Annals of Mathematics and Artificial Intelligence manuscript No.
(will be inserted by the editor)

Producing and Verifying Extremely Large Propositional Refutations:

Have Your Cake and Eat It Too

Allen Van Gelder

Received: date / Accepted: date

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. This paper reports on a tool that has verified proofs more than 1600 gigabytes 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, some of the alternative formats enjoy the advantages of being easy to produce proofs for, and reasonable in their space requirements. This paper reports progress on developing a practical system for formal verification of a more compact certificate format. 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, and is compatible with conflict-clause minimization. Extracting a resolution derivation from other decidable theories is discussed briefly.

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. In such cases, a satisfying assignment essentially exhibits a bug, whereas unsatisfiability implies a lack of bugs, at least for the property being verified. 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]. Sinz and Biere [SB06] and later Biere [Bie08] also discuss proof traces and checking. 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.

Computer Science Dept., SOE–3, Univ. of California, Santa Cruz, CA 95064, USA, `http://www.cse.ucsc.edu/~avg`

Most high-performance satisfiability solvers and special-theory decision procedures are unable to provide a proof of unsatisfiability. Since bugs have been discovered in many such programs long after being put into service [RS01] (discussed briefly in Section 2.3), an uncheckable decision poses a significant problem if important economic or safety decisions are to be based upon it. Our thesis is that decision procedures can and should be designed to be able to output a proof. Shankar proposes a rather far-reaching project along these lines [Sha08].

If an important decision is to be taken based on claims that certain statements have been formally verified, there is a need to be able to verify the verifier. It is probably impractical to prove that the decision procedure is bug-free, but if it produces an *easily verifiable certificate* (i.e., proof), then that *certificate* can be verified without addressing the issue of whether the program is bug-free.

The gold standard that we propose 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.

The thesis of this paper is that *in practice*, any proof language that is seriously proposed for general use, and is in deterministic log space will be easy to write verifiers for. A very robust verification environment may be created by several independent groups writing verifiers for the same language and checking each other's verifiers. Certainly, it is *possible* to write an algorithm that uses only log space and is extremely complicated to verify, but such algorithms will remain unverified and need not be trusted.

An explicit resolution proof is one in which each derived clause is stated explicitly, along with the numbers of 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 verified-unsatisfiable track of the SAT–2005 solver competition. Results from that track were initially disappointing because only one solver had any success (`zchaffSE`), and its proofs were extremely long. However, Section 3.2 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. 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. The verified-unsatisfiable track was repeated in the SAT–2007 solver competition with greater participation and greater success. Results are reported in Section 9.3.

A detailed specification for an explicit resolution derivation (called the **%RES** format) was used for the verified-unsatisfiable track of the SAT–2005 solver competition. The author has developed a verifier for the `%RES` format (named `checker3`) that is able to handle proofs with lengths in the thousands of gigabytes on certain available compute servers. The practical upper limit depends on the amount of disk space in the system that is available to store the proof, which with modern hardware might be thousands, or even hundreds of thousands

of gigabytes.[1] (One gigabyte (GB) is 2^{30} bytes, or about $1.1 \cdot 10^9$ bytes). Remotely mounted NFS disk storage may be used. To achieve reasonable performance, `checker3` treats the entire proof as being in addressable memory. It has actually verified proofs up to 1611 GB.

The verifier `checker3` accepts two binary formats and one ASCII format. The specification document (`ProofChecker-fileformat.txt`) and software (`RUPkit`, `ExpandTrace`) are available on the Internet (see Section 1.4) and in Appendix A.

Notice that a much more compact format, called *resolution proof trace* (**%RPT**, see `ProofChecker-fileformat.txt` or Appendix A), states the two operands needed for each resolution operation, but does not materialize the clause, i.e., does not store the literals that comprise 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. Another compact format, called *reverse unit propagation* (**%RUP**, see `fileformat_rup.txt` or Appendix A), states the derived clauses but does not provide any information about how each was derived. This language also has little hope for log-space recognition. However, both formats are useful as an intermediate format that can be post-processed into the `%RES` format for final checking by an independent checker.

This paper focuses on proofs for the purpose of verifying correctness, but Biere argues that proofs have other uses in several applications [Bie08]. In these contexts, proofs are viewed as explanations, the main goal is to extract useful information, rather than check correctness, and so compact formats are preferred. Biere argues that the proof trace should contain *both* the derived clause and the clause numbers needed to derive it.

1.1 Terminology and Notation

We assume a set of *propositional variables* is given. Propositional variables may take on the values *true* (or 1) and *false* (or 0). A *literal* is a propositional variable, say x, or its negation, $\neg x$. We consider $\neg\neg x$ to be synonymous with x and $\overline{x}$ to be synonymous with $\neg x$. To distinguish propositional variables from literals, usually letters near the middle of the alphabet denote literals, and letters near the end of the alphabet denote propositional variables. For uniformity, *false* is sometimes considered to be a literal, denoted by $\perp$ in these cases.

A *clause* is a disjunctively connected set of literals. An empty clause evaluates to *false*. Unless it is specifically qualified by the term *tautological clause*, the set of literals in a clause is assumed to be consistent, i.e., x and $\neg x$ are not both present. The literals comprising a clause may be shown between square brackets. The *width* of a clause is the number of literals in it. A *CNF formula* (*formula* for short) is a conjunctively connected set of clauses. An empty formula evaluates to *true*. The *representation* of a clause or formula might include duplicates, but the logical objects are sets.

For two clauses, $C_1 = [r, p_1, \ldots, p_k]$ and $C_2 = [\neg r, q_1, \ldots, q_j]$, the rule of inference known as *resolution on* r yields the *resolvent*, $[p_1, \ldots, p_k, q_1, \ldots, q_j]$. In this context, r is called the *clashing literal*. We assume the resolvent is not tautological, unless stated otherwise. A *resolution proof* is a sequence of resolutions whose operand clauses are in the formula under consideration or derived earlier in the proof. A *resolution refutation* (*refutation* for short) is a resolution proof that derives an empty clause. *Unit-clause propagation* consists of doing all possible resolutions in which at least one operand is a unit clause.

[1] A hardware addressing limit of 48 bits might come into play at some point.

The main ideas of this paper should be accessible without knowing all the more technical details associated with modern satisfiability solvers, but explanations are found in papers cited. Some familiarity with this cited literature is needed to follow the more technical parts of this paper.

1.2 Related Work

To our knowledge, no previous paper has discussed a propositional proof language that is in deterministic log space. Some propositional proof verifiers have been described that are tuned to solvers written by the same authors. Prominent among these are the `resolve_trace` format reported in [ZM03b] and the `trace` format reported in [SB06, Bie08]. These proof formats indicate what resolutions should be performed, but do not materialize all the derived clauses.

A quite different direction is taken by Goldberg and Novikov, who materialize the *conflict clauses* but give no information about their derivation [GN03]; however, their code is not public. Roughly speaking, the `tracecheck` format is the union of the `resolve_trace` format and the `conflict clause` format.

Acceptable proofs in the `trace` format are defined only informally, and involve concepts of linearity and regularity. The verifier has only been tested against the authors' own solvers, such as `booleforce` and `picosat`, which produce only a subset of the proofs the verifier is supposed to be able to check.

Indeed, `tracecheck`, the verifier for the `trace` format [Bie08] is supposed to accept *input*, *regular* proofs (see cited paper for the details of the definition), but it is known to reject certain proofs that fit this definition. This is not just a matter of fine tuning the definition, because proofs are known that `tracecheck` accepts if the literals of a derived clause are listed in one order, and rejects if they are listed in a different order.[2]

This finding reinforces one thesis of this paper, that verifiers should be developed independently of the solvers they are to verify. Apparently, `tracecheck` has been used by its developers for years without encountering this common class of proofs that it incorrectly rejects. We conjecture that the reason for this is that their solvers never produce such proofs.

An ongoing theoretical direction concerns the relative strength of general resolution and resolution proofs that can be produced with the techniques called conflict-driven clause learning (CDCL), as used by most of today's leading SAT solvers. Are there families of formulas for which general resolution proofs can be exponentially shorter than CDCL proofs? Bonet and Buss add a recent chapter to what is becoming a saga, and provide a good summary of what is known about this open question [BB12].

1.3 Overview and Summary

The primary contribution of this paper is to provide empirical evidence for the author's thesis about what is needed in the field of verification of statements in propositional logic and its close relatives, as laid out in the beginning of the Introduction. Although there are several theoretical and algorithmic observations along the way, they are based on straightforward combinations of facts and ideas already in the literature, and cited. A reader conversant with this literature should have no trouble reconstructing these combinations, and may well

[2] `TracecheckBug.tar.gz` at the web page in Section 1.4.

improve on the author's constructions. What directions the verification field should pursue is a matter of human judgement, so empirical evidence is important for informing such judgements. Preliminary versions of parts of this work were presented at meetings [VG02a, VG07, VG08]. The remainder of this paper is organized as described below.

This paper describes formats for propositional resolution derivations, first-order theorems and some decidable quantifier-free theories that are recognizable in deterministic log space. Section 2 discusses motivations for defining encodings that are log-space recognizable, and describes such formats for propositional resolution and first-order resolution. This section also describes how to extract resolution proofs from the procedure that detects equivalent literals, and from the congruence-closure algorithm for equality of uninterpreted functions, with low overhead in both cases.

In Section 3 we identify and analyze a pitfall in a published procedure to generate a resolution proof from a conflict graph, showing that this procedure has an exponential worst case. This worst case greatly limited its capability to produce proofs. Sections 4–6 discuss, analyze, and compare two ways to ensure polynomial behavior. Identifying and correcting these performance problems paved the way for an in-depth empirical study, reported in Section 9.

Section 7 presents an attractive proof format named RUP for Reverse Unit Propagation that is general, compact, and easy to implement, or even retrofit into an existing solver. This format is a simple generalization of *conflict-clause proofs*, originated by Goldberg and Novikov [GN03], but for which software was not made public. Section 8 shows some additional properties of RUP and outlines how to use them to reap the benefits of RUP without paying a major performance penalty. A prototype program, `rupToRes`, expands a RUP proof into an explicit (i.e., `%RES`) resolution derivation.

Early experimental results with the 2005 `zchaff` solver were very disappointing, but Section 9 shows that fixed versions, based on Section 3, improved those results by factors in the thousands in some cases. These results on *actual* resolution proof sizes that show that they are much smaller than the *estimated* sizes of Goldberg and Novikov [GN03]. The implication is that `%RES` resolution proofs for reasonably difficult benchmarks can be of a practical size.

A new experiment is reported in Section 9, using some of the same solvers and benchmarks from the verified unsat track of the SAT-2007 competition on a system with much more disk space than the 2007 runs. A verifier for propositional resolution, named `checker3`, has been implemented and tested on proofs greater than 1600 gigabytes in length during this experiment. Other available verifiers are limited to what can fit in memory, usually 2–8 gigabytes. The `checker3` implementation is the first publicly available propositional verifier to be developed independently of any solver, can serve as a model for verifiers for the other theories mentioned.

Data in Section 9 shows that it is quite practical in many cases to produce RUP proofs and then use `rupToRes` to expand them offline to `%RES` proofs. However, sometimes `rupToRes` is very inefficient at discovering proofs. Note that `rupToRes` processes the `%RUP` format, which has no operand information, whereas `tracecheck` processes the `trace` format, which contains this information. For two benchmark families that were introduced in 2007, `rupToRes` produced `%RES` proofs up to 110 times as long as those produced by `tracecheck`. Overall, `rupToRes` was 20 times slower than `tracecheck`. The advantage of the `%RUP` format is its ease of implementation, but this data indicates the magnitude of the computational penalty, at least with the current software. Further analysis of the results indicates a future direction for an improved implementation of `rupToRes`. Section 10 draws conclusions. Appendix A briefly describes several proof formats discussed in the paper.

1.4 ProofChecker Web Page

For files related to this paper, including reports, code, formulas, specifications of proof formats, and other documents, please see

`http://www.cse.ucsc.edu/~avg/ProofChecker/.`

This URL provides a simple directory listing. File names are mentioned throughout the paper for specific topics.

2 Easily Checkable Proofs

If an important decision is to be taken based on claims that certain statements have been formally verified, there is a need to be able to verify the verifier. It is probably impractical to prove that the decision procedure is bug-free, but if it produces an easily checkable proof, then that *proof* can be verified without addressing the issue of whether the program is bug-free.

Propositional resolution proofs are very easy to check, independently of the program that produced the proof. The proof can be presented as a sequence of "records". The m clauses in the original formula (input clauses) are indexed 1 through m in this sequence. After that, each record consists of its index in the sequence, the clashing literal, the two operand clauses (i.e., their indexes), and the contents of the resolvent clause. The index of this record identifies the derived clause for subsequent references. The format, called `%RES`, is summarized in Appendix A.1 and is specified in detail at the web page given in Section 1.4.

The correctness of the proof can be established merely by applying the definition of the resolution operation to each record in isolation. In theoretical terms, the checking problem is in deterministic log space, a very easy complexity class in practice. It is only necessary to show that a fixed number of storage registers suffices to check that a resolution operation reported in the proof is correct. We may specify that each storage register can hold an integer up to L^2 for proof length L. (Any positive constant might be used instead of 2.) This program is an undergraduate exercise.

The remainder of this section informally describes proof formats that are checkable in deterministic log space for other problems, to provide evidence that this complexity class is adequate for a variety of logic problems. Please refer to texts by Loveland [Lov78], Burris [Bur98], Kleine Büning and Lettmann [KBL99], and Immerman [Imm99] for background and for standard terminology and concepts of logic and complexity theory.

2.1 A Format for First-Order Logic without Equality

Resolution proofs for first-order logic without equality are only moderately more difficult than propositional logic to encode in a way that they can be checked in deterministic log space. We briefly sketch the main ideas.

Note that in the first-order context, atomic formulas take the place of propositional variables, but literals, clauses, CNF formulas, resolution, and refutation are built up in the same way. An *atomic formula* is a predicate symbol with first-order terms as parameters. A *first-order term* is a first-order variable, or a first-order constant, or a first-order function symbol

with first-order terms as parameters. Atomic formulas and terms are best thought of as trees, although they are traditionally represented as strings.

Besides the resolution operation, a new clause may be created through a substitution operation. In this context, a *substitution operation*, written $x \leftarrow t$, where x is a first-order variable and t is a first-order term that does not contain x, consists of replacing all occurrences of x in a clause by t. A substitution may cause two or more distinct atomic formulas to become identical in the derived clause.

Two clauses are resolvable only if they contain syntactically identical clashing literals with opposite signs. This method of presentation is simpler to check than the usual one, in which the substitution is embedded in the resolution operation.

A substitution record for $x \leftarrow t$ consists of its index in the proof sequence, an opcode denoting substitution, the variable x, and the string that encodes t in prefix notation. The Unique Readability Theorem [Bur98, Sect. 3.12.3] ensures that different terms cannot be represented by the same string and no term is a proper prefix of another term. It is straightforward to verify with a fixed number of registers that t does not contain x.

A record for a clause derived by substitution consists of its index in the proof sequence, an opcode denoting derivation by substitution, the single operand clause (i.e., its index), the substitution (i.e., its index), and the resulting clause.

To ensure that the substitution can be verified in deterministic log space, the vocabulary of predicate and function symbols should be disjoint and the arity (number of parameters) required by each predicate or function symbol may be encoded in its name. It is straightforward to verify with a fixed number of registers that the only differences between the operand clause and the clause derived by substitution are that all occurrences of x in the operand clause are replaced by the encoding of t in the corresponding literal in the derived clause. Notice that *parsing* the strings that encode terms is not necessary.

2.2 A Format for First-Order Logic with Equality

Resolution proofs for first-order logic with equality [Lov78, Bur98] are more complicated to encode so that they can be verified in deterministic log space, compared to first-order logic without equality, but the basic ideas are the same. It is necessary to justify a new clause that is created through a replacement operation. In this context, a *replacement operation* replaces *one* occurrence of a subterm s_1 in the operand clause by another subterm s_2, where the equation $s_1 = s_2$ appears earlier in the proof as a unit clause. A record for a clause derived by replacement consists of its index in the proof sequence, an opcode denoting derivation by replacement, the single operand clause (i.e., its index), the offset of s_1 in the operand clause, the equation (i.e., its index), and the resulting clause.

We do not claim that the overhead of outputting a first-order proof in the formats sketched above is negligible or moderate in all cases. If a clause has many literals, outputting it completely after each substitution might be expensive in time and space. In practice, a more condensed proof format might be used by the first-order prover, and a post-processor might expand that into a proof that is verifiable in deterministic log space.

2.3 Background on Decidable Theories

In recent research, planning problems, hardware and software verification problems and others have been encoded as satisfiability problems. There is a substantial difference among

these types of problems, however. For planning problems, the successful outcome is a satisfying assignment, which describes the plan, and is easily checkable. For verification problems, the successful outcome is the *lack of* a satisfying assignment, which is not easily checkable.

A similar situation exists for many decision procedures for special theories: they are refutation procedures. For example, the Nelson-Oppen decision procedures for the quantifier-free theory of equality with uninterpreted functions and the quantifier-free theory of LISP list structure, based on their congruence-closure procedure, negate the theorem, put that into disjunctive normal form (DNF), then check that each disjunct is unsatisfiable [NO80]. Many developments in the same vein have been reported [BGV01, VB01, BDL98, DD01].

Another active topic is combining decision procedures for different theories. The general idea is to transport constraints among underlying theories until the constraints in one of the underlying theories becomes unsatisfiable. Two influential early papers are by Nelson and Oppen [NO79] and Shostak [Sho84]. Some recent work is by Dill and co-authors [BDL96, BDS00].

Interestingly, Ruess and Shankar pointed out that many of these procedures contain essentially the same bug, and proposed a correction [RS01]. The common bug was due to an unsound inference originally published in Shostak's paper [Sho84], and adopted in other software.

De Moura and Ruess [dMR02] describe a somewhat different approach to combining decision procedures, in which the underlying theories contribute constraints as propositional CNF and the unsatisfiability is checked in the latter theory, which is standard "SAT." Recently, the combination of propositional SAT solving with several decision theories has gone under the name SAT Modulo Theories (SMT), and has emerged as a very active area [NOT06].

Our thesis is that decision procedures can and should be designed to be able to output a proof. While *finding* a proof is hard, *checking* a proof is straightforward. (By a "proof" we mean a complete proof, with no steps omitted.) In practice, all underlying theories can produce a resolution proof. We shall argue that outputting a proof does not place an undue burden on the decision procedures.

2.4 Equivalent-Literal Processing

It has been shown that equivalent-literal identification[3] based on binary-clause analysis can be accompanied by a resolution derivation with only a constant-factor increase in the running time of the procedure [VG05b].

Transformations of the propositional CNF formula that are based on equivalent literals can also be justified by resolution with little overhead. If $a = b$ has been derived, and a is chosen as the leader, use $[\neg b, a]$ to resolve out b and use $[b, \neg a]$ to resolve out $\neg b$. Implementation issues are discussed elsewhere [VG02c]. Two strategies are (1) to resolve out b and $\neg b$ throughout the formula as soon as $a = b$ is derived, and (2) to record the information in a disjoint-sets data structure and wait until the clause is needed for the proof. Both approaches have the potential to incur sizable overhead *whether or not the procedure outputs a proof*.

[3] Equivalent-literal identification consists of determining that two distinct propositional literals must both be true or both be false in any satisfying assignment.

2.5 Extracting a Proof from Congruence Closure

The Nelson-Oppen congruence-closure procedure can be viewed as a *strategy* for choosing proof operations. We will sketch the main idea for the quantifier-free theory of equality with uninterpreted functions. In this case, the proof operations are substitutions into the equality axioms and resolution. For simplicity, let us assume that the formula whose inconsistency is to be established contains a unary function f and a binary function g and is in disjunctive normal form (DNF). Actually, a separate resolution refutation is constructed for each conjunction in the DNF formula.

The notation τ_i represents some term (including subterms) that occurs in the formula. Such a term is made from the above function symbols, variables and constants. We have the first-order equality axioms in the background (i.e., when we talk about terms and atoms in the formula, we exclude these):

$$x = x \tag{1}$$

$$y \neq x \vee x = y \tag{2}$$

$$x \neq y \vee y \neq z \vee x = z \tag{3}$$

$$v \neq x \vee f(v) = f(x) \tag{4}$$

$$v \neq x \vee w \neq y \vee g(v, w) = g(x, y) \tag{5}$$

For each conjunction of equality literals (positive and negative) a dynamic equivalence relation ($\equiv$) is constructed by starting with the positive equalities: if $\tau_1 = \tau_2$ is a positive literal, set $\tau_1 \equiv \tau_2$. For any positive atom $f(\tau_1) = f(\tau_2)$ in the conjunction such that $\tau_1 \equiv \tau_2$, make $f(\tau_1) \equiv f(\tau_2)$ *and output the following proof steps*:

1. Let τ_3 be the leader of the equivalence class containing τ_1 and τ_2. Apply the substitution $x \leftarrow \tau_1$, $y \leftarrow \tau_3$, $z \leftarrow \tau_2$ to (3), creating

$$\tau_1 \neq \tau_3 \vee \tau_3 \neq \tau_2 \vee \tau_1 = \tau_2. \tag{6}$$

2. Resolve (6) with unit clauses $\tau_1 = \tau_3$ and $\tau_3 = \tau_2$, which are either in the input formula or were derived earlier as part of a congruence operation. This outputs the unit clause

$$\tau_1 = \tau_2. \tag{7}$$

3. Apply the substitution $v \leftarrow \tau_1$, $x \leftarrow \tau_2$ to (4), creating

$$\tau_1 \neq \tau_2 \vee f(\tau_1) = f(\tau_2). \tag{8}$$

4. Resolve (8) and (7) to create the unit clause

$$f(\tau_1) = f(\tau_2). \tag{9}$$

5. Apply the substitution $x \leftarrow f(\tau_2)$, $y \leftarrow f(\tau_1)$ to (2), creating

$$f(\tau_1) \neq f(\tau_2) \vee f(\tau_2) = f(\tau_1). \tag{10}$$

6. Resolve (10) and (9) to create the unit clause

$$f(\tau_2) = f(\tau_1). \tag{11}$$

For any positive atom $g(\tau_1, \tau_3) = g(\tau_2, \tau_4)$ in the conjunction such that $\tau_1 \equiv \tau_2$ and $\tau_3 \equiv \tau_4$, make $g(\tau_1, \tau_3) \equiv g(\tau_2, \tau_4)$ *and output proof steps similar to those described above for* f.

Repeat these congruence operations until no new congruences can be found. Then, if any negative atom $\tau_1 \neq \tau_2$ is found in the conjunction, such that $\tau_1 \equiv \tau_2$, simply resolve it with $\tau_1 = \tau_2$, which must have been derived during congruence closure on this conjunction, to produce the empty clause. If no such literal is found, the formula is satisfiable, as described in the original paper [NO80].

A point we glossed over is that the clause $\tau_1 = \tau_3$ might not have been output as soon as τ_3 became the leader of the equivalence class for τ_1. However, the first time `find(`τ_1`)` is called, as a by-product of path compression, this clause can be derived.

Notice that the overhead of creating the proof only increases the constant factor for the entire decision procedure. It is completely mechanical. The technique is not limited to a strategy that is based on DNF. It can be adapted to any strategy for application of the equality axioms.

3 Extracting Resolution from a Conflict Graph: Pitfalls to Avoid

This section discusses pitfalls that were discovered in one existing solver and greatly limited its capability to produce proofs. A "Special Edition" of `zchaff`, called `zchaffSE` in the tables later in this paper, is dated March 2005 and was entered into the verified-unsatisfiable track of the SAT–2005 competition. It combines the functionality of `zverify_df` and `zchaff` [ZM03b] into a single program that writes the full resolution derivation in the binary format specified for SAT–2005. This section shows that the proof generated can have length that grows exponentially in the length of the conflict graph (described below). Sections 4 and 5 show that the problem is easily fixed. A preliminary version of this section appeared in a short conference paper [VG07].

3.1 A Quick Review of Conflict Graphs

Most, if not all, leading SAT solvers use a *conflict graph* data structure to infer (i.e., "learn") *conflict clauses*. They are often called *conflict-driven clause-learning* (CDCL) solvers. This section reviews conflict graphs briefly, for self-containment. The basic concepts are well developed in the literature [MSS99, ZMMM01, BKS04]. Figure 1 illustrates a conflict graph. Further details are given after introducing some terminology.

Definition 3.1 The *conflict graph* is a logical description of a data structure built by a CDCL solver. Literals are added to an initially empty graph as they are assigned *true* by the solver. Each graph vertex is associated with a different literal, no complementary literals appear as vertices, and the *conflict vertex* is associated with $\bot$, which represents the constant *false*. Each vertex is either an *assumed literal* or an *implied literal*, possibly $\bot$. The vertex for each implied literal, including $\bot$, is labeled with a clause, called the *antecedent clause*. Antecedent clauses are clauses in the original formula or clauses derived earlier and recorded (learned). (The antecedent of $\bot$ is sometimes called the *conflicting clause*, or *conflict antecedent*. This clause should not be confused with the *conflict clause*, defined in Definition 3.3.) The antecedent clause C for implied literal p is the first clause noticed by the solver such that all literals in C other than p have been assigned *false*, either by assumption

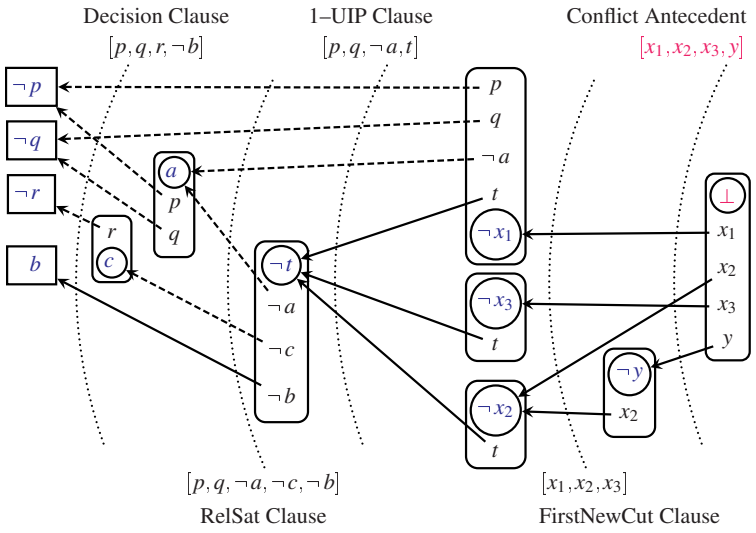

Fig. 1 Conflict graph with several cuts shown.

or implication. In other words, if $C = [p, \overline{q_1}, \ldots, \overline{q_k}]$ ($k \geq 0$), then each of $q_1, \ldots, q_k$ is already in the conflict graph when p is added. There is a directed edge from p to each q_j, $1 \leq j \leq k$.

Assumed (guessed) literals, commonly called "decision literals," do not have an antecedent clause and have no outgoing edges.

The term *implication graph* is used to refer to the graph that contains all assumed and implied literals, and does not necessarily contain $\perp$. Only literals that are reachable from $\perp$ are included in the *conflict graph*. During backtracking, as literals become unassigned, they and their outgoing edges are deleted from the implication graph. □

Our notation varies from some other papers [ZMMM01, BKS04] to better reflect the actual data structures used by the programs; e.g., arrows indicate the reference direction in the data structures rather than the implication direction. Also, the conflict vertex is associated with a clause that became *empty* during unit-clause propagation (containing y in Figure 1), in agreement with the data structures and the original presentation [MSS99], rather than showing separate vertices for the conflicting implied literals and leaving it ambiguous which antecedent clause actually became empty.

Definition 3.2 A *database clause* for a conflict graph is either an *input clause* (a clause of the original formula), or a clause that was derived and recorded prior to completion of this conflict graph. □

Definition 3.3 A *conflict-generating cut* (*cut* for short) in a conflict graph is a partition of the vertices into two sets, called the *conflict side* and the *reason side*, such that the conflict vertex is on the conflict side, all decision literals are on the reason side, and all other vertices are on one of these two sides, subject to the restriction that all vertices on the conflict side are reachable from the conflict vertex via a path that is entirely on the conflict side. An edge from the conflict side to the reason side is said to *cross the cut*. An edge from the reason side

to the conflict side is possible, but has no special status and is not considered to cross the cut for our purposes. The cut is sometimes identified with the set of edges crossing the cut.

Given a conflict-generating cut, the associated *conflict clause* consists of the negations of all vertices on the reason side that have an incoming edge from the conflict side of the cut.

Although cuts that are not conflict-generating cuts are mentioned in the literature, they are little used; all cuts in this paper are conflict-generating cuts. □

The foregoing definitions are illustrated in Figure 1. The vertex for each implied literal (in circles), including $\bot$, is labeled with a database clause, called the *antecedent clause*. The antecedent clause is shown within the rounded box that represents the entire vertex. *Assumed literals* (in square boxes), also called *decision 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.

Dotted arcs show several cuts discussed in the literature. Adjacent to each arc is the associated conflict clause and the name of the cut. This paper is concerned primarily with the 1-UIP cut used by most CDCL solvers, and defined next.

Definition 3.4 A *unique implication point* (UIP) in a conflict graph is a vertex whose literal was assumed or implied at the current decision level such that all paths from the conflict vertex to the literal that was assumed at the current decision level pass through this vertex [MSS99]. In other words, if the UIP literal had been assumed at the current decision level and followed by unit-clause propagation, it could have led to the conflict that actually occurred.

The *UIP cut* places all vertices whose literals were implied subsequently to the UIP on the reason side, and places all other vertices (including the UIP itself) on the conflict side. Vertices on the reason side are necessarily at the current decision level.

The *first unique implication point* (1-UIP) is the vertex closest to the conflict vertex with the UIP property. The 1-UIP cut is the UIP cut associated with the 1-UIP vertex. □

In Figure 1 the 1-UIP vertex is $\neg t$ and the conflict side of the 1-UIP cut consists of the vertices $\neg x_1$, $\neg x_2$, $\neg x_3$, $\neg y$, and $\bot$.

3.2 An Exponential Worst Case

Recent papers have observed the connection between conflict graphs and resolution [GN03, ZM03b, BKS04, VG05a]. Given a cut, the antecedent clauses on the conflict side of the cut (i.e., the side containing $\bot$ as an implied "literal") logically imply the conflict clause, which consists of the negations of those reason-side literals that have a direct edge *from* some vertex on the conflict side. (E.g., in the figure, for the first UIP cut, $\neg p$, $\neg q$, a, and $\neg t$ have edges from 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 on the conflict side. Also, it is easy to see that the set of antecedent clauses is Horn-renamable [VG09], so the problem should be tractable.[4] In the discussion, we shorten "resolution derivation" to "derivation".

[4] Recall that a Horn clause has at most one positive literal, and renaming consists of inverting a subset of the variables in the set of clauses so that they all become Horn.

```
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.

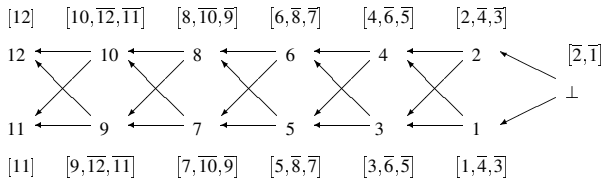

Fig. 3 DAG family with exponential worst case for `zverify_df`, version dated 2004.11.15. Antecedent clauses are shown in brackets. This graph is for $h = 6$.

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, for the version dated 2004.11.15, which stood as the current version for more than two years.

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 $\bot$, the database clause that became empty during the unit propagation(s) that generated the conflict graph (see "Conflict Antecedent" in Figure 1). The *conflict clause* is the final value of `cl`. The figure shows the procedure specialized for generation of the *last* conflict clause, which must be empty if the solver was correct.

More generally, `choose_literal` may "fail" if all remaining literals of `cl` are on the reason side of the cut, causing the `while` loop in lines 2–6 to exit with `cl` nonempty. An invariant in all cases is that `cl` contains the *negations of* some literals in the conflict graph.

Line 3 is implemented two ways in different versions. In an early version, the literal chosen is one in `cl` with maximum DAG height. The *DAG height* of a vertex v is defined as the maximum path length from v to a vertex whose antecedent is a unit clause (zero if the antecedent of v is a unit clause, -1 if v is a decision literal, or is otherwise determined to be on the reason side of the cut). In the subsequent version, dated 2004.11.15, the literal chosen is simply the one with the largest variable number. This is essentially an arbitrary choice, designed to save the cost of computing DAG heights.

We now present a conflict graph family that generates exponential behavior in the size of the conflict graph for the `zchaff` version dated 2004.11.15. The parameter of this family is h, the DAG height of the conflict vertex (which contains $\bot$ as an implied "literal"). A member with $h = 6$ is shown in Figure 3. The generation rule for other h should be obvious from this example.

The refutation developed by Figure 2 for the conflict graph in Figure 3 resolves 2^{h-1} times upon each of the literals $2h$ and $2h-1$; it resolves 2^{h-2} times upon each of the literals $2h-2$ and $2h-3$, etc.; the total number of resolutions is $2\,(2^h - 1)$. This refutation begins

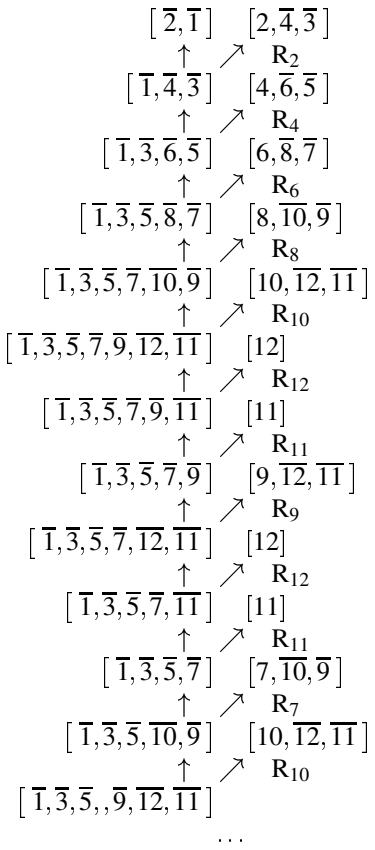

Fig. 4 Initial part of resolution refutation developed by Figure 2 choosing largest numbered variable on line 3. "R_q" denotes resolution with clashing literal q.

as shown in Figure 4. In this example, $h = 6$, and we see that variables 12 and 11 have already been resolved upon twice each, and have been re-introduced yet again in the clause on the last line.

This is a case where theory translates into practice, at least in the case of the simple choice of literal in the `zchaff` version dated 2004.11.15 (it chooses the largest numbered variable). 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 9. Those labeled "`IBM_FV`" are from the "industrial" category of the SAT 2005 Competition.[5]

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

[5] See `http://www.satcompetition.org/2005`.

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

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.

4 Avoiding the Pitfall via "Trivial Resolution" (TVR)

This section shows that the pitfall exposed in Section 3 can be overcome using already-published techniques. Indeed, `zchaffSE` and `zverify_df` were modified with a few book-keeping changes to cure the performance problem. Section 5 discusses an alternative fix. A preliminary version of this section appeared in a short conference paper [VG07].

Several restricted forms of resolutions have been defined and studied over the years. A *linear derivation* is one in which the first clause is a database clause, called the *top clause*, and each resolution operation has the previous clause in the derivation as one operand; the other operand, called the *side clause*, may be a database clause or an earlier-derived clause of the linear derivation. (For our purposes, as in Definition 3.2, a database clause is either a clause of the original formula or a clause that was derived *before* the current linear derivation began.) It is well known that linear resolution comprises a complete system, i.e., every unsatisfiable set of clauses has a linear refutation [AB70]. In this context, "linear" refers to the form of the derivation, not the number of operations.

An *input derivation* is a linear derivation with the further restriction that the second operand must be a database clause; earlier-derived clauses of the same linear derivation are not acceptable. This does *not* provide completeness. Note that Figure 2 provides a framework for input derivations using antecedent clauses.

Beame *et al.* [BKS04] define a *trivial resolution derivation* (**TVR**) to be an input derivation with the further restriction that no clashing literal occurs in more than one resolution operation. They show (their Proposition 4) that the conflict clause defined by any conflict-generating cut (Definition 3.3) in a conflict graph can be derived by a trivial resolution derivation.[6] This TVR uses the antecedent clauses on the conflict side of the cut as the database clauses, and uses the antecedent of the conflict vertex (which has $\perp$ as the implied "literal") as the top clause. For n database clauses the successful TVR has exactly $n - 1$ resolution operations, but using a correct order is crucial.

[6] The paper actually states the Proposition for "any" cut, but the claim does not hold for certain cuts that do not satisfy Definition 3.3 and would be unlikely to occur in any real solver.

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 by the `choose_literal` operation on line 3 of Figure 2.

Definition 4.1 Cut-Crossing Rule [BKS04, reworded]: Choose a literal each of whose incoming edges originates from a vertex whose literal has already been resolved upon, or from the conflict vertex. □

For example, in Figure 1, the clashing literal for the first resolution of a TVR may be $\neg x_1$ or $\neg x_3$ or $\neg y$, but not $\neg x_2$.

4.1 Chaff TVR Order

This section discusses the TVR order used implicitly by the `grasp/chaff` family of solvers, also known as *conflict-driven clause-learning* (CDCL) solvers. The procedure by which this family constructs conflict clauses is described in numerous papers, such as [MSS99, MMZ$^+$01, ZMMM01, BKS04]. As just mentioned, a correct TVR order must satisfy Definition 4.1, but such an order is not readily accessible from the data structure of the conflict graph itself. Some other data structure or computation is needed to provide an appropriate TVR order, and several orders may work, as discussed in Section 4.2.

The saving grace for `zverify_df` is that the input is set up by the companion solver, such as `zchaff`, which already *has* such a data structure. Many solvers, based on the CDCL strategy, create a sequence of "implied" literals in the same order as they are derived and enter the conflict graph, which is called the *chronological order*. (This sequence, sometimes called the *trail*, is needed to undo variable assignments during backtracking.)

At the time a literal p is entered into the conflict graph, all other literals in its antecedent clause have been falsified earlier, hence their negations have been entered into the conflict graph earlier than p. Thus one order that satisfies the cut-crossing rule is the *reverse* chronological order. It follows that TVR is implemented in the CDCL family of solvers, at least implicitly, in the procedure that constructs conflict clauses. (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 9.) Unfortunately, `zchaff` 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 (see Section 4.3 for an alternative) to deliver the rightmost column of Table 1 involved modifying `zchaff` to follow through at decision level zero, mainly using the procedures and data structures already in the program.[7] The fixed program creates an empty conflict clause and encodes its resolution order in the resolve-trace, using the same protocol *already in the `zchaff` code* as used for the nonempty conflict clauses at positive decision levels. Then `zverify_df` was modified slightly to expect this additional line.

[7] Source files are available from the author; "diffs" are available at the URL given in Section 1.4.

	Clause	How Derived
1.	$tmp_1 = [x_2, x_3, y, p, q\neg a, t]$	resolve *ante*$(\bot)$ with *ante*$(\neg x_1)$
2.	$tmp_2 = [x_2, y, p, q\neg a, t]$	resolve tmp_1 with *ante*$(\neg x_3)$
3.	$tmp_3 = [x_2, p, q\neg a, t]$	resolve tmp_2 with *ante*$(\neg y)$
4.	$[p, q\neg a, t]$	resolve tmp_3 with *ante*$(\neg x_2)$

Fig. 5 TVR derivation of the 1-UIP clause in Figure 1. *ante* abbreviates *antecedent*.

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 by a few percent.

On the benchmarks reported in Section 9 it appears that the overhead for producing the "resolve-trace" is now 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.

4.2 TVR Orders in General

The wording of the cut-crossing rule (Definition 4.1) turns out to match well with the standard graph concept of topological order.

Definition 4.2 In a directed acyclic graph (DAG) with n vertices, a *topological order* on the vertices is any one-to-one assignment of the numbers 1–n to the vertices (the *topological number* of a vertex is the number so assigned) assigned such that all edges go from a vertex with a lower topological number to a vertex with a higher topological number. □

Lemma 4.3 An ordering on the vertices on the conflict side of a conflict graph satisfies the cut-crossing rule in Definition 4.1 if and only if it is a topological order (using edge orientations illustrated in Figure 1).

Proof: ($\Rightarrow$) If the order satisfies the cut-crossing rule, then when v is chosen for the TVR, every vertex with an edge to v was previously chosen, so has a lower topological number. Therefore this is a topological order.

($\Leftarrow$) If the order is a topological order, then when v is chosen for the TVR, every vertex with an edge to v has a lower topological number than v, so it was chosen earlier for the TVR. Therefore the cut-crossing rule is satisfied. □

An easy way to compute a topological order is to perform a depth-first search rooted at the conflict vertex, and push the vertices onto a stack at "finishing time," also called "postorder time" [BVG00]. Only vertices on the conflict side of the cut are visited. Then reading from top to bottom provides a topological order.

Example 4.4 Figure 5 shows one possible TVR derivation of the 1-UIP clause in Figure 1, which is $[p, q, \neg a, t]$. □

4.3 Alternative Zchaff Fix

To complete the history of `zchaff`, in early 2007 Zhaohui Fu of the Princeton `zchaff` team discovered that a procedure called "levelizing" in `zverify_df` had been disabled in the release dated 2004.11.15 (and still current at that time), in the belief that it was an heuristic that seemed not to provide a net time benefit. Restoring this procedure also cured the `zchaffSE` performance problem reported in Section 3.

The purpose of "levelizing" was not reported in any papers or explained in comments in the code, but a close look at the code showed that it sorted the vertices into a topological order for the conflict graph at decision level zero. Thus, by Lemma 4.3, `zverify_df` with "levelizing" finds a TVR at decision level zero, as well as at positive decision levels.

Experiments reported in Section 9 had already been performed using the fix originated by the author, but later experiments confirmed that restoring "levelizing" resulted in essentially identical performance improvements.

4.4 TVR Proof Length

We now analyze the range of total derivation sizes for TVR in some detail, so they may be compared with PUP derivations, discussed in Section 5. The comparison is carried out in Section 6. We introduce some terminology and notation to study efficient methods of extraction, with worst-case guarantees.

Definition 4.5 Whatever cut is chosen by the solver, assume that vertices on the reason side of the cut that are *not* touched by an edge crossing the cut have been trimmed out, and all edges between two reason-side vertices also have been trimmed out. (Negations of the remaining vertices on the conflict side comprise the conflict clause.)

We shall use the notation that n is the number of vertices in the conflict graph on the conflict side of the cut, m is the number of edges, d is the 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, which we take as the *size* of the conflict graph. Thus $(m+w)$ is the combined length of the input and output of the extraction procedure without a proof. □

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 (clause `cl` at line 6 in Figure 2) to grow by up to $(d-1)$ literals per resolution for the first $t = \lfloor (n+w)/d \rfloor$ steps to a size of $(w+n-1-t)$, even if the final conflict clause is fairly narrow. (Possibly $t = n-1$.) Thereafter, it can shrink only one literal per resolution, and it must reach size w after the $(n-1)$-th resolution.

The following parameterized formula provides an example. The first literal in each clause is the implied literal, and clauses are in the chronological order of their implied literals. Literals with p are on the conflict side, those with q are in the conflict clause being derived. Studying all details of the formula is not necessary to follow the analysis.

$$C_1 = [p_1, \neg q_1] \tag{12}$$

$$C_i = [p_i, \neg p_{i-1}] \qquad \text{for } 2 \le i \le n-t-1 \tag{13}$$

$$C_{n-t} = \left[p_{n-t}, \neg p_{n-t-1}, \neg q_{1+(d-1)t}, \ldots, \neg q_w\right] \tag{14}$$

$$C_j = \left[p_j, \neg p_{j-1}, \neg q_{1+(d-1)(n-j)}, \ldots, \neg q_{(d-1)(n-j+1)}\right]$$

	Clause	How Derived
1.	$pup(\neg y) = [\neg y, t]$	resolve $ante(\neg y)$ with $pup(\neg x_2)$
2.	$tmp_1 = [x_2, x_3, y, p, q\neg a, t]$	resolve $ante(\bot)$ with $pup(\neg x_1)$
3.	$tmp_2 = [x_3, y, p, q\neg a, t]$	resolve tmp_1 with $pup(\neg x_2)$
4.	$tmp_3 = [y, p, q\neg a, t]$	resolve tmp_2 with $pup(\neg x_3)$
5.	$pup(\bot) = [p, q\neg a, t]$	resolve tmp_3 with $pup(\neg y)$

Fig. 6 A PUP derivation of the 1-UIP clause in Figure 1. Note that *pup* is the same as *ante* for $\neg x_1$, $\neg x_2$, and $\neg x_3$ because all edges leaving these vertices cross the cut.

$$\text{for } n - t + 1 \leq j \leq n - 1 \tag{15}$$

$$C_n = [\bot, \neg p_{n-1}, \neg q_1, \ldots, \neg q_{d-1}] \tag{16}$$

This formula has only one TVR, in which C_n is the top clause and the other clauses appear as side clauses in reverse index order. The first t antecedents to be resolved are width $d + 1$ and the remaining antecedents are width 2. For this worst case the sum of the sizes of all resolvents in the derivation is given by

$$\text{TVR total literals} = \left(d + \tfrac{1}{2}(d-1)t\right)(t-1) + w(n-t) + \tfrac{1}{2}(n-t)(n-t-1) \tag{17}$$

This can be quadratic in m, the size of the conflict graph. For example, if $d = w = t = \sqrt{n}$, then $m \approx 1.5\, n$ and TVR total literals $\approx \frac{1}{2}\, n^2$.

It is also easy to generate examples for which TVR total literals is about m. In any conflict graph with at least one assumption literal $m \geq n + w - 2$. However, n can be as small as 2. For an extreme example, let q be the assumption literal on the current decision level and let the two clauses be $[p, \neg q, \alpha]$, where α is a sequence of literals assigned at earlier decision levels, and $[\neg p, \neg q]$. In the conflict graph, let the first clause be the antecedent of p, and let the second clause be the antecedent of $\bot$. Then $m = |\alpha| + 3$, which is also the TVR total literals.

5 Avoiding the Pitfall via Pseudo-Unit Propagation (PUP)

Although TVR uses the minimum number of resolution *operations* (exactly one for each vertex other than $\bot$ 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**). Like TVR, PUP works for any conflict-generating cut. 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. A preliminary version of this section was presented in a short paper at a symposium [VG08].

We define a *pseudo-unit clause on* p at a particular vertex p of the conflict graph to be a clause consisting of the literal p 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.

If the solver uses a UIP cut (Definition 3.4), then, among the literals of the pseudo-unit clause that are also in the conflict clause, *one* might be the negation of a UIP literal, 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 conflict vertex, or other means [BVG00]. If a depth-first search is rooted at the conflict vertex, then the vertices reach their postorder times in a reverse topological order, and the extraction of the PUP derivation can be embedded in this DFS. (If a UIP cut is in use, only vertices that were implied at the current decision level and later than the UIP will be visited.) When visiting vertex p, at postorder time, derive a pseudo-unit clause on p, which we'll call *pup*(p), as described just below.

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 PUP closely mimics the solver's unit-clause propagation, so is a natural order in this sense.

The invariant that is maintained is that all literals in *pup*(p) other than p itself are also in the conflict clause associated with the cut.

To derive *pup*(p), suppose the antecedent clause is *ante*$(p) = [p, \neg r_1, \ldots, \neg r_k]$. Then for $1 \leq i \leq k$, either $\neg r_i$ is in the conflict clause being derived or r_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*(p). If no r_i are on the conflict side, we are done, and *pup*(p) = *ante*(p). Otherwise, for each r_i on the conflict side, resolve *pup*(r_i), which is already computed, with the current resolvent in the linear derivation. Each resolution can only introduce literals that are in the conflict clause into the current resolvent, and of course $\neg r_i$ is removed. The current resolvent cannot become tautologous, since the conflict clause is not tautologous. The final resolvent is *pup*(p).

At the conflict vertex, $\bot$ can be discarded from *pup*$(\bot)$, 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.

Example 5.1 To illustrate and clarify PUP, Figure 6 shows a PUP derivation of the 1-UIP clause in Figure 1, which is $[p, q, \neg a, t]$. Notice that x_2 is the clashing literal twice, so this is not a TVR derivation. □

With reasonable assumptions about data structures already present in the solver, the derivation of the pseudo-unit clause on p can be performed in time proportional to the sum of the lengths of intermediate clauses derived on the way to the pseudo-unit clause. We use the notation and conventions of Definition 4.5.

For the proof of *pup*(p) each intermediate clause has at most $(k + w)$ literals, where k is the number of edges from vertex p to some other conflict-side vertex (i.e., the antecedent clause has up to $(w + k + 1)$ literals). Each edge *to* a conflict-side vertex is associated with one resolution step. Therefore, a simple and possibly loose upper bound for the entire proof of *pup*$(\bot)$ is $m\,(n + w) \leq m^2$.

The sum of the lengths of clauses in one proof of *pup*(p) 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 $(w + 1)$. However, the analysis is complicated by the fact that this upper bound cannot be reached at all vertices for all combinations of values for d, w, m, and n. Therefore we consider several special cases.

For notation, let the vertices on the conflict side be $p_1, \ldots, p_n$, in their order of implication by the solver. Let k_i be the number of edges from vertex i to some other conflict-side vertex j; necessarily, each $j < i$ and $k_i < i$. Note that *pup*(1) = *ante*(1), so it is not counted in the proof length. Let $h = \sum_{1 \leq i \leq n} k_i$; this is the number of edges *to* a conflict-side

vertex and is the number of resolution operations for the entire proof of *pup*($\bot$). We have $m \geq w + h$.

First, consider $d \geq w \geq n$. The following parameterized formula provides a worst case for PUP (within a constant factor). The first literal in each clause is the implied literal, and clauses are in the chronological order of their implied literals. Literals with p are on the conflict side, those with q are in the conflict clause being derived. Studying all details of the formula is not necessary to follow the analysis.

$$C_1 = [p_1, \neg q_1, \ldots, \neg q_w] \tag{18}$$

$$C_i = [p_i, \neg p_1, \ldots, \neg p_{i-1}] \qquad \text{for } 2 \leq i \leq n-1 \tag{19}$$

$$C_n = [\bot, \neg p_{n-1}, \neg p_1, \ldots, \neg p_{n-1}, \neg q_1] \tag{20}$$

In this formula, *pup*(2) is computed with one resolution and has length $w+1$. Subsequently the length of the derivation of *pup*(i) is $k_i\,(w+(k_i+1)/2)$, as stated above, where $k_i = i-1$. The sum of these lengths is about $(w\,n^2/2 + n^3/6)$.

$$\text{PUP total literals} \approx \frac{w\,n^2}{2} + \frac{n^3}{6} \tag{21}$$

However, $h \approx n^2/2$ and $m > h + w$, so the total length is maximum as a function of m for $w \approx \frac{1}{2}\,n^2$ and $m \approx n^2$, giving

$$\text{PUP total literals} \approx m^2/4 \tag{22}$$

in the worst case.

Next we consider $d \ll n$ and $w \ll n$. Since each edge is associated with at most one resolution and each resolvent has width at most $(d + w)$,

$$\text{PUP total literals} \leq m\,(d + w) \tag{23}$$

If d and w are considered to be constant, the proof is linear in m.

6 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 quadratic worst cases. We now show that the constructions leading up to the bounds given in Eqs. 17 and 22 provide examples for which either method might be better than the other for a particular conflict graph by a linear factor.

For (17) the antecedents in forward chronological order were a series of $n-t-1$ binary clauses, followed by $t \approx n/d$ clauses of width about d, giving $m \approx 3\,n$ (see Eqs. 12–16). TVR had quadratic behavior if d and w are constants. We see that *pup*(p_1) = *ante*(p_1) and *pup*(p_i), $2 \leq i \leq n-t-1$, are binary, derived with one resolution each. The remaining t *pup* clauses require only one resolution apiece because each has only one edge to a conflict-side vertex. They contribute at most $t\,(w+1)$ in proof length. Thus the PUP derivation is linear in m if d and w are constants.

For (22) the vertices p_2 through p_n have edges all lower numbered conflict-side vertices and no other edges, except that vertex p_n has one edge to q_1 on the reason side (see Eqs. 18–20). PUP had quadratic behavior if $w = \frac{1}{2}n^2$ and $d = w$. For TVR successive current resolvents shrink by one literal at a time. The top clause is C_n in (20). The first derived clause is $[\neg q_1, \neg p_1, \ldots, \neg p_{n-2}]$, and the j-th derived clause is $[\neg q_1, \neg p_1, \ldots, \neg p_{n-j-1}]$,

for $2 \leq j \leq n-2$. Finally, C_1 in (18) has negations of all w literals in the conflict clause, so the final resolution produces the conflict clause with $\frac{1}{2}n^2$ literals. The sum of the lengths of all TVR resolvents is about n^2, the same as m.

The high-level picture is that TVR wins when the conflict side has a moderate number of very long clauses and PUP wins when the conflict side has a large number of moderately short 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 the fly" 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-the-fly resolutions that turn out to be unneeded), compared to the after-the-fact TVR method.

7 Reverse Unit Propagation (RUP) Proofs

Reverse Unit Propagation (**RUP**) proofs are based on the idea of *conflict-clause proofs* from Goldberg and Novikov [GN03]. This section develops the basic idea and explores its theoretical properties. The language of correct RUP proofs is P-complete, which is widely believed to have complexity higher than deterministic log space. However, it is shown that a combination of P-complete procedures to generate a sequence of resolution proofs and deterministic log space procedures to verify each of them is able to verify correctness of a RUP proof. A preliminary version of this section was presented in a short paper at a symposium [VG08].

7.1 RUP Inferences

Definition 7.1 A clause $C = [p_1, \ldots, p_k]$ is a *RUP inference* from formula F if adding the unit clauses $[\neg p_i]$, 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 the same order that they were derived, giving a sequence of formulas, $F_j = F_{j-1} \cup \{C_j\}$. They stated (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. (Although they did not prove this, they probably envisioned a PUP proof of the empty clause for each RUP inference, as described in Section 5, and considered the observation to be obvious.) Later, Beame *et al.* [BKS04] proved the following, more general, proposition (again, using different terminology, their Propositions 3 and 4).

Proposition 7.2 A conflict clause defined by any conflict generating cut (Definition 3.3) 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 can be derived with a "trivial resolution derivation," as discussed in Section 4. □

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* [VG02b]. Conflict analysis (also called clause learning and nogood inference) is post-order. However, binary-clause analysis, equivalent-literal analysis, failed literal analysis and similar operations are pre-order. (Recently the term "look-ahead" has been used for pre-order reasoning.) 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 1.4 and elsewhere), which was offered in the verified-unsatisfiable track of the SAT–2007 solver competition, and will continue to be used in the future.

7.2 RUP and Conflict Clause Minimization

Another important observation is that *conflict-clause minimization* falls into the RUP framework. Conflict clause minimization was briefly introduced in a simple form by Beame *et al.* [BKS04]. A more elaborate form was implemented in `MiniSat` by Eén and Sörensson, but their only joint report is a poster at SAT 2005.[8] Recent papers analyze the `MiniSat` algorithm, now called *recursive conflict-clause minimization*, and it is now included in many solvers. Experimental data shows that recursive conflict-clause minimization is quite beneficial to performance on "industrial" SAT problems [Bie08, SB09, VG09]. Although the `MiniSat` algorithm does not correspond to a TVR derivation, an efficient procedure for deriving the same "minimized" conflict clause is now known [VG09]. Here, "minimized" is put in quotes because an even stronger clause might be derivable using clauses that are in the database, but not in the conflict graph [ABH$^+$08].

Recall that the conflict clause associated with a cut consists of the negations of the literals on the reason side that are reachable from the conflict side by a single edge. Starting from a particular cut (usually the 1-UIP cut), it is sometimes possible to shift some literals from the reason side to the conflict side, defining a new cut whose conflict-clause literals are a proper subset of those in the starting conflict clause. Note that all the shifted literals are effectively resolved away.

For a simple example, see Figure 1. Suppose the starting conflict clause is the one based on 1-UIP, i.e., $[p, q, \neg a, t]$. Shifting literal a to the conflict side creates a new cut whose conflict clause is $[p, q, t]$. In this case, resolving the antecedent of a with the 1-UIP clause immediately produces a stronger clause. In more complicated cases, early resolutions add new literals and later resolutions resolve the new literals away, finally obtaining the stronger clause, provided that at least one original literal was resolved away somewhere along the line.

In view of the experimentally observed performance benefits, it is useful to realize that Proposition 7.2 implies both that the "minimized" conflict clause is a RUP inference and that it has a TVR derivation.

[8] See `http://minisat.se/Papers.html`.

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 2); i.e., the program that needs to be trusted verifies a proof language that is in *deterministic log space*.

Recall that a language is said to be *logspace complete for* P (P*-complete* for short) if every language in deterministic polynomial time (P) has a deterministic log space reduction to this language, a concept introduced by Cook although the first paper did not coin this term [Coo74]. It is widely believed that P-complete problems are not in deterministic log space or even $\log^k$ space for any fixed k [Coo74], [Imm99, Sect. 2.6], [Sip07, Sect. 8.6]. The latter two texts also mention that the Circuit Value Problem is P-complete. Recall that the input for the Circuit Value Problem is an encoding of the logic gates and the values of the inputs, and the question is, does the circuit output a 1.

It is easy to see that verifying a RUP derivation of a single clause is P-complete by reduction from the Circuit Value Problem. Encode the circuit as clauses with output gate named z, and input values represented by unit clauses on the literals inputs $q_1, \ldots, q_k$. Then assert that the clause $[z, \neg q_1, \ldots, \neg q_k]$. is RUP-derivable.

At first blush it appears that, as a practical matter, verifying an entire RUP proof would be in a higher complexity class than deterministic log space. We claim that our supporting software provides sufficient extra information to permit a RUP proof to be verified in deterministic 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. The first program composes two sequences of extended formulas,

$$\left.\begin{array}{l} F_j = F_{j-1} \cup \{C_j\} \\ G_j = F_{j-1} \cup \{\overline{C_j}\} \end{array}\right\} \qquad \text{for } 1 \leq j \leq k, \tag{24}$$

where $\overline{C_j}$ denotes the set of unit clauses obtained by negating the disjunction 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.

The second program solves a P-complete problem, so for practical purposes, it does not fit 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, (24). Even though the first program could be implemented using logarithmic working storage, it is unnecessary to trust it. (We do not even *think* about trusting the actually implemented 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: the unsatisfiability of each G_j can be verified independently. Also, as pointed out by Goldberg and Novikov, by starting the checking at index k and working backwards to 1, each RUP clause C_i that is referenced while verifying G_j can be marked (C_k is marked initially).

When it is time to verify G_j, if C_j is unmarked, this verification can be skipped. Unfortunately, it is not straightforward to use this optimization in conjunction with parallelization.

8 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, as discussed at the end of Section 7.1. Section 7.3 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 describe how 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.* [BKS04] holds (see Proposition 7.2 in Section 7.1). This provides hope that we can have our cake (an easy and flexible RUP implementation) and eat it too (efficiently check the output). A preliminary version of this section appeared in a short symposium paper [VG08].

Continuing with the notation of Section 7.3, let $C_j = [\neg q_i,\ 1 \leq i \leq w]$ 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 form G_j as follows: put unit clauses q_i ($1 \leq i \leq w$; this is the negation of C_j) in the unit-clause queue (queue, for short) for further unit-clause propagation. Let unit-clause propagation run to completion. All unit-clauses derived (including $[\bot]$), both during this process and during the "preprocessing" of F_{j-1}, are possible vertices of the resulting conflict graph. If $\bot$ is not derived, C_j does not qualify as a RUP inference. Otherwise, the actual conflict-graph vertices are those reachable from $\bot$ through a chain of antecedents.

As described in Section 4 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 q_i in the queue need to be backed out; earlier-derived unit clauses can be kept and re-used for the verification of 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 code base. The following brief description of this implementation assumes familiarity with the operation of solvers in the CDCL family.

The main idea is to enqueue the unit clauses q_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 q_i's do not have antecedents. If some of the q_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 (q_1, q_2, ...) to be at a *higher* decision level. But the conflict analysis uses first UIP, and would not derive the desired clause if each q_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 in `zchaff`.

However, by labeling all the q_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.

9 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 as well as comparison of various formats. An abbreviated version of subsections 9.1 and 9.2 was presented in a short paper at a conference [VG07].

A central question is whether RUP proofs can viably be post-processed into an explicit resolution format, relieving implementers of the burden of producing an explicit format directly out of their solvers.

The *conflict-clause proofs* of Goldberg and Novikov have been discussed in Section 7.1. The RUP format consists of one or more lines per RUP clause, in the standard ASCII DIMACS format, i.e., 0-terminated.

The *resolve-trace* format reported by Zhang and Malik [ZM03b] 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 4. Recall that the order for a trivial resolution is not unique, in general, but also is not arbitrary. 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 [SB06, Bie08] is essentially the union of the RUP and resolve-trace formats (with an option to omit the RUP part). However, the Sinz and Biere trace format does *not* imply a correct TVR order, only the *set* of clauses needed.

A "Special Edition" of `zchaff`, call it `zchaffSE`, is dated March 2005 and was entered into the verified-unsatisfiable 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 Section 3.2.) 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 in Section 4, 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

Table 2 Proof length comparisons. Sizes are thousands of literals, numbers, variables, or clauses. "Numbers" means how many numbers (indexes) comprise the resolve-trace proof. "Ratio" means numbers or literals for the proof method divided by literals for 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

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 8, 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* (on disk) as an array of 32-bit ints on x86 (little-endian) architectures. This allows the program to operate on files much larger than the computer's virtual memory.

Computations were done on 64-bit dual-core AMD Opteron systems with either 2.6-GHz CPUs and 8 GB of real memory (Sections 9.1–9.2), or 2.0-GHz CPUs and 16 GB of real memory (Section 9.3). 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 163 GB proof in 17 CPU minutes (see IBM···rb30_Sd.k15 in Tables 4 and 5), the CPU utilization was only 5%, and the elapsed time was 5.3 hours. The CPU was idle for most of the remaining 95%, waiting for disk fetches. 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; they are labeled "GN03" and were furnished by those authors. Many of the same benchmarks 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 in seconds unless stated otherwise.

9.1 Space Comparisons

The first question is how proof lengths compare for the various formats. Simply comparing file lengths would be misleading because some formats are binary and others are ASCII; consequently we we measure proof size by how many integers are needed to represent a proof. In most cases disk space is more of a limiting factor than time. Table 2 shows zchaff results in our runs, for the 27 benchmarks used by Goldberg and Novikov (their numbers are not repeated here). Benchmark by benchmark there are wide fluctuations between their number and ours. This is not surprising, because they reported on BerkMin, while we report on the zchaff strategy. Overall, their "conflict-clause proofs" and our RUPs are of comparable sizes. We also checked the corresponding resolve-trace sizes reported by Zhang and Malik, but found no meaningful correlations with our observations, which are based on zchaff 2005.11.15; apparently zchaff underwent extensive tuning since their 2003 paper.

Our first new finding is that the full resolution proofs produced by zchaffSE are 100 times shorter than the estimated sizes reported by Goldberg and Novikov (they had no program to produce such proofs). Some of this effect might be due to the fact that zchaffSE proofs are already trimmed to an unsatisfiable core [ZM03a], as far as conflict clauses go. Note that zchaff programs were fixed in two independent ways, compared to the initial 2005 release, as discussed in Section 3. Both fixes produce essentially the same performances. The table shows J07 versions. See Section 9.3 for more details on versions.

Table 3 Proof timing comparisons. Times are CPU seconds. Columns are explained at the beginning of Section 9.2.

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

We conjecture that much of the difference is due to the Goldberg and Novikov *estimates* being based on an on-the-fly PUP method for converting the conflict graph to a resolution derivation (see Section 5). The `BerkMin` code with proof generation is not public, and they do not give details on their estimation method.

Another important difference in our experimental 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% of the full `%RES` proof.

Another compact binary format for SAT–2005, called *Resolution Proof Trace* (`%RPT`), contains exactly four integers 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, in terms of the number of integers needed to represent each proof. However, `%RPT` includes the clashing literal and thereby is able to express a generalization of resolution [VG05a] that cannot be expressed with *resolve-trace*.

The last three columns of Table 2 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 the derivation sizes can vary widely.

9.2 Time Comparisons

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 (called *increment* in the table) to generate a full (%RES) refutation, shown in the second data column, is substantial, exceeding the solve time in a few cases. Checking the proof (third data column) usually requires time comparable to generating it.

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 2.

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 zverifyJ07 times are considerably lower, relative to the solve time, than previously reported by Zhang and Malik. This is logical, because the modifications made for this paper (see Section 4) transferred some of the workload from the verifier to the solver. The alternative fix by the zchaff team at Princeton (see Section 4.3) became available after the experiments, but should give the same overall picture. 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 all the conflict clauses in memory, meaning that it is limited by the amount of virtual memory (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.

Subtracting the sum of data columns 2 and 3 from the last column gives 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 (as described in Section 7.3) 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. That is, in the terminology of Section 7.3, a *unit resolution* refutation for each G_j, for $j = 1, \ldots, 13000$, was generated and verified. 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.

9.3 Verified Unsatisfiable Track of SAT–2007 Competition

An adjunct of the annual conferences on Theory and Applications of Satisfiability Testing is the SAT competition, which usually has several tracks, and varies in format from year to year. For 2007, the *verified unsatisfiable* track had six solvers. This track was more of a proving ground than a competition. The emphasis was on 17 formulas drawn from the Industrial category benchmarks used in the 2007 SAT competition. For this report we selected the 12 benchmarks that were solved within one CPU hour and had proofs of a challenging size. These benchmarks were rerun on a system with CPUs slightly slower than those used in the 2007 event, but with more memory and much more disk space.

Information on the selected Industrial benchmarks is given in Table 4. However, tests on the well-known pigeon-hole series were also conducted (details are reported in the poster `cert-poster-sat07.pdf` and a reformatted version, `cert-tables-sat07.pdf` at the URL in Section 1.4). Results on the Industrial benchmarks are summarized in Table 5.

Most of the solvers were based on the CDCL model with conflict clauses playing a major role. The exception was `tts`, contributed by Ivor Spence, which is designed for small structured formulas [Spe08].

The solver `tts` presents proofs in the compact `%RPT` format. Although `tts` could not solve the huge formulas typical of Industrial benchmarks, it achieved significant success on the pigeon-hole series, coming within a constant factor of the best known refutation size ($(P-1)(P+2)2^{P-3}$ clauses for $P \geq 3$ pigeons). The minimum known refutation sizes for this series are derived analytically. The formula was first written up by Stephen Cook back in 1971, later rederived by Armin Haken about 1985, and again by this author in 2003. Although `tts` is not designed specifically for the pigeon-hole series (and the `tts` author did not know these would be among the benchmarks), this solver produced resolution proofs for formulas up to 19 pigeons, while the other solvers grew at much faster rates than the known optimum and exceeded one CPU hour after 10 to 13 pigeons.

The other solvers used a variety of proof formats. Armin Biere contributed `booleforce`, which presents proofs in his own format, simply called `trace` [Bie08]. It runs in conjunction with a post-processor named `tracecheck` with a special flag to produce the explicit `%RES` format, described in Section 4.3 of the cited paper. He also contributed two versions of `picosat`, one presenting his own `trace` format, to be post-processed into the explicit `%RES` format, and another using the more compact `%RUP` format. The latter format differs significantly in its information content from the others. The `%RUP` format includes only certain derived clauses without giving the steps needed to derive them (see Section 7.1). In the case of `picosat`, the derived clauses were the conflict clauses, which is the expected strategy for CDCL solvers. A simple unix script suffices to convert the `trace` format into the `%RUP` format.

Zhaohui Fu contributed `zchaffSE07`, which had a small, but very important, update to the version of the `zchaffSE` solver from 2005, as described in Section 4.3. It avoids the performance problem discovered in the 2005 version, and performs essentially the same as `zchaff_0`, which fixed the 2005 problem as described in Section 4. Both `zchaff` solvers present proofs in the explicit `%RES` format.

The three solvers based on the CDCL model produced (with the help of a post-processor in some cases) `%RES` proofs that varied in size by factors up to ten for the same benchmark, with different solvers producing the smallest proofs in different cases. The `booleforce` solver was somewhat more successful than the others, so we chose this one to perform a more extensive comparison of proof methodologies on a system with much larger disk files

Table 4 Statistics for Industrial benchmarks, SAT–2007 competition verified unsat track.

(Abbreviated) Benchmark name	2007 fastest sol'n.	num. of variables	num. of clauses	max. clause length	number of clauses of size: 1	2	3	4	5	> 5	num. of literals
AProVE07-15	46	21104	74257	171	1	36142	34380	2745	538	451	192789
AProVE07-20	218	7847	73394	4166	1	62806	5772	389	99	4327	206938
AProVE07-22	81	15589	54263	96	1	26004	25780	2100	198	180	140423
IBM· · ·rb30_Sd.k15	121	29084	119659	12	2072	83935	21996	4194	2369	5093	303691
dated-10-15-u	18	193016	885873	10	16194	700175	133124	4040	16160	16180	2034196
dated-5-11-u	161	108786	482639	10	9230	395289	57400	2300	9200	9220	1096328
dated-5-15-u	864	151952	697321	10	12754	551183	104744	3180	12720	12740	1601192
eq.atree.braun.7	0.80	505	1696	13	3	719	948	10	10	6	4422
eq.atree.braun.8	5	684	2300	15	3	978	1284	16	12	7	5992
eq.atree.braun.9	22	892	3006	17	3	1283	1676	18	16	10	7828
total-5-11-u	22	156980	696581	10	13310	570471	82900	3320	13280	13300	1582352
total-5-13-u	75	178708	806681	10	15072	648853	108896	3760	15040	15060	1842626

than were available for the 2007 experiments. We continued to use the solver versions that were submitted to the 2007 event, although some of them might have newer versions now.

This section reports on an experiment to compare the `%RUP` format and the `trace` format, which offer substantial trade-offs. The `%RUP` format is easy to implement, and even retrofit (the author has retrofitted it to `MiniSat2`, including its preprocessor). This is because it simply writes out derived clauses that the solver will store anyway, one clause per line. However, it puts a serious burden on the post-processor (`rupToRes` for this experiment), to figure out a resolution derivation.

The `trace` format is more involved to implement because, besides writing the derived clause, it appends, on the same line, the indexes of the clauses that supported the derivation, although not necessarily in the order they should be used for the resolution derivation. The solver needs to do careful bookkeeping to identify the set of clauses that was used, However, the post-processor (`tracecheck` for this experiment), only needs to look for a resolution derivation using the specified clauses, and the possibilities are further limited by the requirement that the derivation must fit the TVR framework (see Section 4). A further complication is that in the `trace` proof, all input clauses that are used in the proof must be included, and they may be intermixed with derived clauses. Again, the implementation of `tracecheck` is more involved than the implementation of `rupToRes`, but the program should be much faster. Our primary interest is how the proof sizes compare, because space has been more of a limiting resource than time, in this area.

For the experiment, `booleforce` was run on 12 Industrial category benchmarks selected from the 17 used in the verified unsat track of the SAT 2007 competition. These benchmarks were solved within one CPU hour by at least one solver in the unsat track and had proofs of a challenging size. The proof produced by `booleforce`, in the `trace` format, was converted to the `%RUP` format by a unix script. Then `tracecheck` expanded the `trace` format to a resolution proof and `rupToRes` expanded the `%RUP` format to a resolution proof. Both resolution proofs use the `%RES` format.

Proofs sizes resulting from this experiment are shown in Table 5. The statistics on the benchmarks used appear in Table 4. In Table 5, benchmark size and solution time are shown, to give a sense of anticipated difficulty. The first observation is that `%RUP` proof lengths are an order of magnitude smaller than `trace` proof lengths, fairly consistently. Stated another

Table 5 Proof sizes for Industrial benchmarks, SAT–2007 competition verified unsat track. Solver is `booleforce`. Files sizes in GB; literals and clauses in thousands. CPU is *user time* in mins. **M**: exceeded internal 8 GB memory limit.

(Abbreviated) Benchmark name	lit-erals	`boole-force` CPU	conflict (RUP) clauses	trace proof length	`tracecheck` %RES proof length	RUP proof length	RUP clauses used	`rupToRes` %RES proof length
AProVE07-15	193	4.1	248	0.801	152.805	0.095	219	449.644
AProVE07-20	207	8.5	314	0.807	76.979	0.053	303	121.106
AProVE07-22	140	6.1	224	0.688	102.421	0.075	211	152.252
IBM···rb30_Sd.k15	304	10.1	270	0.767	162.792	0.098	221	283.101
dated-10-15-u	2034	1.2	270	0.044	3.105	0.006	125	196.617
dated-5-11-u	1096	9.6	306	0.434	27.134	0.051	221	1611.357
dated-5-15-u	1601	65.4	853	1.518	79.395	0.138	735	**M**
eq.atree.braun.7	4	0.1	57	0.020	0.606	0.004	56	0.523
eq.atree.braun.8	6	0.4	178	0.075	2.496	0.013	176	1.827
eq.atree.braun.9	8	2.8	545	0.353	18.982	0.059	542	11.014
total-5-11-u	1582	0.9	181	0.057	2.880	0.004	90	204.272
total-5-13-u	1843	2.7	262	0.149	6.807	0.011	145	744.267

way, most of the content in the `trace` proof describes *how* to derive, rather than *what* to derive. The second observation is that the resolution proofs found by `rupToRes` (without the "how to derive" information) are usually much larger than those generated by `tracecheck`. Factors of about 60 are seen for the *dated* family, and factors of 70 and 110 are seen for the *total* family, whereas factors less than one are seen for the much easier *eq.atree.braun* family. This length increase is accompanied by a large increase in CPU time: `tracecheck` generated all 12 proofs in 44 minutes of user CPU time, while `rupToRes` consumed 2735 minutes (almost 2 CPU days), and one proof was not completed, due to needing too much memory. The overall conclusion, also taking into account Table 2, is that `rupToRes` does not discover proofs efficiently for certain benchmark families. The degree of inefficiency seems to be correlated with benchmark size and time of solution, but has no obvious correlation with the size of the proof found by the solver.

To put these results in perspective, `tracecheck` is a sophisticated and optimized program resulting from substantial implementation effort, and is tuned to the proofs that `booleforce` produces. However, `rupToRes` is an unoptimized prototype, not tuned for any particular proof style. The column "RUP clauses used" shows that many of the conflict clauses used by the solver (and hence by `tracecheck`) were *not* used by `rupToRes`. This tendency is most pronounced for the *dated* and *total* families, where the largest ratios of proof sizes were observed. Thus a logical approach to improving the performance of `rupToRes` is to find a way to give preference to earlier derived clauses when searching for a RUP derivation of the current derived clause.

The main purpose of this section is to evaluate and compare space requirements for large proofs using various formats. However, some timing information for the verifier `checker3` is of interest. This verifier requires about 0.08 CPU minutes per GB of proof length on the systems used, when run without substantial contention. Here, CPU minutes consists of the reported *user time*, the most repeatable measure. This figure can double when the system has several competing jobs. We did not have a dedicated system available with the needed disk space. The largest verified proof was 1611 GB and required about 0.10 CPU minutes per GB (see `dated-5-11-u` in Tables 4 and 5). Studying this rate makes sense for `checker3` because it has between two and three literal accesses per proof literal, and as implemented,

each literal access takes constant time. For large proofs the user CPU time is dominated by literal accesses.

In aggregate for the 12 benchmarks studied, in terms of user CPU time, `tracecheck` required about 40% of the `booleforce` solution time and `checker3` required about 60% of the `booleforce` solution time. These timing figures are not repeatable, so details are omitted. They are intended to suggest orders of magnitude only.

The *system time* for `checker3` was usually about half of the user time, but it ranged up to being about equal to the user time in some cases. *System time* consists of time that the CPU was busy inside the operating system, but considered to be serving the user's job.

The *elapsed time* per verified GB varied widely, and depends on many details of the system architecture and hardware, was well as the number of competing jobs. Most values were in the range of 1.5 to 10 minutes per GB, but one benchmark took 23 minutes per GB, while another took 0.1 minutes per GB. There was no apparent relationship between proof-file size and elapsed time per GB among files substantially too large to fit in real memory. Just reading the files required 1.5 minutes elapsed per GB for files between 10 and 100 GB. For the system used in this experiment, with NFS-mounted disks, and a varying number of competing jobs, for estimation purposes, we expect the ratio of elapsed time to user CPU time for `checker3` to be in the range of 20 to 50 for proof files of 100 GB and up. For the largest proof of 1611 GB, the observed ratio was 40. These ratios are far from repeatable, so details are omitted.

10 Conclusion

We have argued that decision procedures can be designed to output proofs that support their decision and are easily checkable by an entirely independent program. We believe that this discipline can assist in debugging the decision procedures as they are developed. In languages that support conditional compilation, the code to output the proof can be encapsulated, so that the program can be compiled to skip this output in the interest of running faster. When an important decision needs to be verified, the program can be rerun with proof-output enabled.

Another implementation consideration is that parts of the proof output might not contribute to the derivation of the empty clause. A post-processing step might be useful to discard unreferenced clauses and renumber the useful clauses to eliminate gaps. This makes it easier for the proof checker to keep the clauses in an array for fast lookup. To keep the overhead of proof-output down, both in time and space, the initial output should be in binary. With multiple output buffers, the delays of disk latency can be largely avoided. We recognize that technical implementation issues like these are not of interest to most researchers, but they are important to make the move from research to production.

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. The primary advantage of the RUP format is its ease of implementation inside a solver. 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 author has retrofitted RUP output to `MiniSat2` and its built-in preprocessor.

We developed a program `checker3` that is able to check `%RES` proofs over 1000 GB in theory, and we actually checked a 1611 GB proof in 166 CPU minutes, although it was 6 days 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 [GN03]. For problems comparable to the benchmarks in their paper, 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. Sinz and Biere recently proposed the `trace` format that combines a relaxed version of *resolve-trace* with RUP. Experiments showed that their format can be converted to a `%RES` proof efficiently in both time and space, when the `trace` proof is produced by a solver written by the same authors. However, other solvers might have their `trace` proofs rejected incorrectly, as discussed in Section 1.2.

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, for proofs arising from some benchmark families in the 2005 and 2007 SAT competitions. However, the prototype `rupToRes` proved to be very inefficient for certain other families, introduced in 2007. The degree of inefficiency seems to be correlated with benchmark size and time of solution, but has no obvious correlation with the size of the proof found by the solver.

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 `tracecheck` or `rupToRes` or another solver) 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.

Acknowledgements We thank Armin Biere, Zhaohui Fu, and Ivor Spence for contributing solvers that output proofs for the SAT 2007 competition. Numerous email discussions with Armin Biere about proof issues were helpful. We thank the anonymous referees for many helpful comments.

References

[AB70] R. Anderson and W. W. Bledsoe. A linear format for resolution with merging and a new technique for establishing completeness. *Journal of the ACM*, 17(3):525–534, 1970.

[ABH+08] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Saïs. A generalized framework for conflict analysis. In *Theory and Applications of Satisfiability Testing – SAT 2008, LNCS 4996*. Springer-Verlag, 2008.

[BB12] M. L. Bonet and S. Buss. An improved separation of regular resolution from pool resolution and clause learning. In *Theory and Applications of Satisfiability Testing – SAT 2012, LNCS 7317*, pages 44–57, Trento, Italy, 2012. Springer.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, Palo Alto, CA, 1996. Springer-Verlag.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *35th Design Automation Conference*, San Francisco, 1998.

[BDS00] Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In *17th International Conference on Computer-Aided Deduction*, 2000.

[BGV01] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1), 2001.

[Bie08] Armin Biere. Picosat essentials. *Journal of Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[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.

[Bur98] S. Burris. *Logic for Mathematics and Computer Science*. Prentice Hall, 1998.

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

[Coo74] S. A. Cook. An observation on time-storage trade-off. *Journal of Computer and Systems Sciences*, 9:308–316, 1974.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *IEEE Symposium on Logic in Computer Science*, Boston, 2001.

[dMR02] L. de Moura and H. Ruess. Lemmas on demand for satisfiability solvers. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 244–251, Cincinnati, OH, 2002.

[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.

[Imm99] N. Immerman. *Descriptive Complexity*. Springer, 1999.

[KBL99] H. Kleine Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[Lov78] D. W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, Amsterdam, 1978.

[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.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53:937–977, 2006.

[RS01] H. Ruess and N. Shankar. Deconstructing shostak. In *IEEE Symposium on Logic in Computer Science*, Boston, 2001.

[SB06] C. Sinz and A. Biere. Extended resolution proofs for conjoining bdds. In *1st Intl. Computer Science Symp. in Russia (CSR 2006), LNCS 3967*, St. Petersburg, 2006. Springer-Verlag. (see also `http://fmv.jku.at/tracecheck`).

[SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Theory and Applications of Satisfiability Testing – SAT 2009, LNCS 5584*, pages 237–243, Swansea, Wales, 2009. Springer.

[Sha08] N. Shankar. Trust and automation in verification tools. In *Automated Technology for Verification and Analysis*, Seoul, 2008.

[Sho84] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[Sip07] M. Sipser. *Introduction to the Theory of Computation*. PWS, 2007.

[Spe08] Ivor Spence. tts: A SAT-solver for small, difficult instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:173–190, 2008.

[VB01] M. N. Velev and R. E. Bryant. EVC: A validity checker for the logic of equality with uninterpreted functions and memories, exploiting positive equality and conservative transformations. In *Computer-Aided Verification (LNCS 2102)*, pages 235–240. Springer-Verlag, July 2001.

[VG02a] 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).

[VG02b] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*, 2002. (Also at `http://www.cse.ucsc.edu/~avg/Papers/sat-pre-post.pdf`.).

[VG02c] Allen Van Gelder. Generalizations of watched literals for backtracking search. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002. (Also at `http://www.cse.ucsc.edu/~avg/Papers/watched-lits.pdf`.).

[VG05a] 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.

[VG05b] Allen Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Annals of Mathematics and Artificial Intelligence*, 43(1–4):239–253, 2005.

[VG07] A. Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In *Theory and Applications of Satisfiability Testing – SAT 2007, LNCS 4501*, pages 328–333, Lisbon, 2007. Springer-Verlag.

[VG08] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Tenth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, 2008. (Also at `http://www.cse.ucsc.edu/~avg/Papers/ proofs-isaim08.pdf`).

[VG09] Allen Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *Theory and Applications of Satisfiability Testing – SAT 2009, LNCS 5584*, pages 141–146, Swansea, Wales, 2009. Springer.

[ZM03a] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. Theory and Applications of Satisfiability Testing*, pages 239–249, Santa Margherita Ligure – Portofino, 2003. (available from authors).

[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.

A Proof Formats

This appendix briefly describes proof formats `%RES`, `%RPT`, and `%RUP` (see also Section 1.4). They all begin with a 256-byte header with identifying information. The original formula, consisting of clauses numbered 1 through m, is not included in the proof. In this formula variables are numbered in the range 1–n and negative integers denote negative literals. The values of n and m appear in ASCII in the proof header.

A.1 `%RES` Format

The proof is a sequence of integers that represents the concatenation of subsequences called *proof operations*, where each proof operation consists of one of the following:

- one binary resolution operation or
- one unary copy operation or
- one unary delete operation or
- one zero-ary output operation.

Only the binary resolution and output operations are described here. The following terminology is used.

An operand is a positive unsigned integer. An operand in the range 1–m refers to the corresponding clause in the input formula. A null operand is 0, i.e., not applicable for the operation.

A clause number is a positive unsigned integer in the range 1–m.

A label is a positive unsigned integer greater than m, or a clause number in the range 1–m. All labels for resolution or copy operations must be greater than m and form a strictly increasing sequence. Normally, the first resolution or copy operation has label $m + 1$ and

each subsequent resolution or copy operation has a label one greater than its predecessor. However, to facilitate proof compaction, nonconsecutive labels are permitted.

A literal is a positive integer in the range 1–n or the negation of such an integer. A null literal is 0, i.e., not applicable for the operation.

A clause is a subsequence consisting of one unsigned integer, say k, followed by k literals, followed by an unsigned integer with value k again. The value of k may be 0, denoting the empty clause. The literals may include duplicates and/or contradictory pairs.

Notice that the clause format differs from that of DIMACS CNF files. The `%RES` format permits a program to jump over any proof operation in either direction, in constant time.

Let L, P_1, and P_2 be labels; let q be a literal; let C_0 be a clause. A *resolution operation* consists of:

$$L \quad q \quad P_1 \quad P_2 \quad C_0 \tag{25}$$

where $P_1 < L$ and $P_2 < L$.

Let C_1 be the clause whose label is P_1 and let C_2 be the clause whose label is P_2. For correctness, C_1 must contain –q and C_2 must contain q, and C_0 must contain the same literals as $(C_1 - [–q]) \cup (C_2 - [q])$ However, the order and number of duplicates may be different. For flexibility, duplicate literals are always treated as a single literal in deciding correctness. In this notation, $(C_1 - [–q])$ means that *all* occurrences of –q are deleted from C_1, etc.

Notice that resolution involving a tautologous clause is acceptable, although it is probably useless. Example: the resolvent of $[a, thinb, thin\neg c]$ with $[c, thin\neg c, thind]$ produces $[a, thinb, thin\neg c, thind]$. The resolvent is subsumed by the first operand, so no logical error occurred.

An *output operation* is a subsequence of four 0s. It denotes that C_0 of the preceding resolution operation is an output clause of the proof. This capability is included for flexibility, so that a proof can identify certain derived clauses as outputs of the proof. The last resolution operation of the file is always considered to produce an output clause, and need not be followed by an output operation. In normal usage, this last resolution operation is the empty clause, and the proof is a refutation.

Example A.1 Say the input file has these clauses (the labels are not in the file):

```
c example 1
p cnf 2 3
1 -2 0
1  2 0
-1 0
```

Thus the input clauses are referenced by 1–3. The proof might be these integers (newlines are for readability):

```
4  2 1 2 2 1 1 2
5 -1 4 3 0 0
0  0 0 0
```

This proof contains two resolution operations and explicitly outputs the last derived clause, which is the empty clause. Note that the first derived clause has two copies of the literal 1. □

A.2 %RPT Format

The proof trace is a sequence of integers that represents the concatenation of groups of four integers each, called *trace operations*, where each trace operation consists of one of the following:

- one binary resolution trace or
- one unary copy trace or
- one unary delete trace or
- one zero-ary output operation.

Only the binary resolution trace and output operations are described here.

The definitions given for **%RES** carry over for n, m, label, and literal.

Let L, P_1, and P_2 be labels; let q be a literal. A *resolution trace operation* consists of:

$$L \quad q \quad P_1 \quad P_2 \tag{26}$$

where $P_1 < L$ and $P_2 < L$. The requirements for correctness are similar to those following (25) for resolution operations, except that the derived clauses among C_0, C_1, and C_2 are implicit in the proof. Note that the clashing literal is explicit, however.

Output operations are identical to those in Section A.1.

For the input file of Example A.1, the trace of the same proof may be

```
4  2 1 2
5 -1 4 3
```

In this case the proof implicitly outputs the last derived clause.

A.3 %RUP Format

The proof is a sequence of integers that represents the concatenation of subsequences called *derived clauses*.

A *derived clause* in the Dimacs-like format D32 is a subsequence consisting of k literals, followed by the unsigned integer 0. The value of k may be 0, denoting the empty clause. The literals may include duplicates, but *not* contradictory pairs.

The short story is that CDCL solvers (those in the vein of grasp, chaff, berkmin, minisat) can simply output their conflict clauses (the final one being the empty clause). The rest of this section goes into details in case there are solvers that use a different approach.

As described in Section 7.1 in more detail, the method of deriving the derived clause is not important. It might be by resolution or some other means. It is only necessary that the soundness of the derived clauses can be verified by a procedure that we call *reverse unit-propagation* (RUP for short). If the claimed derived clause is D, RUP asserts the negation of each literal in D as a separate unit clause. Then RUP tries to derive the empty clause by unit-propagation. If this succeeds, D is known to be sound; otherwise, the proof is rejected.

Example A.2 Say the input file $\mathcal{F}$ has these clauses:

```
c example 1
p cnf 4 4
1 -4 -3 0
1  4 0
-1 0
-4 3 0
```

The proof might be these integers (newlines are for readability):

```
4 3 0
0
```

This proof contains two derived clauses, $[4, 3]$ and []. Negating $[4, 3]$ gives $[\neg 4]$ and $[\neg 3]$, which are added to $\mathcal{F}$. Then unit-clause propagation derives []. Next, add $[4, 3]$ (the first derived clause) to $\mathcal{F}$, and again unit-clause propagation derives []. So the proof is verified as sound, although we have no idea why the program output $[4, 3]$ as a derived clause. □

A.4 Remarks

The purpose of the resolution proof format (`%RES`) is to enable the proof checker to identify a specific operation as an error, if an error occurs. It also provides the most information, to simplify the verification task.

A solver might choose to output the resolution proof trace to save time and space. The resolution proof trace format (`%RRT`) can only determine that an error has occurred somewhere, if an operand of resolution fails to contain the clashing literal, but this might be many steps later than the deviation between the solver and the proof trace checker. In the worst case, at the end of the proof trace, the proof trace checker finds that the final clause is not empty and no error has been detected.

Either a proof or proof trace might be subjected to post-analysis to delete irrelevant operations. Clearly, the fixed format of the proof trace is more convenient for reachability analysis.

The purpose of the `RUP` proof format is to allow a solver to "retro-fit" some verification output without too much trouble, and to allow for some possibility of derivation rules other than resolution.

One drawback is that checking the output is quadratic in its length, by any methods known so far. Unit-clause propagation needs to be repeated for each succeeding derived clause.

Another drawback is that the proof checker can report that a clause is not verified, but cannot necessarily explain exactly why. In this case, some clauses might be left over after unit-clause propagation terminates. If the formula is unsatisfiable, *any* clause is logically implied by it. However, as a debugging tool, this system may well provide helpful information.