Compressing Propositional Refutations

Hasan Amjad [1]

Computer Laboratory
University of Cambridge

Abstract

We report initial results on shortening propositional resolution refutation proofs. This has an application in speeding up deductive reconstruction (in theorem provers) of large propositional refutations, such as those produced by SAT-solvers.

Key words: Proof verification, Propositional refutations

1 Introduction

The emergence of powerful SAT-solvers [6] in recent years has boosted the use of propositional proof in verification [4]. More recently, complete SAT-solvers [13] that can provide proofs of unsatisfiability allow one to settle validity by using a SAT-solver to demonstrate unsatisfiability.

One application of complete SAT-solvers is to use them to efficiently find proofs for large problems, and then, relatively cheaply, verify the proof in a deductive environment, such as a theorem prover. From the theorem proving community point of view, a SAT-solver then becomes a powerful oracle for propositional proof. This application is steadily gaining ground in theorem proving circles [7,12].

The major obstacle to this application is that the size of the proof produced is often prohibitively large to be imported deductively. In this paper, we present some initial results on shortening such proofs prior to deductive reconstruction.

The unsatisfiability proof produced by a SAT-solver is a resolution refuation, created by reading off the implicit implication graph maintained by the solver. This reading off is done on-the-fly, to reduce memory usage overhead. Most current popular SAT-solvers are based on some enhancement of the DPLL [5] algorithm. DPLL performs a backtracking search for satisfying assignments, and throws away unsuccessful branches of the search. Thus, due

[1] Email:ha227@cl.cam.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to the on-the-fly nature of the proof generation by the SAT-solver, the final proof can be expected to contain redundant branches, and, within branches, redundant applications of the resolution rule.

Fortunately, a simple reverse traversal of the resolution graph, i.e., starting with the final empty clause and working back to the initial problem clause, suffices to discover which branches were actually used. The others can be discarded.

Less fortunately, redundant rule occurrences within branches are not so easy to spot, since the redundancy is caused by global aspects of the proof. We show that by analysing the global structure of the proof, shorter proofs can often be obtained.

The next section sets up some basic definitions. We then present the main algorithm in §3, followed by experimental results in §4. We conclude with a brief look at related work and some ideas for future development.

2 Preliminaries

The input problem is presented in conjunctive normal form (CNF), which is an iterated conjunction of *clauses*. A clause is an iterated disjunction of *literals*. A literal is either an atomic proposition (or atom), or a negated atom. Atoms are represented by numbers, and literals by pairing each atom with a boolean polarity indicator. Clauses occurring in the initial problem will be called *initial*.

By associativity and commutativity of conjunction and disjunction, the ordering of clauses and of literals within clauses is unimportant. Thus we may often refer to them as sets rather than as formulas. A clause is considered *subsumed* by any subset of itself.

A clause with n elements is an *n-clause*. 1-clauses are known as *units*.

The resolution rule is

$$\frac{p \vee C \qquad \neg p \vee D}{C \vee D}$$

where p is an atom and C and D are (possibly empty) clauses. The two clauses in the premises are *antecedent* clauses, and the consequent clause of the rule is called a *learnt* clause, or a *resolvent*. The atom p which is *resolved on* is called the *pivot*.

Any literal that occured in both C and D can be safely represented by a single occurance in the resolvent, and is said to be *merged*. The occurrences of non-pivot literals in the antecedents are the *predecessors* of their occurrences in the resolvent. The *ancestor* relation is represented by the transitive closure of the predecessor relation. Successors and descendants are defined analogously.

A resolution proof is a derivation consisting solely of resolution rules, such that all antecedents in the leaves are initial clauses. The proof is a refutation if the empty clause is derived at some point.

Visually, the proof looks like a tree. Since a clause may be used several

times in the proof, each such use is referred to as a *clause occurrence*, represented by pairing the clause with an occurrence count. The literals in the clause occurrence are then called *literal occurrences*, given by pairing each literal with a clause occurrence.

We obtain total orderings on clauses, literals and their occurrences via the usual lexicographic construction.

As initial or root clauses may be used several times, the proof is represented as a DAG, with clauses as nodes, and edges from antecedents to resolvents. Any initial clauses are then sources. Ideally, in a refutation, there would be only one sink, the empty clause. Due to the nature of the proof search however, unused resolvents may also occur as sinks. Such a DAG is called a *resolution graph*.

3 The Algorithm

It is known that there is no efficient deterministic algorithm for shortening resolution proofs using resolution alone. Our method relies on a heuristic reordering of the resolutions in the proof, in the hope of finding a shorter proof, by deriving the empty clause before all the resolutions have been processed.

3.1 Overview

The main algorithm has four phases, the middle two of which may be iterated:

(i) Parse SAT-solver proof into resolution graph.

(ii) Implicitly construct the resolution tree from the graph, tracking *links* between any literals that are the ancestors of literals that were resolved upon. Rank the links using some heuristic.

(iii) Convert links into resolutions, in order of link rank, to produce new resolution graph. Perform subsumption checks after every resolution. Finish when empty clause is derived.

(iv) Verify the shortened proof in theorem prover of choice.

The first phase simply reads in the SAT-solver proof and constructs an implicit representation of the resolution graph, as a list of resolutions.

We now look at each of the remaining phases in detail.

3.2 Finding links

The first step is to develop a representation of the proof that is more amenable to global analysis than a list of resolutions. We borrow from the idea of a connections matrix [3] from connections method literature [2,10].

We wish to keep track of literals in the initial clauses whose descendants were resolved with each other in the original proof. Such literals are said to be *linked*. A link is thus just a pair of literal occurrences. For efficiency,

some meta-level information about the corresponding resolution operation is also recorded with the link, such as the pivot, any merged literals etc. Each link has an associated serial number, corresponding to when its underlying resolution was applied in the original proof.

The graph with literal occurrences as nodes and links as edges is called a *link graph*. This graph G is constructed as follows:

(i) Obtain the set of resolution rule applications from the original proof and sort them topologically in ascending chronological order (i.e., the order in which they were applied in the original proof). This gives a list L of resolutions.

(ii) Initialise the occurrence count $\#C$ of each initial clause C to 0.

(iii) Initialise the ancestor set $Anc(C)$ of each initial clause $C = \{p_0, \ldots, p_n\}$, to

$$\{(p_0, \{(p_0, (C, \#C))\}), \ldots, (p_n, \{(p_n, (C, \#C))\})\} \cup \{(p, \emptyset) | p \notin C\}$$

Note that $(C, \#C)$ is a clause occurrence, and so $(p, (C, \#C))$ is a literal occurrence. Thus ancestor sets contain, for each literal of the clause, a set of ancestor literal occurrences. The sets are singleton for initial clauses, but need not be so for resolvent clauses, because of merging.

(iv) For each $L[i]$, where $0 \leq i < |L|$, let C and D be the antecedents, and R the resolvent. Then,

(a) Initialise the occurrence count $\#R$ to 0.

(b) Compute

$$Anc(R) = \{(p, p_C \cup p_D) | (p, p_C) \in Anc(C) \wedge (p, p_D) \in Anc(D)\}$$

where p is not the pivot.

(c) For the pivot p, compute the set of links $p_C \times p_D$, where $(p, p_C) \in Anc(C)$ and $(p, p_D) \in Anc(D)$, and add them to G.

(d) Increment $\#C$ and $\#D$ and update $Anc(C)$ and $Anc(D)$ with the new values of $\#C$ and $\#D$.

In practice, some other bookkeeping information is also recorded, for efficiency.

EXAMPLE Consider the tautology

$$\neg(x \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y))$$

which is, conveniently for us, already the negation of a CNF formula. There are two atoms, and three clauses. Let the clauses be $C_0 = \{x\}$, $C_1 = \{\neg x, y\}$ and $C_2 = \{\neg x, \neg y\}$.

Figure 1 shows a possible resolution graph (in this simple case, just a tree) for deriving the empty clause, requiring two resolutions. The links in the link graph are then $((C_1, 0), (C_2, 0))_y$, $((C_0, 0), (C_1, 0))_x$ and $((C_0, 0), (C_2, 0))_x$,

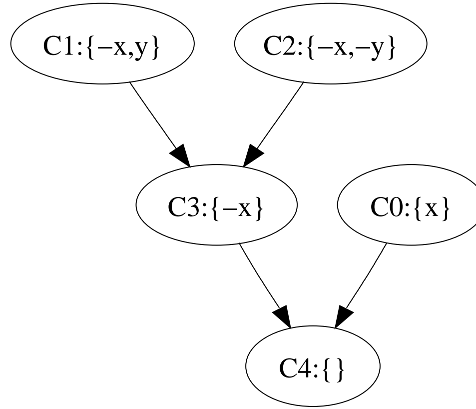

Fig. 1. Resolution graph for the example of §3.2

with the attached pivots moved into the outer subscript to avoid notational clutter. □

All clauses are used once, so clause occurrence counts are all at zero. The example illustrates the main drawback of this approach: it is possible to generate more links than the number of resolutions. In the worst case, the number of links generated can be quadratic in the number of clauses. Note that only initial clauses show up in the links.

By ranking links according to the serial numbering, and applying resolution in order of rank, we end up with the original proof. However, there is much to be gained by reordering the applications, as we shall see in the next section. When the resolution operation corresponding to a link is applied, we say the link has been *used*.

In practice, since resolvents are typically used many times, we compute full ancestor information for only the first occurrence of a resolvent. In terms of the link graph, this means that learnt clause occurrences show up in the links, in a sense abbreviating their own derivation. We thus lose out on being able to fully reorder all resolutions that lead up to the use of the resolvent, but the price of keeping track of ancestor information for all resolvent occurrences (i.e, restricting links to having only initial clauses) is having a prohibitively large link graph.

Despite this complication, it is critical that we view the proof as a tree, because that is what it is as a mathematical object; hence the tracking of individual clause occurrences in the link graph. The DAG representation is useful for implementation purposes, but not for analysing the proof. To see why, we need to introduce the notion of blocked links.

Recall that a link is a pair of literal occurrences. Due to merged literals, it is possible that a single literal occurrence may be multiply linked, i.e., occur in more than one link. A link, one or both of whose literal occurrences is multipl $\#C$ and $\#D$y linked, is called *blocked*. In the example above, the second and third links are blocked since both link to the literal occurrence x in clause occurrence $(C_0, 0)$. If this is the case, we have two choices when we are about

to use such a link:

- Either, we *forbid* using blocked links, in the hope that by removing the remaining links, eventually these will unblock,
- or, we *copy* the remaining links for the resolvent as well.

In practice, forbidding the use of blocked links is not efficient, because then all unblocked links are considered first, whether they are needed for the proof or not. With copying, even though we have the copying create more links, at least we are not forced to use them.

We can now see why a DAG will not do. Suppose we convert the tree to a DAG by collapsing all occurrences of a clause into a single occurrence. Then the corresponding link graph cannot distinguish between clause occurences, and that can possibly create cycles in the graph. If we use the forbid rule for using links, a cycle of blocked links is never used, losing completeness. If we use the copy rule, we can forever be creating new copies of links on the cycle, again losing completeness.

3.3 Using the links

Once we have the link graph, we can rank the links and use them in ranking order. We are guaranteed to find the empty clause. The hope is to find it in fewer resolutions than the original proof. Our algorithm is based on the following rules:

(i) Avoid using links involving learnt clauses, if possible.

(ii) Avoid using blocked links, if possible.

(iii) Prefer newer links to older ones (i.e., prefer higher serial numbers).

(iv) Maximise the use of merged literals.

(v) Perform subsumption checks at every opportunity.

(vi) Track 2-clauses and units.

The heuristic rules (i)-(iv) are incorporated directly into the ranking function for links, with lower numbered rules having higher priority. Rule (v) is applied when processing the resolution corresponding to the link being used, and (vi) is applied after every link use.

It should be noted that the usual heuristics used in proof search need not be very useful in proof compression. Indeed, they often worsen the situation. For instance, one rule common to practically every proof search procedure (ground or first-order) is to try and find smaller clauses. Unfortunately, SAT-solvers make heavy use of resolution with unit clauses. If the unit clause we find was a learnt clause, we then have to optimise lots of subtrees in the proof that use it, resulting in a longer proof. To some extent, rule (iv) attempts to keep a check on clause sizes, without actively looking for smaller ones.

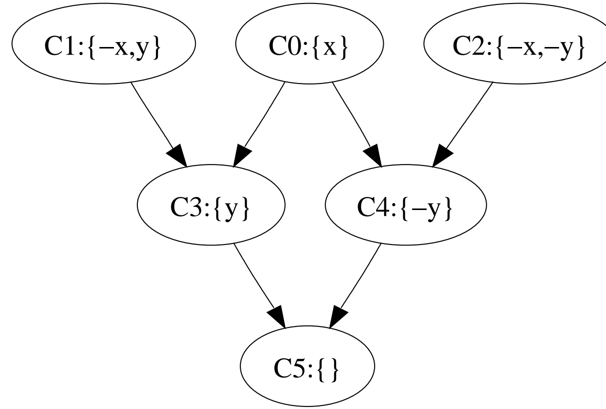

Fig. 2. Resolution graph for the example of §3.3

Indeed, rule (i) is present precisely to discourage using clauses learnt by the SAT-solver (the link graph has already captured that information), since their derivation is unlikely to be the shortest possible one, and, as noted in the previous section, it is even less likely that we can find a shorter one ourselves. Of course, sometimes we have no choice, in which case we fall back on the SAT-solver derivation and hope that rule (v) will help.

The reason for rule (ii) is that we are using the copy rule for using blocked links, rather than forbidding them. The copy rule introduces new links into the link graph, which must be ranked on-the-fly and perhaps used. This is best avoided.

EXAMPLE Figure 2 shows an alternative, inefficient version of the proof of Figure 1, requiring three resolutions. The reader can confirm that this generates precisely the same link graph as for the earlier example (except for the clause occurrence count for C_0). Then, the link ranking rules (i)-(iv) will force the resolution of the link pivoted on y first of all, following which the unit clause tracking in rule (vi) will allow immediate termination with a second resolution. □

Finally, we consider the last two rules in a little more detail.

3.3.1 Subsumption checking

A clause is subsumed by any subset of itself. Subsuming clauses are stronger and take us closer to the empty clause. So subsumption checks can be useful for us in two ways:

(i) When using a link, the would-be resolvent may be subsumed by the resolvent of a previously used link. If so, this link can be discarded without being used, saving us a resolution.

(ii) The antecedents of the link may already be subsumed by existing clauses. In that case, we can use the subsuming clauses as antecedents instead, to derive a stronger resolvent.

In both cases, we can remove other links from the link graph at the same time. These are those links whose literal occurences will no longer appear in the resolvent. Subsumption checks succeed quite often, because, due to the nature of DPLL search, many branches may contain the same sequence of resolution operations.

To check for subsumption, we maintain a cache of clauses we have already derived. This cache is initialised with the initial clauses. It is used to scan for clauses subsuming the antecedents or resolvents of the link under consideration.

A side-effect of adding subsumption checking is that we often end up "resolving" two clauses one or both of which may not contain the pivot. This effect had to be kept in mind when implementing the verification phase.

The cache is implemented as a trie, treating clauses as strings of literals under some total ordering. The lookup for a clause subsuming a clause C is implemented as follows:

(i) If a subsuming unit clause is found, return it immediately

(ii) If a subsuming non-unit clause D is found,
   (a) Compute $D' = C - D$
   (b) Remove the first (wrt to the literal ordering) literal of D to get D''
   (c) Recursively look up D' and D'', choosing the smaller of the returned values, if any

This algorithm is able to look up the smallest subsuming clause in time linearithmic[2] in the size of the target clause, in the average case, and in quadratic time in the worst case. Unfortunately, the worst case contains the situation when a subsuming clause does not exist in the cache, so it happens more often than we would like.

The general problem is known as forward subsumption, and both the SAT and first-order communities have come up with several sophisticated schemes for improving subsumption checking performance. We intend to look at one, zero-suppressed BDDs (ZBDD) [1], as part of future work.

3.3.2 Tracking 2-clauses and units

If we have unit clauses containing the same atom with opposite polarity, we can immediately derive unsatisfiability. So we keep track of unit clauses obtained so far, and, whenever a link use results in a unit clause, check to see if we are done.

Similarly, two 2-clauses which can be resolved such that the remaining literal is a merged literal, allow us to derive a unit. It is fairly straightforward to keep track of 2-clauses as well, with efficient checks for the possibility of deriving units.

Keeping track of 3-clauses is unlikely to gain us anything, since getting

[2] $O(n \log(n))$

anything useful out of a set of 3-clauses is known to be as hard as the general satisfiability problem.

3.3.3 Iteration

We noted earlier that the link finding and link using phases can be iterated. This is because the link using phase results in a new resolution proof, which can be fed back into the link finding phase. Two iterations turn out to be quite useful. We believe the reason may be that the first shortened proof may contain SAT-solver derivations of learnt clauses (that the first iteration was unable to avoid despite rule (i)). The second iteration is possibly able to optimise these derivations.

For the moment, adding more than two iterations seems to provide marginal value at best. We have not yet thought about more useful ways of deciding whether to iterate further or not.

3.4 Verification

Once we have a shorter proof, the proof compression algorithm is finished as such. However, partially because this was our motivation, and partially to sanity check our prototype implementation, we have a final phase in which the proof is deductively verified in a programmable theorem prover.

We have chosen the HOL theorem prover [8] for this task. HOL is an interactive theorem prover for higher-order logic, with a vast collection of theorem libraries and automated proof procedures. The term structure is polymorphic simply typed λ-calculus, and the formula syntax is higher-order predicate calculus with equality. The proof system is classical natural deduction with additional rules for simple type theory.

The main reason for choosing HOL is our familiarity with the tool. However, HOL has two features that make it particularly suitable for the task at hand.

First, HOL is programmable. Rules of inference are programmed in Standard ML and can be composed to form more powerful rules. This allows users to program their own proof procedures (to wit, a refutation verifier).

Second, HOL is fully expansive. This means that inference rules must be built out of a small kernel of eight primitive rules and five axioms, and theorems may only be derived via inference rules. Thus, so long as the kernel is sound, the entire system is sound. In particular, this allows users to program their own proof rules with high assurance that soundness cannot be compromised. This gives us a way of vetting our prototype implementation.

In HOL, inference rules take theorems as inputs and output theorems. This does not fit in with the way we stated our resolution rule, where inputs and outputs are clauses. The cleanest solution is to represent every clause as a theorem. We do not cover the details here, but state simply that a clause

$\{p_0, \dots, p_n\}$ can be represented by the theorem

$$\neg p_0, \dots, \neg p_n, p_0 \vee \dots \vee p_n \vdash \mathbf{ff}$$

where the turnstile symbol separates the premises of the theorem from its conclusion. This representation allows us to represent a resolution operation by a cut. Suppose we have two clauses, represented by

$$v, \neg p_0, \dots, \neg p_n, \neg v \vee p_0 \vee \dots \vee p_n \vdash \mathbf{ff}$$

and

$$\neg v, \neg q_0, \dots, \neg q_m, v \vee q_0 \vee \dots \vee q_m \vdash \mathbf{ff}$$

with pivot v. Then we resolve as follows:

(i) Obtain $\neg p_0, \dots, \neg p_n, \neg v \vee p_0 \vee \dots \vee p_n \vdash \neg v$ by implication introduction on v followed by negation introduction on $v \Rightarrow \mathbf{ff}$

(ii) Cut with the second clause to obtain

$$\neg p_0, \dots, \neg p_n, \neg v \vee p_0 \vee \dots \vee p_n, \neg q_0, \dots, \neg q_m, v \vee q_0 \vee \dots \vee q_m \vdash \mathbf{ff}$$

as desired

Note that the resolvent is carried implicitly in the hypotheses of the theorem. Eventually, we are able to derive a theorem in which all hypotheses except those corresponding to the clause terms have been cut away, showing that the clauses imply false.

The only thing to be careful about in implementing this deductive rule was to make it robust enough to handle situations where the pivot did not occur in one or both antecedents, due to rule (v) of §3.3.

3.5 Soundness and Completeness

Showing soundness is easy since the resulting proof is also a resolution proof, and the resolution rule is known to be sound. Subsumption checking does not affect this, since, even if the pivot is missing from one or both antecedents, the required operation can still easily be cast in terms of a resolution operation on weaker clauses.

Theorem 3.1 *The compression algorithm is sound*

Since the SAT-solver is complete, to check completeness it suffices to check that the algorithm never fails to return a proof of unsatisfiability from the same initial clauses (though it may not necessarily return a shorter proof). This is easily established.

Theorem 3.2 *The compression algorithm is complete*

Proof Since we start with a valid refutation proof, and convert every resolution operation into at least one link, processing all the links is guaranteed to

derive the empty clause at some point. The only danger comes from reordering the links. It is possible that a link that removed a literal from a clause is processed before a link that introduces the same literal, when in the original proof they were ordered the other way around. However, in this case, the danger only occurs if the introduced literal was a merged literal, since then we risk missing out on removing any one of the multiple possible ancestors of that literal. But the link graph construction creates separate links for all ancestor literal occurrences, so this can never happen. □

This proof is a more precise restatement of why we risk losing completeness if we use a DAG as the underlying representation of the proof when constructing the link graph. We note here that a similar argument is not sufficient to establish completeness of proof *search* using a similar approach. That requires a more involved argument [3].

4 Experimental Results

To test the algorithm, we tried it on 50 problems from a standard suite of propositional tautologies distributed with HOL. Each problem was passed to a proof-producing version of the MiniSAT SAT-solver, and the resulting proof was passed through two iterations of the compression algorithm.

The results for a representative subset of the problem set are summarised in Table 1, showing the number of resolution operations in the original and in the compressed proof, sorted in ascending order of the size of the original proof. Note that the original size as reported here does not include redundant branches, which are easy to filter out.

The results are promising. We are able to find redundancies in the proof generated from an extremely efficient proof search engine. It should however be kept in mind that there is no guarantee that we will always get a shorter proof.

The compression ratio is not correlated with the size of the proof. It would be interesting to see if it correlates with other measures of problem hardness such as phase transition thresholds.

However, this assessment has two major omissions. First, we have not tried out the algorithm on large SAT problems (the biggest problem in the table has only 733 clauses). Second, we do not present a timing analysis: it may well be the case that the compression overhead is too expensive.

The reason for both omissions is the same. We only have a prototype implementation at present, with not much thought given to conserving memory or to using the best possible algorithm at every step. This is very much a work in progress. Furthermore, at least the verification phase automatically places an upper bound on the size of problems we can represent, since the fully expansive nature of HOL necessarily carries with it a performance penalty.

Problem	Original	Compressed	Ratio
puz030_1	105	88	0.16
rip02_be	233	218	0.06
u5	352	295	0.16
jnh211	376	255	0.32
hostint1_be	904	784	0.13
ztwaalf1_be	965	840	0.13
dubois20	1016	596	0.41
ssa0432_003	1162	858	0.26
misg_be	1390	937	0.33
ztwaalf2_be	1425	1004	0.29
z4_be	1782	1525	0.14
x1dn_be	2016	1411	0.3
vg2_be	2196	1547	0.3
mjcg_yes	2276	1769	0.22
add1_be	2952	2581	0.13
mul03_be	3235	2974	0.08
rip06_be	3631	2344	0.35
dk17_be	3814	3433	0.1
risc_be	4667	4298	0.08
sqn_be	5733	4645	0.18
msc007_1_008	68987	30936	0.55

Table 1
Number of resolutions in original and compressed proofs

4.1 Complexity

We have not yet undertaken a full complexity analysis of the algorithm, since we have a long way to go before an efficient implementation is in place. Nonetheless, we can venture some remarks.

Resolution is not *automatizable*, i.e., in general it is not possible to efficiently find a shorter resolution proof given a resolution proof. This places limitations on what we can expect from the algorithm above. However, it is

still an open problem whether or not resolution is weakly automatizable, i.e, given a resolution proof, a shorter proof of the same formula can be efficiently found in a stronger proof system. This is one possible area of investigation.

This ties in with the proof complexity of resolution more generally. Eventually, we hope to create a general platform for empirical investigation of our intuitions about the proof complexity of resolution-based proof systems.

5 Conclusion

Proof compression only makes sense if the proof is to be replayed, or verified in a higher assurance but slower system, after having been found. Thus, one area which has seen work on proof compression is proof-carrying code [11]. This is not very relevant to our work however.

Increasingly, high-assurance proof platforms such as HOL exploit the ease of proof verification versus the difficulty of proof search by contracting out proof search to specialised engines, and then rebuilding the proof in the native logic [7,12].

To the best of our knowledge however, compression as a separate stage has not been the focus of research, possibly because, until now, the proofs were simply not large enough to warrant the compression overhead. Here we make a distinction between research on compression, i.e., shortening an already derived proof, and research on better decision procedures that find shorter proofs than earlier procedures [9].

Our work is in a very early stage and many improvements are in the pipeline. Top priorities at the moment are to investigate the isolated impact of the various heuristics, and to improve the implementation to the point where a timing analysis can be carried out. Eventually, we hope to use this tool to investigate automatizability questions in a more general setting.

References

[1] Aloul, F. A., M. N. Mneimneh and K. A. Sakallah, *ZBDD-based backtrack search SAT solver*, in: *International Workshop on Logic Synthesis*, University of Michigan, 2002.

[2] Andrews, P. B., *Refutations by matings*, IEEE Trans. Computers **25** (1976), pp. 801–807.

[3] Bibel, W., *On matrices with connections*, JACM **28** (1981), pp. 633–645.

[4] Biere, A., A. Cimatti, E. M. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems*, LNCS **1579** (1999).

[5] Davis, M., G. Logemann and D. Loveland, *A machine program for theorem proving*, Journal of the ACM **5** (1962).

[6] Eén, N. and N. Sörensson, *An extensible SAT-solver*, in: E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference*, Lecture Notes in Computer Science **2919** (2003), pp. 502–518.

[7] Fontaine, P., J.-Y. Marion, S. Merz, L. P. Nieto and A. F. Tiu, *Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants*, in: H. Hermanns and J. Palsberg, editors, *TACAS*, LNCS **3920** (2006), pp. 167–181.

[8] Gordon, M. J. C. and T. F. Melham, editors, "Introduction to HOL : A theorem-proving environment for higher order logic," Cambridge University Press, 1993.

[9] Jussila, T., C. Sinz and A. Biere, *Extended resolution proofs for symbolic SAT solving with quantification*, in: *9th Intl. Conf. on Theory and Applications of Satisfiability Testing*, LNCS (2006), to appear.

[10] Kowalski, R. A., *A proof procedure using connection graphs*, JACM (1975).

[11] Sarkar, S., B. Pientka and K. Crary, *Small proof witnesses for LF*, in: M. Gabbrielli and G. Gupta, editors, *ICLP*, LNCS **3668** (2005), pp. 387–401.

[12] Weber, T., *Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover*, Technical Report PRG-RR-05-02, Oxford University Computing Laboratory (2005), Theorem Proving in Higher Order Logics - Emerging Trends Proceedings.

[13] Zhang, L. and S. Malik, *Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications*, in: *Design, Automation and Test in Europe Conference and Exposition (DATE)* (2003), pp. 10880–10885.