_literal` on line 3 of Figure 2 should be:

Cut-Crossing Rule: Choose a literal all of whose incoming edges originate from a vertex whose literal has already been resolved upon, or from the *false* vertex (reworded from Beame *et al., op cit.*).

Some other data structure is needed to provide or compute an appropriate order.

The saving grace is that the input for `zverify_df` is set up by the companion solver, such as `zchaff`, which already *has* such a data structure. The solver creates a sequence of "implied" literals in the chronological order in which they entered the conflict graph. When this sequence is available, `zchaff` uses the *reverse* of this order to generate the trace of an input derivation for `zverify_df` to verify [ZM03b]. It is not difficult to see that reverse chronological order satisfies the cut-crossing rule. (A theoretical nitpick is that the sequence might include numerous implied literals that are not in the conflict graph, but have to be looked at anyway, so the *time* is not strictly bounded by the size parameters of the conflict graph.)

As it happens, `zchaff` communicates the successful order to `zverify_df` in its encoding of the "resolve-trace" for all conflict clauses with a *positive* "decision level." (Details are given in Section 4.) Unfortunately, `zchaff` (through 2006) treats decision level zero differently, does not actually create an empty conflict clause, and so the crucial order for decision level zero is *not* in the encoding of the resolve-trace. Instead the final conflict graph itself is encoded in the output. Thus `zverify_df` had an opportunity to go astray.

The "fix" applied by this author to deliver the rightmost column of Table 1 (available in the next `zchaff` release) involved modifying `zchaff` to follow through at decision level zero, mainly using the procedures and data structures already in the program. The fixed program creates an empty conflict clause and encodes its resolution order in the resolve-trace, using the same protocol as for the nonempty conflict clauses at positive decision levels. Then `zverify_df` was modified slightly to expect this additional line. A side benefit is that the final conflict graph is no longer needed in the output, allowing the "resolve-trace" files to decrease in size by as much as a factor of ten, but usually a few percent.

On the benchmarks reported in Section 4 it appears that the overhead for producing the "resolve-trace" is about 11% of the solution time without producing the "resolve-trace." This is an increase from about 4% for the original program, version 2004.11.15. We are unable to quantify the impact on `zverify_df` because the original version did not terminate within a reasonable time in some cases.

As we noted, although TVR uses the minimum number of resolution *operations*, it might produce a derivation of quadratic length. We now describe a different extraction method, that uses more operations, but produces shorter clauses, in most cases. We call this method *pseudo-unit propagation* (**PUP**).

For this method, we define a *pseudo-unit clause on x* at a particular vertex x of the conflict graph to be a clause containing only the literal x and some subset of the literals of the conflict clause being derived. No other literals in the conflict graph are present. Among the literals of the pseudo-unit clause that are also in the conflict clause, *one* might be the negation of the "unique implication point" (**UIP**) literal [MSS99], while the others are literals that were falsified at lower decision levels. The UIP literal might be the assumed literal at the current decision level. At decision level zero, there are no "extra" literals and the pseudo-unit clauses are true unit clauses. In any case, a pseudo-unit clause has at most $(w+1)$ literals.

The extraction method works as follows: Starting at the *false* vertex, perform a depth-first search of the conflict graph (using edge orientations illustrated in Figure 1), as described in several algorithms texts [CLRS01,BVG00]. Only visit vertices that were implied at the current decision level and are *not* the negated UIP literal. At post-order time for vertex x, derive a pseudo-unit clause on x, which we'll call $pup(x)$.

To derive $pup(x)$, suppose the antecedent clause is $ante(x) = [x, \overline{y_1}, \ldots, \overline{y_k}]$. Then for $1 \leq i \leq k$, either $\overline{y_i}$ is in the conflict clause being derived or y_i is a vertex in the conflict graph whose pseudo-unit clause has been derived already. Start a linear derivation with $ante(x)$. For each y_i in the conflict graph, resolve $pup(y_i)$ with the current resolvent in the linear derivation. The final resolvent is $pup(x)$. At the *false* vertex, *false* can be discarded from $pup(false)$, leaving the conflict clause that was to be derived.

Alternatively, if the chronological sequence in which the vertices were implied into the conflict graph is known, the derivation of pseudo-unit clauses can be performed in *forward* chronological order. In this case it closely mimics the solver's unit-clause propagation. In any case, careful memory management is needed to avoid leaving garbage behind.

With reasonable assumptions about data structures already present in the solver, the derivation of the pseudo-unit clause on x can be performed in time proportional to the sum of the lengths intermediate clauses derived on the way to the pseudo-unit clause. Each intermediate clause has at most $(k+w)$ literals, where k is the out-degree of vertex x (i.e., the antecedent clause has $(k+1)$ literals). The sum of these lengths is at most $k\,(w + (k+1)/2)$ (because clause lengths can shrink at most one literal per resolution and the final length is at most $(w+1)$.

For the entire derivation of the conflict clause, analysis with LaGrange multipliers shows that the number of literals in the derivation is maximized when most of the edges are concentrated at as few vertices as possible. In this worst case, the leading term of the total derivation size is $(w + (d+1)/2)\,m$, measured in number of literals. In all cases, exactly m resolutions are performed.

In summary, both the TVR and PUP methods guarantee that total derivation size polynomial in the size of the conflict graph, but have nonlinear worst cases. Their worst cases are incomparable: PUP is lower if $d \ll \sqrt{n}$, but is quite sensitive to d, and is larger for $d \gg \sqrt{n}$. Remembering that n is the number of vertices in the conflict graph and $(d+w+1)$ is the size of the largest antecedent clause, we suspect that $d \gg \sqrt{n}$ when the antecedent clauses include plentiful earlier-derived conflict clauses. Experimental data on industrial benchmarks (not presented in detail) shows that PUP derivations are 60% longer than TVR on average and are longer on about 74% of the benchmarks tested.

The significance of this data and take-home message is: A program that generates resolutions "on-line" during unit clause propagation will do essentially the same resolutions as a PUP after-the-fact system. Both the theoretical and empirical analyses suggest that this produces more verbose resolution derivations (aside from the on-line resolutions that turn out to be unneeded), compared to the after-the-fact TVR method.

3 Reverse Unit Propagation (RUP) Proofs

Reverse Unit Propagation (**RUP**) proofs are based on the idea of *conflict clause proofs* from Goldberg and Novikov [GN03]. A clause $C = [x_1, \ldots, x_k]$ is an *RUP inference* from formula F if adding the unit clauses $[\overline{x_i}]$, for $1 \leq i \leq k$, to F makes the whole formula refutable by unit-clause propagation. An *RUP proof* from an initial formula F_0 is a sequence of clauses C_i, for $i \geq 1$, such that for all i C_i is an RUP inference from F_{i-1}, where $F_j = F_{j-1} \cup \{C_j\}$, for $j \geq 1$. If some C_j is the empty clause, the sequence is called an *RUP refutation*.

Goldberg and Novikov considered only the case where all conflict clauses C_j are added to the initial formula F_0, in chronological order, giving a sequence of formulas, $F_j = F_{j-1} \cup \{C_j\}$. They stated without proof (using different terminology) that C_j is an RUP inference from F_{j-1} for all $j \geq 1$, that is, the sequence of all derived conflict clauses is an RUP proof. (They probably envisioned a PUP proof of the empty clause for each RUP inference, as described in the previous section, and considered the observation to be obvious.) Later, Beame *et al.* [BKS04] proved (again, using different terminology, their Propositions 3 and 4) that a conflict clause defined by any cut in a conflict graph is an RUP inference from the formula consisting of the antecedent clauses in the conflict graph, and moreover, the conflict clause could be derived with a "trivial resolution derivation," as discussed in Section 2.

Our definition generalizes Goldberg and Novikov only to the extent that we do not require the derived clauses to be conflict clauses, i.e., clauses based on a cut of some conflict graph induced by unit-clause propagation in the solver. This opens the door for solvers using other data structures and inference rules to use the RUP format.

We have developed a detailed specification (`fileformat_alt` at the URL in Section 1 and elsewhere) of an RUP refutation, which will be accepted in the verification track of the SAT07 solver competition. The main point of this section is to argue that an RUP refutation can be verified to the same level of confidence as an explicit resolution proof (see Section 1), i.e., in *deterministic log space.*

Digraph reachability cannot be recognized in deterministic log space, let alone unit clause propagation. Therefore, at first blush, verifying an RUP proof is necessarily in a higher complexity class (e.g., P-complete). We claim that our supporting software provides sufficient extra information to permit an RUP proof to be verified in log space.

The system works as follows. The input consists of a formula F_0 and a sequence of clauses C_i, $1 \leq i \leq k$ that is claimed to be an RUP refutation of F_0. One program composes two sequences of extended formulas,

$$\left.\begin{array}{l} F_j = F_{j-1} \cup \{C_j\} \\ G_j = F_{j-1} \cup \{\overline{C_j}\} \end{array}\right\} \qquad \text{for } 1 \leq j \leq k, \tag{1}$$

where $\overline{C_j}$ denotes the set of unit clauses obtained by negating C_j. A second program attempts to refute each G_j for $1 \leq j \leq k$, *using only unit-clause resolution*, and outputs an explicit resolution proof if it is successful; call this proof P_j.

Neither of these programs fits the log-space criterion. But now, our trusted verifier `checker1` is invoked to verify that P_j is a correct refutation of G_j. This removes the need to trust the program that *produced* P_j.

If this whole system checks P_j for $1 \leq j \leq k$ without detecting an error, then the RUP proof of C_k has been verified, subject to correctness of the generated sequences F_j and G_j. If C_k is the empty clause, the RUP refutation of F_0 has been verified.

It remains to verify that the sequences F_j and G_j produced by the first program are what they purport to be, i.e., that they satisfy their defining equations, (1). Of course, we do not even *think* about trusting the program that generated these sequences! It's `csh` and `awk` scripts, for goodness sake! However, given a reasonable encoding of F_0, C_j, F_j, and G_j, it *can* be checked in log space that the sequences F_j and G_j satisfy their defining equations.

Partly as a by-product of being in log space, the checking system is massively parallel: each j can be processed independently. Also, as pointed out by Goldberg and Novikov, by starting the checking at $j = k$ and working backwards, it might be determined that some RUP clauses are not used to derive any later RUP clause (including the empty clause), and such clauses need not be checked. Unfortunately, it is not straightforward to use this optimization in conjunction with parallelization.

4 Experimental Results

For verification to become practical it is crucial to know what magnitude of resources are needed for industrial benchmarks, or other benchmarks of interest. Two ground-breaking papers in this area study compact proof formats: Goldberg and Novikov [GN03] and Zhang and Malik [ZM03b]. This paper provides the first in-depth data on explicit resolution proofs.

The *conflict clause proofs* of Goldberg and Novikov have been discussed in Section 3. The RUP format consists of one line per RUP clause, in ASCII DIMACS format, i.e., 0-terminated.

The *resolve-trace* format reported by Zhang and Malik consists of one ASCII line for each derived conflict clause, in chronological order. That line provides the index for the new clause and lists the earlier clauses, by their indexes, that should be resolved in the order listed to produce the new clause. Thus each line describes one "trivial resolution," as defined in Section 2. The system is implemented as a solver, `zchaff`, and a verifier `zverify_df`. The most recent release as of this writing is 2004.11.15.

A "Special Edition" of `zchaff`, call it `zchaffSE05`, is dated March 2005 and was entered into the verification track of the SAT05 competition. It combines the functionality of `zverify_df` into `zchaff` and writes the full resolution derivation in the binary format specified for SAT05 and identified by a header beginning "`%RESL32`.". (Data produced during SAT05 for `zchaffSE05` is skewed due to the issues discussed in Section 2.) This program *accumulates* all the data that would normally be output on-line for later processing by `zverify_df`. Then, imitating `zverify_df`, it identifies which conflict clauses are used to derive the final level-zero conflict (the unsat core), and only writes resolution derivations for these clauses. Clearly, this was a substantial implementation burden, even starting with `zchaff` and `zverify_df`, and a major purpose of this study is to see if that burden can be lightened by using the RUP format.

As mentioned in Section 2, we made minor modifications for `zchaff` to include the empty clause as the last line, and for `zverify_df` to expect it, obviating the need to also provide the final conflict graph. Corresponding modifications were made to `zchaffSE`. The data reported is for the modified versions, called `zchaffJan07`, `zverify_dfJan07`, and `zchaffSEJan07`, in the tables.

To facilitate studying RUP derivations, we wrote scripts to convert *resolve-trace* files into RUP proofs, line for line. The version of `BerkMin` used by Goldberg and Novikov for their paper is not publicly available.

The benchmarks used are those reported by Goldberg and Novikov, to facilitate comparisons; many are also reported by Zhang and Malik. Please see those papers [GN03,ZM03b] for additional details about them. On some benchmarks the "ooo" in the file name is omitted in some papers, but retained in our tables. Times are normalized for an AMD Opteron 2.6 GHz cpu and are in seconds, unless stated otherwise.

The first question is how proof lengths compare for the various formats. In most cases disk space is more of a limiting factor than time. Table 2 shows the results for the 27 benchmarks used by Goldberg and Novikov. There are wide fluctuations between their number and ours, benchmark by benchmark, but their "conflict clause proofs" and our RUPs are of comparable sizes, overall. We also checked the corresponding numbers in Zhang and Malik, but found no meaningful correlations: apparently `zchaff` underwent extensive tuning since their paper.

However, our full resolution proofs are 100 times shorter than the estimates of Goldberg and Novikov (they had no program to produce such proofs). Some of this effect might be due to the fact that our proofs are already trimmed to an unsatisfiable core [ZM03a], as far as conflict clauses go. However, other studies

Table 2. Proof length comparisons. Sizes are thousands of literals, numbers, variables, or clauses. Ratios are to Full Resolution literals.

GN03	Input		Full Resolution		Resolve-Trace			RUP		
Benchmark	Vars	Cls	lits	cls	nbrs	ratio	cls	lits	ratio	cls
5pipe	9	195	15,729	60	219	0.014	13	953	0.061	13
5pipe_1_ooo	8	188	96,469	276	703	0.007	31	5,067	0.053	31
5pipe_5_ooo	10	241	98,566	232	588	0.006	25	2,741	0.028	25
6pipe	16	395	101,712	242	869	0.009	48	7,726	0.076	48
6pipe_6_ooo	17	546	509,608	722	1,495	0.003	53	7,942	0.016	53
7pipe	24	751	334,496	416	1,625	0.005	82	18,332	0.055	82
9vliw_bp_mc	20	179	60,817	254	789	0.013	44	4,447	0.073	44
exmp72	44	149	483,394	1,143	1,962	0.004	27	3,612	0.007	27
exmp73	61	220	1,796,211	2,100	4,679	0.003	52	11,849	0.007	52
exmp74	41	141	279,970	828	1,978	0.007	34	3,235	0.012	34
exmp75	85	284	542,114	1,042	2,098	0.004	33	3,621	0.007	33
barrel7	4	14	5,243	59	559	0.107	17	1,130	0.216	17
barrel8	5	20	76,546	213	1,996	0.026	36	3,944	0.052	36
barrel9	9	37	83,886	203	1,042	0.012	36	2,726	0.032	36
longmult12	6	19	3,223,323	10,042	49,940	0.015	493	117,595	0.036	493
longmult13	7	20	4,004,512	10,328	55,124	0.014	545	138,413	0.035	545
longmult14	7	22	2,662,334	8,226	39,436	0.015	419	90,568	0.034	419
longmult15	8	24	583,008	3,592	11,609	0.020	199	23,256	0.040	199
c3540	3	9	81,789	724	1,871	0.023	42	4,091	0.050	42
c5315	5	15	9,437	359	718	0.076	22	668	0.071	22
c7552	8	20	23,069	544	1,237	0.054	34	1,565	0.068	34
w10_45	17	52	47,186	320	437	0.009	5	294	0.006	5
w10_60	27	84	617,611	1,530	1,956	0.003	14	2,246	0.004	14
w10_70	33	104	2,581,594	4,328	6,136	0.002	33	7,658	0.003	33
fifo8-200	130	354	206,569	697	2,377	0.012	47	4,980	0.024	47
fifo8-300	195	531	601,883	1,336	5,322	0.009	90	14,511	0.024	90
fifo8-400	260	708	9,148,826	5,258	16,038	0.002	201	87,096	0.010	201

never found 100-to-1 reductions. We conjecture (the BerkMin code with proof generation is not public) that much of the difference is due to the Goldberg and Novikov estimates being based on the PUP method for converting the conflict graph to a resolution derivation (see Section 2).

Another important difference in our results is that Goldberg and Novikov saw a trend for the relative size advantage of RUP to increase dramatically for larger formulas in the same family; they cited the pipe and fifo families. In our data the trend is much less pronounced, or absent. However, there is a strong trend for *resolve-trace* sizes.

In terms of economy of disk space, resolve-trace is the clear winner, hovering around one percent or less of the space needed for a full (explicit) proof. RUP also is quite compact, less than eight percent. Another compact binary format, called "%RPT" (Resolution Proof Trace) for SAT05, contains exactly four numbers per derived clause in the "Full Resolution cls" column, and is not shown. Although

Table 3. Proof timing comparisons. Times are CPU seconds.

GN03 benchmark	`zchaff` Time	Res.Prf Incr.	`checker1` Time	Res.Trace Incr.	`zverify` Time	RUP verify Time
5pipe	11	4	2	4	1	15,242
5pipe_1_ooo	25	12	13	4	2	
5pipe_5_ooo	29	12	12	11	2	
6pipe	80	23	12	31	3	
6pipe_6_ooo	142	58	67	6	4	
7pipe	254	54	39	16	5	
9vliw_bp_mc	39	14	8	2	2	
exmp72	54	55	56	19	2	
exmp73	219	179	246	5	4	
exmp74	70	41	33	24	2	
exmp75	128	59	64	1	3	
barrel7	6	1	1	0	0	
barrel8	27	9	9	1	1	
barrel9	42	11	11	13	1	
longmult12	1,102	445	N/A	143	21	
longmult13	1,663	343	N/A	163	23	
longmult14	974	369	412	153	16	
longmult15	257	72	69	37	5	
c3540	11	9	11	2	1	
c5315	3	2	1	1	0	
c7552	8	3	3	2	1	
w10_45	4	5	6	0	0	
w10_60	27	60	73	10	2	
w10_70	97	250	391	2	4	
fifo8-200	136	27	27	41	3	
fifo8-300	370	76	77	66	6	
fifo8-400	2,737	1,089	N/A	375	16	

it is occasionally shorter than "Resolve-Trace," it is usually 1.5 to 2 times longer, by number count.

Table 3 shows some data on CPU times for proof generation and verification. The first data column shows solve time without any kind of proof generation. This is usually the major cost. The overhead to generate an explicit resolution refutation is substantial, sometimes exceeding the solve time. Checking the proof requires comparable time. In three cases the file was too big for the checker, as currently implemented.

The overhead to generate the resolve-trace is more moderate, but a higher fraction than reported by Zhang and Malik [ZM03b]; their fractions were usually 0.04–0.05 with only one exceeding 0.10. On the other hand, the present `zverify_df` times are considerably lower, relative to the solve time, than previously reported. This is logical, because the modifications made for this paper (see Section 2) transferred some of the workload from the verifier to the solver. Clearly this system is faster than producing and checking an explicit proof, but it is tuned to chaff algorithm, so this can be expected. Its space advantage is more significant than the time advantage.

Finally, we include one data point for checking an RUP proof. This proof had about 13,000 RUP inferences to check. Each inference was first validated as described in Section 3, using `zchaffSE07` to generate the explicit unit resolution derivation, which it does if the conflict is discovered during "preprocessing." That unit resolution derivation was verified with `checker1`. Although each program took about 1/2 second per RUP inference, the total was overwhelming.

5 Have Your Cake and Eat It Too?

Section 4 provided data on several formats for proof presentation. The leanest is "Resolve-Trace," but it is keyed to algorithms that are essentially the same as chaff. The "%RPT" format is fairly compact, and specifies the operands and clashing literal needed for each resolution in a full resolution "%RES" derivation, so it is general for resolution derivations. However, it puts more implementation burden on the programmer of the solver than the RUP format.

The RUP format is longer than the first two by a factor of two to four, but is the easiest to implement, supports reasoning operations other than unit-clause propagation and resolution, and it is miniscule in comparison to a full resolution proof. Its primary drawback is that it is time consuming to check clause by clause.

In terms of ease of verification of a claimed proof, the "%RES" is easiest, and the "%RPT" would be expanded off-line into "%RES" and piggy-back off its checker. The "Resolve-Trace" is more difficult to verify and it is more difficult to establish complete confidence in any program that claims to check it. However, it could also expand off-line into "%RES."

The main point of this section is to argue that the RUP format can also be expanded efficiently into the "%RES" format. Essentially we show that the converse of Proposition 3 from Beame *et al.* [BKS04] holds (see Section 2). This provides hope that we can have our cake (an easy RUP implementation) and eat it too (efficiently check the output).

Let $C_j = [\overline{x_i},\ 1 \leq i \leq k]$ be an RUP inference from formula F_{j-1}. First, apply unit-clause propagation to F_{j-1} (this can be incremental from the result of unit-clause propagation on F_{j-2} for $j \geq 2$). Each derived unit clause is associated with an antecedent clause, as usual. Now put x_i $(1 \leq i \leq k)$ in the queue for further unit-clause propagation and let it run to completion. All unit-clauses derived (including *false*), both during this process and during the "preprocessing" of F_{j-1}, are possible vertices of the resulting conflict graph. If *false* is not derived, C_j does not qualify as an RUP inference. Otherwise, the actual conflict-graph vertices are those reachable from *false* through a chain of antecedents. As described in Section 2 and implemented in `zchaff` and similar solvers, by accessing the vertices in the reverse of the chronological order in which their unit clauses were derived, a correct order for a "trivial resolution derivation" of C_j is achieved.

To keep the process incremental, once C_j has been derived by resolution, the unit clauses derived after putting x_i in the queue need to be backed out; earlier-derived unit clauses can be kept and re-used for C_{j+1}. Now add C_j to F_{j-1} as though it were a conflict clause; in particular, if two literals to "watch" cannot be found, either a conflict or a new unit clause is derived in F_j. Otherwise, the unit-clause "preprocessing" of F_{j-1} carries over to F_j intact.

While many procedures in an existing solver can be used to implement the functions described here, the adaptation requires care and has not been carried out at this time. We hope to be able to report on it in the near future.

References

[BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[BVG00] S. Baase and A. Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 3rd edition, 2000.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[GN02] Eugene Goldberg and Yakov Novikov. Berkmin: a fast and robust sat-solver. In *Proc. Design, Automation and Test in Europe*, pages 142–149, 2002.

[GN03] Eugene Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proc. Design, Automation and Test in Europe*, pages 886–891, 2003.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, June 2001.

[MSS99] J. P. Marques-Silva and K. A. Sakallah. GRASP–a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[VG02] A. Van Gelder. Decision procedures should be able to produce (easily) checkable proofs. In *Workshop on Constraints in Formal Verification*, Ithaca, NY, 2002. (in conjunction with CP02).

[VG05] A. Van Gelder. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), LNAI 3835*, pages 580–594, Montego Bay, Jamaica, 2005. Springer-Verlag.

[ZM03a] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. Theory and Applications of Satisfiability Testing*, 2003.

[ZM03b] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. Design, Automation and Test in Europe*, 2003.

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, Nov. 2001.