Verifying RUP Proofs of Propositional Unsatisfiability: Have Your Cake and Eat It Too

Allen Van Gelder
Computer Science Dept., SOE–3, Univ. of California,
Santa Cruz, CA 95064, http://www.cse.ucsc.edu/∼avg

Abstract

The importance of producing a certificate of unsatisfiability is increasingly recognized for high performance propositional satisfiability solvers. 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 resolution proofs can be extremely long. Several other certificate formats have been proposed and studied, but the verifiers for these formats are beyond any hope of automated verification in their own rights. However, they enjoy the advantages of being easy to implement and reasonable in their space requirements. This paper reports progress on developing a practical system for formal verification of a more compact certificate format, and experimental comparisons are presented. A format called RUP (for Reverse Unit Propagation) is introduced and two implementations are evaluated. This method is an extension of conflict clause proofs introduced by Goldberg and Novikov.

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 (Goldberg & Novikov 2003), who built on BerkMin (Goldberg & Novikov 2002), and Zhang and Malik (Zhang & Malik 2003b; 2003a), who built on Chaff (Moskewicz *et al.* 2001). Sinz and Biere also discuss proof traces and checking (Sinz & Biere 2006). In all these cases, the authors are checking their own solvers. 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 (Van Gelder 2002a) 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.

The first known attempt to have satisfiability solvers produce proofs to be verified by an independently written verifier occurred in the verification track of the SAT–2005 solver competition. Results from that track were initially disappointing because only one solver had any success, and its proofs were extremely long. However, a recent short paper shows that this was due to an inefficiency in that solver; its procedure to generate a resolution proof from a conflict graph had a surprising worst case that was exponential in the size of the conflict graph (Van Gelder 2007b). The procedure was revised and proofs got shorter by several orders of magnitude, demonstrating that production of proofs and verification of proofs is now within reach for substantial benchmarks.

A detailed specification for an explicit resolution derivation (**%RES**) was used for the verification track of the SAT–2005 solver competition. We have a verifier for the `%RES` format (named `checker3`) that is able to handle proofs up to about 750 gigabytes (GB) on certain available compute servers. The limitation on many systems is that only 40 bits of memory address are us-

able, which is 1024 GB, although the CPU chip is called "64-bit".

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 reviews conflict graphs; Section 3 presents an attractive proof format named RUP for Reverse Unit Propagation that is easy-to-implement, general, and compact. Section 4 shows some additional properties of RUP and outlines how to use them to reap the benefits of RUP without paying a major performance penalty, by using the ideas of conflict graphs. The ideas are implemented in the program `rupToRes`. Section 5 experimentally compares several approaches for proof generation and checking, and Section 6 draws conclusions.

2 A Quick Review of Conflict Graphs

Most, if not all, leading SAT solvers use a *conflict graph* data structure to infer *conflict clauses*. This section reviews them briefly, for self-containment.

Figure 1 illustrates a conflict graph. Our notation varies from other papers (Zhang *et al.* 2001; Beame, Kautz, & Sabharwal 2004) to better reflect the actual data structures used by the programs, and agrees more closely with the original presentation (Marques-Silva & Sakallah 1999). 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 (in circles), including *false*, is labeled with an "input" clause, called the *antecedent clause*. *Decision literals* (in boxes), also called *assumed literals* or *guessed literals*, do not have an antecedent clause. Solid arrows point to vertices associated with the current (highest) *decision level*, while dotted arrows point to vertices that were assumed or implied at earlier (lower) decision levels. Each decision literal is on a different decision level.

For our purposes an "input" clause is either a clause in the original formula or a clause that was derived before the most recent decision literal was guessed, hence the quotes.

Recent papers have observed the connection between conflict graphs and resolution (Goldberg & Novikov 2003; Zhang & Malik 2003b; Beame, Kautz, & Sabharwal 2004; Van Gelder 2005). Given a cut, the antecedent clauses on the conflict side of the cut (i.e., the side containing *false*) logically imply the conflict clause, which consists of the negations of those reason-side literals that are adjacent to some vertex on the conflict side (e.g., in the figure, for the first UIP cut, $\neg p$, a, and $\neg t$ are adjacent to some conflict-side vertex). Of course, by the completeness of resolution there must be a resolution derivation of the conflict clause from the antecedent clauses. In the discussion, we shorten "resolution derivation" to "derivation".

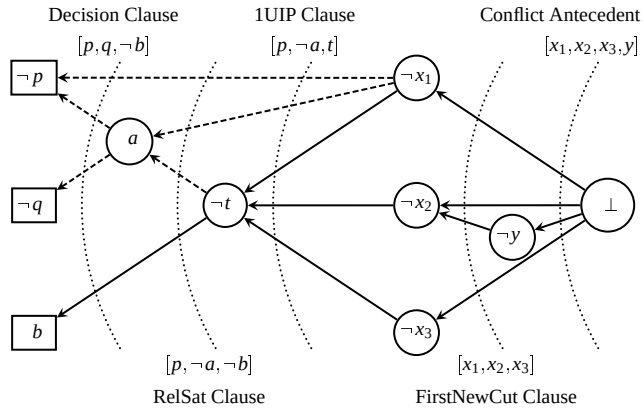

Figure 1: Conflict graph with several cuts shown.

2.1 Trivial Resolution Derivation (TVR)

Several restricted forms of resolutions have been defined and studied over the years. A *linear 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 other operand may be an "input" clause or an earlier-derived clause of the linear derivation. An *input derivation* is a linear derivation with the further restriction that the "other" operand must be an "input" clause; earlier-derived clauses of the same linear derivation are not acceptable.

Beame *et al.* (Beame, Kautz, & Sabharwal 2004) 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. Using a correct order is crucial to achieve a successful trivial resolution derivation.

A difficulty with TVR scheme is that a correct order is not readily accessible from the data structure of the conflict graph. To obtain a valid TVR order, the following rule should be obeyed:

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.

Many solvers, based on the `grasp/chaff` strategy, create a sequence of "implied" literals in the chrono-

logical order in which they entered the conflict graph. (This sequence is needed to undo variable assignments during backtracking.) It is not difficult to show that one order that satisfies the cut-crossing rule is the *reverse* chronological order. Thus TVR is implemented in such solvers, at least implicitly, in the procedure that constructs conflict clauses.

2.2 Pseudo-Unit Propagation (PUP)

Although TVR uses the minimum number of resolution *operations* (exactly one for each vertex other than *false* on the conflict side), it might produce a derivation whose length is quadratic in the size of the conflict graph, because clause *sizes* can range up to the number of vertices in the conflict graph. We now describe a different extraction method, which we call *pseudo-unit propagation* (**PUP**). It uses more operations, but produces shorter clauses, in most cases. PUP is of interest also because it is a natural order, in a sense that will be explained shortly.

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. Therefore, if the conflict clause has w literals, a pseudo-unit clause has at most $(w+1)$ literals, by definition. If the conflict clause is the empty clause, then pseudo-unit clauses are true unit clauses.

Among the literals of the pseudo-unit clause that are also in the conflict clause, *one* might be the negation of a *unique implication point* (**UIP**) literal (Marques-Silva & Sakallah 1999), while the others are literals that were falsified at lower decision levels. The UIP literal might be the decision literal at the current decision level.

The extraction method works as follows: Visit vertices on the conflict side of the cut in a reverse topological order (using edge orientations illustrated in Figure 1). A topological order may be found by a depth-first search from the *false* vertex, or other means. When visiting vertex x, derive a pseudo-unit clause on x, if necessary, which we'll call $pup(x)$.

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, so is a natural order in this sense.

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 (and on the conflict side) whose pseudo-unit clause has been derived already. Start a linear derivation with $ante(x)$. If no y_i are on the conflict side, we are done, and $pup(x) = ante(x)$. Otherwise, for each y_i on the conflict side, 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. The number of resolution operations is the number of edges *to* vertices on the conflict side.

2.3 Comparison of TVR and PUP

In summary, both the TVR and PUP methods guarantee that total derivation size is polynomial in the size of the conflict graph, but have nonlinear worst cases. Either method might be an order of magnitude better than the other for a particular conflict graph.

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 does essentially the same resolutions as a PUP after-the-fact system, and most likely 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 (Goldberg & Novikov 2003). A clause $C = [x_1, \ldots, x_k]$ is a *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. A *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 a 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 a *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 clause C_j is a RUP inference from F_{j-1} for all $j \geq 1$; that is, the sequence of all derived conflict clauses is a 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.* (Beame, Kautz, & Sabharwal 2004) proved (again, using different terminology, their Propositions 3 and 4) that

Proposition 3.1 *A conflict clause defined by any cut in a conflict graph is a 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 2.1.* □

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.

In general, search-based solvers use reasoning to reduce searching that can be classified as *pre-order* or *post-order* (Van Gelder 2002b). (Recently the term "look-ahead" has been used for pre-order reasoning.) Conflict analysis (also called clause learning and several other names) is post-order, whereas binary-clause analysis and equivalent-literal analysis and similar operations are pre-order. The RUP format can be used to record both kinds of reasoning; it is not limited to programs that only perform conflict analysis.

We have developed a detailed specification of a RUP refutation (`fileformat_rup.txt` at the URL in Section 1 and elsewhere), which was accepted in the verification track of the SAT–2007 solver competition. The main point of this section is to argue that a 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, and it is easily reduced to unit clause propagation. Therefore, at first blush, verifying a 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 a 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 a RUP refutation of F_0. One program composes two sequences of extended formulas,

$$\left.\begin{array}{rcl} F_j & = & F_{j-1} \cup \{C_j\} \\ G_j & = & F_{j-1} \cup \{\overline{C_j}\} \end{array}\right\} \qquad \text{for } 1 \leq j \leq k \quad (1)$$

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 `checker3` 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 Have Your Cake and Eat It Too?

The RUP format is reasonably compact, is the easiest to implement of all formats proposed to date, and has flexibility to accommodate various solver strategies. Section 3 showed that RUP proofs can be transformed into a form that can be checked in log space, which is theoretically appealing. However, the procedure is redundant, and in terms of time, practical experience shows that it is excessively slow.

The main point of this section is to argue that the RUP format can also be expanded efficiently into the "`%RES`" format (which then can be checked in log space). Essentially we show that the converse of Proposition 3 from Beame *et al.* (Beame, Kautz, & Sabharwal 2004) holds (see Proposition 3.1 in Section 3). This provides hope that we can have our cake (an easy and flexible RUP implementation) and eat it too (efficiently check the output).

Continuing with the notation of Section 3, let $C_j = [\overline{x_i}, 1 \leq i \leq k]$ be a 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 unit clauses x_i ($1 \leq i \leq k$; this is the negation of C_j) 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 a RUP inference. Otherwise, the actual conflict-graph vertices are those reachable from *false* through a chain of antecedents.

As described in Section 2.1 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.

We developed a prototype implementation of the above proof transformation, called `rupToRes`, using `zchaff` as a base. The main idea is to enqueue the

unit clauses x_i all at *decision level* 1 as though they were decision literals, or "guesses". All earlier-derived unit clauses are associated with *decision level* 0. As "decision literals," the x_i's do not have antecedents. If some of the x_i's were derived as unit clauses at level 0 earlier (and this does happen in practice), they are not enqueued redundantly. Then the derived clause actually subsumes C_j.

The big difference from normal operation is that `zchaff` and similar solvers expect each succeeding guess (x_1, x_2, ...) to be at a *higher* decision level. But the conflict analysis uses first UIP, and would not derive the desired clause if each x_i were at its own decision level. Referring to Figure 1, the "Decision Cut" would be needed, instead of the "First UIP Cut", which is implemented.

However, by labeling all the x_i's as being at *decision level* 1, the already implemented conflict analysis derives the desired clause, C_j. Also, the already implemented procedure for backtracking out of the variable assignments that were made at *decision level* 1 works without change, as does the already implemented procedure for adding the newly derived clause to the database. Some tweaks were needed so the program did not get upset that there were multiple "decision literals" at one level, and so that it never tried to make any guesses of its own.

5 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 (Goldberg & Novikov 2003) and Zhang and Malik (Zhang & Malik 2003b). This paper provides the first in-depth data on explicit resolution proofs as well as comparison of various formats.

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 *nonempty* 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 such line describes one "trivial resolution," as defined in Section 2.1. The final derivation of the empty clause is presented in a different, more involved, format. The system is implemented as a solver, `zchaff`, and a verifier `zverify_df`. The most recent release as of the experiments was 2005.11.15. The trace format proposed by Sinz and Biere (Sinz & Biere 2006) is essentially the union of the RUP and resolve-trace formats (with an option to omit the RUP part).

A "Special Edition" of `zchaff`, call it `zchaffSE`, is dated March 2005 and was entered into the verification track of the SAT–2005 competition. It combines the functionality of `zverify_df` into `zchaff` and writes the full resolution derivation in the binary format specified for SAT–2005 and identified by a header beginning "`%RESL32`.". (Data produced during SAT–2005 for `zchaffSE` is skewed due to the issues discussed in (Van Gelder 2007b).) This program *accumulates* all the data that allows it to reconstruct derivations of conflict clauses, then when unsatisfiability has been established, 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 its companion program `zverify_df`, and a major purpose of this study is to see if that burden can be lightened by using the RUP format.

As described in more detail elsewhere (Van Gelder 2007b), there were minor modifications made for `zchaff` to include the empty-clause derivation as the last line, in the same resolve-trace format as the nonempty clauses, and for `zverify_df` to expect it; this obviated the need to also provide the final conflict graph. Corresponding modifications were made to `zchaffSE`. The data reported is for the modified versions, called `zchaffJ07`, `zverifyJ07`, and `zchaffJ07SE`, in the tables. A new version of `zchaff` and `zchaff_df` (2007.3.12), posted after our experiments were run, gives very similar results.

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.

As described in Section 4, we developed *rupToRes* to transform a RUP proof into `%RES` format. The program `checker3` was used to verify `%RES` proofs, using the standard `mmap` facility to simulate having the entire proof file in memory. The `%RESL32` format was designed so the file could be processed *in situ* as an array of 32-bit `ints` on `x86` (little-endian) architectures.

Computations were done on AMD Opteron systems with 2.6 GHz 64-bit dual-core CPUs and 8 GB of real memory. For `checker3`, it is best if the system has an available local disk large enough to store the proof. Although a remote NFS-mounted disk has been used successfully to verify a 40 GB proof in 4.6 CPU minutes, the CPU utilization was only 3%, and the elapsed time was well over two hours. Using a local disk brought CPU utilization up to 10–20%. For proofs that fit in real memory the CPU utilization was near 100%. The programs other than `checker3` have a much smaller memory requirement and run easily on 32-bit configurations (which have 4 GB of memory address space).

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 (Goldberg & Novikov 2003; Zhang & Malik 2003b) 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 in seconds unless

Table 1: Proof length comparisons.

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

GN03	Input		Full Resolution		Resolve-Trace			RUP			`rupToRes`		
Benchmark	Vars	Cls	lits	cls	nbrs	ratio	cls	lits	ratio	cls	lits	ratio	cls
5pipe	9	195	15,729	60	219	0.014	13	953	0.061	13	17,590	1.12	268
5pipe_1_ooo	8	188	96,469	276	703	0.007	31	5,067	0.053	31	122,457	1.27	495
5pipe_5_ooo	10	241	98,566	232	588	0.006	25	2,741	0.028	25	99,383	1.01	470
6pipe	16	395	101,712	242	869	0.009	48	7,726	0.076	48	175,149	1.72	697
6pipe_6_ooo	17	546	509,608	722	1,495	0.003	53	7,942	0.016	53	530,143	1.04	1,260
7pipe	24	751	334,496	416	1,625	0.005	82	18,332	0.055	82	465,423	1.39	1,260
9vliw_bp_mc	20	179	60,817	254	789	0.013	44	4,447	0.073	44	79,148	1.30	453
exmp72	44	149	483,394	1,143	1,962	0.004	27	3,612	0.007	27	472,453	0.98	1,280
exmp73	61	220	1,796,211	2,100	4,679	0.003	52	11,849	0.007	52	1,608,988	0.90	2,335
exmp74	41	141	279,970	828	1,978	0.007	34	3,235	0.012	34	282,250	1.01	982
exmp75	85	284	542,114	1,042	2,098	0.004	33	3,621	0.007	33	534,820	0.99	1,332
barrel7	4	14	5,243	59	559	0.107	17	1,130	0.216	17	6,076	1.16	74
barrel8	5	20	76,546	213	1,996	0.026	36	3,944	0.052	36	76,481	1.00	227
barrel9	9	37	83,886	203	1,042	0.012	36	2,726	0.032	36	80,497	0.96	233
longmult12	6	19	3,223,323	10,042	49,940	0.015	493	117,595	0.036	493	11,672,980	3.62	27,339
longmult13	7	20	4,004,512	10,328	55,124	0.014	545	138,413	0.035	545	14,139,745	3.53	30,833
longmult14	7	22	2,662,334	8,226	39,436	0.015	419	90,568	0.034	419	9,538,940	3.58	23,405
longmult15	8	24	583,008	3,592	11,609	0.020	199	23,256	0.040	199	1,460,110	2.50	7,400
c3540	3	9	81,789	724	1,871	0.023	42	4,091	0.050	42	202,756	2.48	1,326
c5315	5	15	9,437	359	718	0.076	22	668	0.071	22	13,118	1.39	379
c7552	8	20	23,069	544	1,237	0.054	34	1,565	0.068	34	36,279	1.57	695
w10_45	17	52	47,186	320	437	0.009	5	294	0.006	5	48,366	1.03	372
w10_60	27	84	617,611	1,530	1,956	0.003	14	2,246	0.004	14	619,099	1.00	1,606
w10_70	33	104	2,581,594	4,328	6,136	0.002	33	7,658	0.003	33	2,923,503	1.13	4,731
fifo8-200	130	354	206,569	697	2,377	0.012	47	4,980	0.024	47	215,206	1.04	1,051
fifo8-300	195	531	601,883	1,336	5,322	0.009	90	14,511	0.024	90	685,761	1.14	1,891
fifo8-400	260	708	9,148,826	5,258	16,038	0.002	201	87,096	0.010	201	11,905,078	1.30	6,391

stated otherwise.

5.1 Space Comparisons

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 1 shows our 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.

Our first new finding is that 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 (Zhang & Malik 2003a), as far as conflict clauses go. 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 an online PUP method for converting the conflict graph to a resolution derivation (see Section 2.3).

Another important difference in our results is that Goldberg and Novikov saw a trend for the relative size advantage of "conflict clause proof" over resolution proof 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, for RUP vs. resolution proofs. Again, this may be due to their estimates assuming a different strategy for generating resolution proofs.

In terms of economy of disk space, *resolve-trace* is the clear winner, hovering around 1% or less of the space needed for a full `%RES` proof. RUP also is quite compact, typically about 5%.

Another compact binary format for SAT–2005, called *Resolution Proof Trace* (`%RPT`), contains exactly four numbers per derived clause and is not shown (see the "Full Resolution cls" column for the numbers of derived clauses). Although `%RPT` is occasionally shorter than *resolve-trace*, it is usually 1.5 to 2 times longer, by number count. However, `%RPT` includes the clashing literal and thereby is able to express a generalization of

Table 2: Proof timing comparisons. Times are CPU seconds.

GN03 benchmark	zchaffJ07 Time	zchaffJ07SE Increment	checker3 Time	Res.Trace Increment	zverifyJ07 Time	rupToRes + checker3 Time
5pipe	11	4	2	4	1	7
5pipe_1_ooo	25	12	13	4	2	32
5pipe_5_ooo	29	12	12	11	2	23
6pipe	80	23	12	31	3	61
6pipe_6_ooo	142	58	67	6	4	110
7pipe	254	54	39	16	5	169
9vliw_bp_mc	39	14	8	2	2	53
exmp72	54	55	56	19	2	86
exmp73	219	179	27	5	4	285
exmp74	70	41	33	24	2	60
exmp75	128	59	64	1	3	114
barrel7	6	1	1	0	0	6
barrel8	27	9	9	1	1	38
barrel9	42	11	11	13	1	23
longmult12	1,102	445	57	143	21	3681
longmult13	1,663	343	89	163	23	4887
longmult14	974	369	44	153	16	3775
longmult15	257	72	69	37	5	625
c3540	11	9	11	2	1	36
c5315	3	2	1	1	0	4
c7552	8	3	3	2	1	11
w10_45	4	5	6	0	0	9
w10_60	27	60	73	10	2	88
w10_70	97	250	391	2	4	394
fifo8-200	136	27	27	41	3	56
fifo8-300	370	76	77	66	6	184
fifo8-400	2,737	1,089	397	75	16	2553

resolution (Van Gelder 2005) that cannot be expressed with *resolve-trace*.

The last three columns of Table 1 show data for the `%RES` proofs generated from the RUP proofs by `rupToRes`. In most cases the ratio is near 1.00, but the maximum is 3.62. This variation occurred although the same conflict clauses, in the same order, were the bases for both proofs. This reinforces the observation that there are many possible resolution derivations of a given conflict clause from a given set of input clauses, and their sizes can vary widely.

5.2 Time Comparisons

Table 2 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 a full (`%RES`) refutation, shown in the second data column, is substantial, sometimes exceeding the solve time. Checking the proof (third data column) usually requires comparable time.

Again we notice that *longmult* 12, 13, and 14 behave differently from the other benchmarks. Here the verification time is much less than the proof generation overhead, whereas usually they are about the same. Also, the RUP files of these three benchmarks , when transformed by `rupToRes` into `%RES` files, were more than 3.5 times the sizes of the original `%RES` files, whereas most other benchmarks transformed into `%RES` files about the same size as the originals, as noted in Table 1.

The overhead to generate the resolve-trace is more moderate, but a higher fraction than reported by Zhang and Malik (Zhang & Malik 2003b); their fractions were usually 0.04–0.05 with only one exceeding 0.10. On the other hand, the `zverifyJ07` times are considerably lower, relative to the solve time, than previously reported. This is logical, because the modifications made for this paper (see beginning of this section) transferred some of the workload from the verifier to the solver. Clearly this combination is faster than producing and checking an explicit `%RES` proof, mainly due to not writing to disk and reading back the explicit proof. The trade-off is that it needs to construct the all the conflict clauses in memory, meaning that it is limited by the sum of real memory and swap space. By using `mmap`, `checker3` is able to use whatever disk space the file actually occupies as "memory." Also, the algorithm is somewhat more involved, compared to `checker3`, because it needs to construct clauses, manage memory, and so on.

Comparing the last column to the sum of data columns 2 and 3 shows the time penalty for using the RUP format to record the proof, followed by `rupToRes` and `checker3` to verify it, as opposed to producing the `%RES` proof directly. In most cases the penalty is negligible to moderate. But for the *longmult* benchmarks it is substantial.

Naively verifying RUP clause by clause is very expensive. We include one data point to make this concrete. For *5pipe*, one of the easier benchmarks considered, the RUP proof had about 13,000 clauses to check. Each incremental RUP proof was validated as described in Section 3. We used `zchaffJ07SE` to generate the `%RES` *unit resolution* refutation, which it does if the conflict is discovered during "preprocessing." A flag was added to tell `zchaffJ07SE` to fail if it could not derive the empty clause with unit propagation only. That unit resolution refutation was then verified with `checker3`. Although each program took about 1/2 second per RUP clause, the total was overwhelming, amounting to 4.2 CPU hours. In contrast, `rupToRes` followed by `checker3` took 7 seconds.

6 Conclusion

We compared several approaches to verification of propositional proofs of unsatisfiability. Such proofs amount to proofs of propositional theorems, and were considered intractable due to their length, until recently. Our emphasis was on making it practical to separate the developer from the verifier. All previously reported efforts we are aware of consisted of the developers "verifying" their own work and no one else being compatible.

We developed a program `rupToRes` to expand a RUP proof into an explicit `%RES` proof, as well as formally specifying both formats for other developers to use. We developed a program `checker3` that is able to check `%RES` proofs up to 750 GB in theory, and we actually checked a 56 GB proof in 5.5 CPU minutes, although it was 3.2 hours of elapsed time because the proof resided on an NFS file system. The data suggests that CPU time varies linearly with proof length, as predicted by theoretical analysis.

We found that resolution proofs were much smaller than estimated by Goldberg and Novikov. For problems comparable to the benchmarks tested, it is quite practical to produce RUP proofs, and expand them offline to `%RES` proofs which can be checked in deterministic log space. We regard having a very simple-to-verify format to be crucial to having complete independence of and confidence in the verifier.

We found that `zverify_df` was considerably faster than `checker3`, after being fixed (be sure to use a version dated 2007.3.12 or later). However, the format in the public version is not formally specified, and is tuned for `zchaff` or a very similar solver. To the best of our knowledge, no other developer has used that format. Our tests are based on a simplification of the format used by the public version. Sinz and Biere recently proposed a format that combines our version of *resolve-trace* with RUP. We have not had time yet to experiment with their software.

The RUP format (described in more generality than Goldberg and Novikov, but essentially the same syntax) has been shown to be almost as practical to verify as other formats that are considerably more difficult to implement.

The work described in this paper has made it possible for developers to add RUP output routines with relative ease to their favorite solvers and enter them in the "verified unsatisfiable" track of the annual SAT Solver Competitions.

The theme of all the proof options offered in the SAT Solver Competitions is that we only need to trust one very simple program: `checker3`, or another implementation that does the same task. All the other software (e.g., `zchaffJ07SE` or `rupToRes` or other solvers) prepares input for `checker3`, but `checker3` checks the claimed proof against the original CNF formula. If the proof is correct, the formula is verified to be unsatisfiable, and it does not matter if the programs that prepared the proof are buggy.

Although most of the benchmarks in the suite we adopted from Goldberg and Novikov behaved predictably and consistently, *longmult* 12, 13, and 14 were outliers. Understanding what aspects of those benchmarks causes their variant behavior deserves future study, and might lead to additional insights about solving such problems, as well as generating proofs.

References

Beame, P.; Kautz, H.; and Sabharwal, A. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research* 22:319–351.

Goldberg, E., and Novikov, Y. 2002. Berkmin: a fast and robust sat-solver. In *Proc. Design, Automation and Test in Europe*, 142–149.

Goldberg, E., and Novikov, Y. 2003. Verification of proofs of unsatisfiability for cnf formulas. In *Proc. Design, Automation and Test in Europe*, 886–891.

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

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

Sinz, C., and Biere, A. 2006. Extended resolution proofs for conjoining bdds. In *1st Intl. Computer Science Symp. in Russia (CSR 2006), LNCS 3967*. St. Petersburg: Springer-Verlag. (see also `http://fmv.jku.at/tracecheck`).

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

Van Gelder, A. 2002b. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*. (Also at `http://cse.ucsc.edu/~avg/Papers/sat-pre-post.pdf`.).

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

Van Gelder, A. 2007b. Verifying propositional unsatisfiability: Pitfalls to avoid. In *SAT–2007*. Lisbon: Springer-Verlag. (preprint at `http://www.cse.ucsc.edu/~avg/Papers/proofs-sat07.pdf`).

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

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

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