Industrial-Strength Certified SAT Solving through Verified SAT Proof Checking

Ashish Darbari[1] and Bernd Fischer[2] and Joao Marques-Silva[3]

[1] ARM, Cambridge, CB1 9NJ, England**
[2] School of Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK
[3] School of Computer Science and Informatics
University College Dublin, Belfield, Dublin 4, Ireland

Abstract. Boolean Satisfiability (SAT) solvers are now routinely used in the verification of large industrial problems. However, their application in safety-critical domains such as the railways, avionics, and automotive industries requires some form of assurance for the results, as the solvers can (and sometimes do) have bugs. Unfortunately, the complexity of modern and highly optimized SAT solvers renders impractical the development of direct formal proofs of their correctness. This paper presents an alternative approach where an untrusted, industrial-strength, SAT solver is plugged into a trusted, formally verified, SAT proof checker to provide industrial-strength certified SAT solving. The key characteristics of our approach are (i) that the checker is not tied to a specific SAT solver but certifies any solver respecting the agreed format for satisfiability and unsatisfiability claims, (ii) that the checker is automatically extracted from the formal development, and (iii) that the combined system can be used as a standalone executable program independent of any supporting theorem prover. The core of the system is a checker for unsatisfiability claims that is formally designed and verified in Coq. We present its formal design and outline the correctness criteria. The actual standalone checker is automatically extracted from the the Coq development. An evaluation of the checker on a representative set of industrial benchmarks from the SAT Race Competition shows that, albeit it is slower than uncertified SAT checkers, it is significantly faster than certified checkers implemented on top of an interactive theorem prover.

1 Introduction

Advances in Boolean satisfiability (SAT) technology have made it possible for SAT solvers to be routinely used in the verification of large industrial problems, including problems from safety-critical domains such as the railways, avionics, and automotive industries [11,16]. However, the use of SAT solvers in such domains requires some explicit form of assurance for the results since SAT solvers can and sometimes have bugs. For example, in the SAT 2007 competition, the solver SATzilla CRAFTED reported incorrect outcomes on several problems [20].

Two alternative methods can be used to provide assurance. First, the solver could be proven correct once and for all, but this approach had limited success. For example,

** This author was at the University of Southampton whilst this work was carried out.

Lescuyer et al. [13] formally designed and verified a SAT solver using the Coq proof assistant [3], but without any of the techniques and optimizations used in modern solvers. Smith and Westfold [24] use a correct-by-construction approach to simultaneously derive code and correctness proofs for a family of SAT solvers, but their performance falls again short of the current state of the art. Reasoning about the optimizations makes the formal correctness proofs exceedingly hard. This was shown in the work of Marić [14], who verified at the pseudo-code level the algorithms used in the ARGO-SAT solver but did not verify the actual solver itself. In addition, the formal verification has to be repeated for every new SAT solver (or even a new version of a solver), or else users are locked into using the specific verified solver.

Alternatively, a *proof checker* can be used to validate each individual outcome of the solver independently; this requires the solver to produce a *proof trace* that is viewed as a certificate justifying its outcome. This approach was popularized by the Certified Unsatisfiable Track of the SAT 2007 competition [21] and was used in the design of several SAT solvers such as tts, booleforce, picosat, and zChaff. However, the corresponding proof checkers are typically implemented by the developers of the solvers whose output they check, which can lead to problems in practice. In fact, the checkers booleforce-res, picosat-res, and tts-rpt reported both "proof errors" and "program errors" on some of the benchmarks, although it is unclear what these errors signify.

Confidence can be increased if the checker is proven correct, once and for all. This is substantially simpler than proving the solver correct, because the checker is comparatively small and straightforward, and avoids system lock-in, because the checker can work for all solvers that can produce proof traces in the agreed format. This approach originates in the formal development of a proof checker for zChaff and Minisat proof traces by Weber and Amjad [26], and we follow it here as well. However, we depart considerably from Weber and Amjad in how we design and implement our solution. Their checker replays the derivation encoded in the proof trace *inside* an LCF-style theorem prover such as HOL 4 or Isabelle. Since the design and implementation of these provers relies on using the primitive inference rules of the underlying theorem prover, assurance is very high. However, their checker can run *only* inside the supporting prover, and not as a standalone tool, and performance bottlenecks become prominent when the size of the problems increases. Our checker, SHRUTI,[4] is formally designed and verified using the higher-order logic based proof assistant Coq [3], but we never use Coq for execution; instead we *automatically extract* an OCaml program from the formal development that can be compiled and used independently of Coq. This prevents the user from being locked-in to a specific proof assistant, and allows us to wrap SHRUTI around an industrial-strength but untrusted solver, to provide an industrial-strength certified solver that can be used as a regular component in a SAT-based verification work flow.

Our aim is not a fully formal end-to-end certification, which would in an extreme view need to include correctness proofs for file operations, the compiler, and even the hardware. Instead, we focus on the core of the checker, which is based on the resolu-

[4] SHRUTI in Sanskrit symbolizes 'knowledge' from a spoken word. In our case the outcome of our verified proof checker provides the knowledge about the correctness of a SAT solver and is therefore called SHRUTI.

tion inference rule [18], and formally prove its design correct. We then rely on Coq's program extraction mechanism and some simple glue code as trusted components to build the entire checker. This way we are able to combine a high degree of assurance (much the same way as Amjad and Weber did) with high performance: as we will show in Section 4, SHRUTI is significantly (up to 32 times) faster than Amjad's checker implemented in HOL 4.

2 Propositional Satisfiability

2.1 Satisfiability Solving

Given a propositional formula, the goal of satisfiability solving is to determine whether there is an assignment of the Boolean truth values (i.e., true and false) to the variables in the formula such that the formula evaluates to true. If such an assignment exists, the given formula is said to be *satisfiable* or SAT, otherwise the formula is said to be *unsatisfiable* or UNSAT. Many problems of practical interest in system verification involve proving unsatisfiability, for example bounded model checking [5].

For efficiency purposes, SAT solvers represent the propositional formulas in conjunctive normal form (CNF), where the entire formula is a conjunction of *clauses*. Each clause itself denotes a disjunction of *literals*, which are simply (Boolean) variables or negated variables. An efficient CNF representation uses non-zero integers to represent literals. A positive literal is represented by a positive integer, whilst a negated one is denoted by a negative integer. Zeroes serve as clause delimiters. As an example, the (unsatisfiable) formula $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$ over two propositional variables a and b is thus represented in the widely used DIMACS notation as follows:

```
1 2 0        -1 2 0        1 -2 0        -1 -2 0
```

SAT solvers take a Boolean formula, and produce a SAT/UNSAT claim. A *proof-generating* SAT solver produces additional evidence (also called *certificates*) to support its claims. For a SAT claim, the certificate is simply an assignment, i.e., an enumeration of Boolean variables that need to be set to true in the input problem. It is trivial to check whether that assignment—and thus the original SAT claim—is correct: we simply substitute the Boolean values given by the assignment in the formula and then evaluate the overall formula, checking that it indeed is true. For UNSAT claims, the solvers return a resolution *proof trace* as certificate which is more complicated to check.

2.2 Proof Checking

When a solver claims a given problem is UNSAT and returns a proof trace as certificate, we can independently re-play the trace to check that its claim is correct: if we can follow the resolution inferences given in the trace to derive an empty clause, then we know that the problem is indeed UNSAT, and can conclude that the claim is correct.

A proof trace consists of the subset of the original input clauses used during resolution and the intermediate resolvents obtained by resolving the input clauses. The part of the proof trace that specifies how the input clauses have been resolved in sequence to derive a conflict (i.e., the empty clause) is organized as *chains*. These chains are also

called regular input resolution proofs, or trivial proofs [2,4]. We call the input clauses in a chain its *antecedents* and its final resolvent simply its *resolvent*.

A key correctness constraint for the proof traces (and thus for the proof checker) is that whenever a pair of clauses is used for resolution, *at most* one complementary pair of literals is deleted, i.e., that the chains represent well-formed trivial resolution proofs [4]. Otherwise, we might erroneously "certify" an UNSAT claim for the satisfiable problem $(a \vee \neg b) \wedge (\neg a \vee b)$ by "resolving" over both complementary pairs of literals at once, to derive the empty clause. For efficiency reasons the chains are assumed to be ordered in such a way that we need to resolve only adjacent clauses, and, in particular, that *at least* one pair of complementary literals is deleted in each step. This allows us to avoid searching for the right clauses during checking, and to design a linear-time (in the size of the input clauses) algorithm.

2.3 PicoSAT Proof Representation

Most proof-generating SAT solvers [4,9,29] preserve the two criteria given above. We decided to work with PicoSAT [4] for three reasons. First, PicoSAT is efficient: it ranked as one of the best solvers in the industrial category of the SAT Competitions 2007 and 2009, and in the SAT Race 2008. Second, PicoSAT's proof representation is simple and records only the essential information. For example, it does not contain information about the pivot literals over which it resolves. Third, the representation is in *ASCII* format, which makes it easier to read and process than the more compact binary formats used by other solvers such as Minisat. Together the last two points help us simplify the design of SHRUTI and minimize the size of the trusted components outside the formal development.

A PicoSAT proof trace consists of rows representing the input clauses, followed by rows encoding the proof chains. Each row representing a chain consists of an asterisk (*) as place-holder for the chain's resolvent, followed by the identifiers of the clauses involved in the chain. Each chain row thus contains at least two clause identifiers, and denotes an application of one or more of the resolution inference rule, describing a trivial resolution derivation. Each row also starts with a non-zero positive integer denoting the identifier for that row's (input or resolvent) clause, and ends with a zero as delimiter. For the UNSAT formula shown in the previous section, the corresponding proof trace generated from PicoSAT looks as follows:

```
1   1   2  0      5  *  3  1  0
2  -1   2  0      6  *  4  2  5  0
3   1  -2  0
4  -1  -2  0
```

Rows 1 to 4 denote the input clauses from the original problem that are used in the resolution, with their identifiers referring to the original clause numbering, whereas rows 5 and 6 represent the proof chains. For example, in row 6 first the original clauses 4 and 2 are resolved and then the resulting clause is resolved against the resolvent from the previous chain, in total using two resolution steps.

By default, PicoSAT creates a compacted form of proof traces, where the antecedents for the derived clauses are not ordered properly within the chain. This means that there

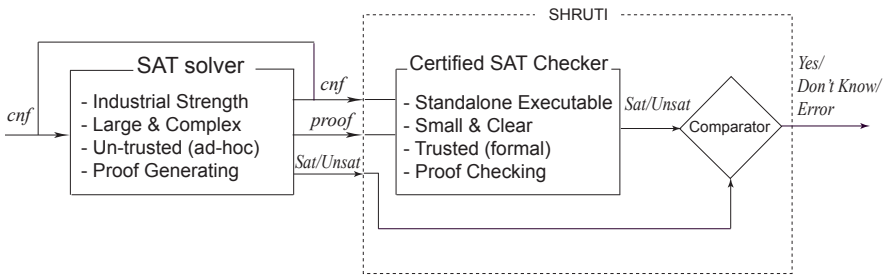

Fig. 1: SHRUTI's high-Level architecture

are instances in the chain where we "resolve" a pair of adjacent clauses but no literal is deleted. In this case we cannot deduce an existence of an empty clause for this trace unless we re-order the antecedents in the chain. However, PicoSAT comes with an uncertified proof checker called Tracecheck that can not only check the outcome of PicoSAT but also corrects this mis-ordering of traces. The outcome of Tracecheck is an extended proof trace and this then becomes the input to SHRUTI, i.e., we consider the combination of PicoSAT and Tracecheck as the solver to be checked. Hence, we can detect errors both in PicoSAT and Tracecheck's re-ordering algorithm, but do not distinguish them.

Similarly, it is possible to integrate other SAT solvers into SHRUTI, even if their proof traces use a different format, by developing a proof translator. This is usually straightforward [25]. As a proof of concept, we developed a translator from zChaff's proof format to PicoSAT's proof format. We again consider the combination of the core solver (i.e., zChaff), post-processor (i.e., the proof translator) and Tracecheck (used for extending the proof trace) as the system to be checked.

3 The SHRUTI System

3.1 High-level Architecture

SHRUTI consists of a formally certified proof checker and a simple comparator that decides whether the solver's claim was correct. It takes as input a CNF file which contains the original problem description and a certificate (i.e., an assignment for a SAT claim or a resolution proof trace for an UNSAT claim). The checker evaluates the certificate and checks whether the two together denote a matching pair of SAT/UNSAT problem and solution. If this is the case, SHRUTI will accept the claim and output "yes", otherwise it will reject the claim and output "don't know". Note that the latter response does not imply that the solver's original claim is wrong—the problem may well be satisfiable or unsatisfiable as claimed. It only indicates that the given evidence (i.e., the assignment or the proof trace) is insufficient to corroborate the claim of the solver (i.e., the assignment does not evaluate to true, or the proof trace is not correct). This can happen due to mis-alignment of chains in the resolution proof as explained in Sect 2.3, or because the proof trace is not well-formed.

The crucial cases are where the problem is satisfiable (resp. unsatisfiable) but the solver claims the opposite, and produces a well-formed certificate for this wrong claim. SHRUTI contains a legitimacy check to prevent it from accepting forged certificates: it outputs "error" if it detects that the certificate is not legitimate, i.e., refers to variables (for a SAT claim) or clauses (for an UNSAT claim) that do not exist in the input problem. Hence, the only possibility under which SHRUTI would certify a wrong claim is if itself contained an error, but (accepting our characterization of the resolution function as correct) this is ruled out by our formal development. A high-level architectural view of our approach is shown in Figure 1.

We designed, formalized, and verified checkers for both satisfiability and unsatisfiability claims. In this paper, we focus on the more interesting aspect of checking unsatisfiability claims; satisfiability claims are significantly easier to check. The core component of the unsatisfiability checker is the development of the binary resolution inference rule inside the Coq proof assistant [3]. We show that the resolvent of a given pair of clauses is logically entailed by the two clauses (see Sect. 3.3), and that our implementation has the properties of the resolution inference rule [18,19] (see Sect. 3.4). In addition, we show that it maintains well-formedness of the clauses (see Sect. 3.5).

Once the formalization and proofs are complete, OCaml code is extracted from the development through the extraction API included in Coq. The extracted OCaml code expects its input in data structures such as tables and lists. These data structures are built by some glue code that also handles file I/O and pre-processes the proof traces (e.g., removes the zeroes used as separators for the clauses). Together with the comparator, the glue code is wrapped around the extracted checker and the result is then compiled to a native machine code executable that can be run independently of Coq.

```
let checkUnsat(cnf, trace) = let (clauses, chains) = split trace in
                             if checkLegal(clauses, cnf)
                             then let res = resolve(clauses, chains) in
                                  if (res = [ ]) then print "yes"
                                  else print "don't know"
                             else print "error"
```

The top-level function $checkUnsat$ first splits the given trace into its constituent input clauses and the proof chains. It then checks whether the clauses used in the resolution proof are legitimate, i.e., whether all clauses used in the resolution proof trace are contained in the original problem CNF, or are derived by applying the resolution inference rule on legitimate clauses. Note that this checks only which clauses are used, not whether the result of an inference is correct. If the certificate is legitimate, then $resolve$, which is a wrapper around the formally verified binary resolution function, is used to derive the empty clause by re-playing the proof steps in the chains.

3.2 Formalization of Resolution in Coq

Coq is based on the Calculus of Inductive Constructions [7,8] and encapsulates the concepts of typed higher-order logic. It uses the notion of proofs as types, and allows constructive proofs and use of dependent types. It has been successfully used in the design and implementation of large scale certification of software such as in the CompCert [12]

project. Our formal development in Coq follows the LCF style [22]; in particular, we only use definitional extensions, i.e., new theorems can only be derived by applying previously derived inference rules. We never use axiomatic extensions, which would allow us to assume the existence of a theorem without a proof, and thus invalidate the correctness guarantees of the extracted code.

In this section we present the formalization of SHRUTI in Coq. Its core logic is formalized as a shallow [1,17] embedding in Coq. In a shallow embedding we identify the object data types (i.e., the types used for SHRUTI) with the types of the meta-language (i.e., the Coq datatypes). Thus, inside Coq, we denote literals by integers, and clauses by lists of integers. Antecedents (denoting the input clauses) in a proof chain are represented by integers and a proof chain itself by a list of integers. We then define our resolution function to work directly on this integer-based representation.

The choice of a shallow embedding and the use of the integer-based representation were conscious design decisions, which make the internal data representation conceptually identical to the external problem representation. Consequently, our parsing functions can remain simple (and efficient), which minimizes the size of the trusted computing base. This is also a difference to Amjad's approach where external C++ functions were used for parsing and translating the integers into Booleans [27].

The main data structures that we used in the Coq formalization are lists and finite maps. The maps are used to represent resolution proofs internally. Their keys are the row identifiers obtained from the proof trace file. Their values are the actual clauses obtained by resolving the clauses specified in the proof trace—note that these are not part of the traces but must be reconstructed. When the trace is read, the identifier corresponding to the first proof chain becomes the starting point for checking. Once the resolvent is calculated for this, the process is repeated for all remaining chain rows, until we reach the end of the trace. If the entry for the last row is the empty clause, we conclude that the given problem and its trace represent an UNSAT instance.

We use the usual notation for quantifiers and logical connectives but distinguish implication over propositions ($\supset$) and over types ($\rightarrow$) for presentation clarity, though they are the same inside Coq. The notation $\Rightarrow$ is used during pattern matching (using $\mathsf{match} - \mathsf{with}$) as in other functional languages. For type annotation we use :, the set of integers is denoted by $\mathbb{Z}$, polymorphic lists by $list$ and list of integers by $list\,\mathbb{Z}$. The empty list is denoted by nil, and for the cons operation we use ::. List membership is represented by $\in$ and its negation by $\notin$. The function abs computes the absolute value of an integer. We use the keyword Definition to present our function definitions implemented in Coq but use let to define a function implemented directly in OCaml.

We define our resolution function ($\bowtie$) with the help of two auxiliary functions $union$ and $auxunion$. Both functions expect the input clauses to respect three well-formedness criteria: there should be no duplicates in the clauses ($NoDup$); there should be no complementary pair of literals *within* any clause ($NoCompPair$), and the clauses should be sorted by absolute value ($Sorted$). The first two assumptions are essentially the constraints imposed on input clauses when the resolution function is applied in practice. Sorting is enforced by us to keep our algorithm efficient. The predicate Wf encapsulates these properties.

Definition $Wf\ c = NoCompPair\,c\ \wedge\ NoDup\,c\ \wedge\ Sorted\,c$

Both $union$ and $auxunion$ use an accumulator to merge two clauses represented as sorted (by absolute value) integer lists, but differ in their behavior for complementary literals. $union$ computes the resolvent by pointwise comparison of the literals. When it encounters a complementary pair of literals it *removes* both the complementary literals and calls $auxunion$ to process the remainder of the lists. When $auxunion$ encounters a complementary pair of literals it simply *copies* both the literals into the accumulator and recurses. Ideally, the proof traces contain only one pair of complementary literals for any pair of clauses that are resolved. However in reality, a solver or its proof trace can have bugs and it can create instances of clauses in the trace with multiple complementary pair of literals in a pair of clauses. Hence, we employ the two auxiliary functions to ensure that the resolution function deals with this in a sound way. Both functions also implement factoring, i.e., if they find the same literal in both clauses, only one copy is kept in the accumulator. Both functions also keep the accumulator sorted, which can be done by simply reversing, since all elements are in descending order.

```
Definition c1 ⋈ c2 = union c1 c2 nil

Definition union (c1 c2 : list ℤ)(acc : list ℤ) = match c1, c2 with
  | nil, c2 ⇒ app (rev acc) c2
  | c1, nil ⇒ app (rev acc) c1
  | x :: xs, y :: ys ⇒ if (x + y = 0) then auxunion xs ys acc
                       else if (abs x < abs y) then union xs (y :: ys)(x :: acc)
                       else if (abs y < abs x) then union (x :: xs) ys (y :: acc)
                       else union xs ys (x :: acc)
end

Definition auxunion (c1 c2 : list ℤ)(acc : list ℤ) = match c1, c2 with
  | nil, c2 ⇒ app (rev acc) c2
  | c1, nil ⇒ app (rev acc) c1
  | x :: xs, y :: ys ⇒ if (abs x < abs y) then auxunion xs (y :: ys) (x :: acc)
                       else if (abs y < abs x) then auxunion (x :: xs) ys (y :: acc)
                       else if x=y then auxunion xs ys (x :: acc)
                       else auxunion xs ys (x :: y :: acc)
end
```

3.3 Logical Characterization of the Resolution Function

The implementation of the checker is based on the operational characterization of the resolution inference rule, and in the next section, we will prove it correct with respect to this. However, we can also use the logical characterization of resolution, and prove the checker sound with respect to this. We need to prove that the resolvent of a given pair of clauses is logically entailed by the two clauses. Thus at an appropriate meta-level (since clauses are lists of non-zero integers, not Booleans), we need to prove a theorem of the following form $\forall c_1\, c_2\, c_3 \cdot (\{c_1, c_2\} \vdash_{\bowtie} c_3) \implies \{c_1, c_2\} \models c_3$.

Here, $\{c_1, c_2\} \vdash_{\bowtie} c_3$ denotes that c_3 is derivable from c_1 and c_2 using the resolution function $\bowtie$, and $\models$ denotes logical entailment. We can use the deduction theorem $\forall a\, b\, c \cdot \{a, b\} \models c \equiv (a \wedge b \implies c)$ and the fact that $\{c_1, c_2\} \vdash_{\bowtie} c_3$ is equivalent

to $c_1 \bowtie c_2 = c_3$ to re-state this as $\forall c_1\, c_2 \cdot (c_1 \wedge c_2) \implies (c_1 \bowtie c_2)$ which we prove its contrapositive form, $\forall c_1 c_2 \cdot \neg(c_1 \bowtie c_2) \implies \neg(c_1 \wedge c_2)$

In order to actually do this proof, we need to lift the interpretation of clauses and CNF from the level of the integer-based representation to the logical level. We thus define two evaluation functions $EvalClause$ and $EvalCNF$ that map an interpretation function I of type $\mathbb{Z} \rightarrow Bool$ over the underlying lists.

Definition $EvalClause\ nil\ I\ =\ False$
$EvalClause\ (x :: xs)\ I\ =\ I\ x\ \vee\ (EvalClause\ xs\ I)$

Definition $EvalCNF\ nil\ I\ =\ True$
$EvalCNF\ (x :: xs)\ I\ =\ (EvalClause\, x\ I)\ \wedge\ (EvalCNF\ xs\ I)$

Definition $Logical\ I = \forall (x : \mathbb{Z}) \cdot I(-x) = \neg(I\ x)$

The interpretation function must be logical in the sense that it maps the negation on the representation level to the negation on the logical level. With this, we can now state the soundness theorem that we proved.

Theorem 1. *(Soundness theorem)*
$\forall c_1 c_2 \cdot \forall I \cdot Logical\ I\ \supset\ \neg(EvalClause\ (c_1 \bowtie c_2)\ I)\ \supset\ \neg(EvalCNF\ [c_1, c_2]\ I)$

Proof. The proof proceeds by structural induction on c_1 and c_2. The first three sub-goals are easily proven by term rewriting and simplification by unfolding the definitions of $\bowtie$, $EvalClause$ and $EvalCNF$. The last sub-goal is proven by doing a case split on if-then-else and then using a combination of induction hypothesis and generating conflict among some of the assumptions. A detailed transcription of the Coq proof is available from http://www.darbari.org/ashish/research/shruti/. □

The soundness proof provides an explicit argument that the resolution function "does the right thing." This is different from Amjad and Weber's approach, who implemented their checker to work on the Bool representation of literals inside HOL and therefore relied on the implicit assurance obtained from using the inference rules of the HOL logic. They provide no explicit proof that their encoding is correct and soundness was never explicitly proven.

3.4 Correctness of the Resolution Function

In this section we prove that our implementation of the resolution function is operationally correct i.e., has the properties expected of the resolution function [18,19]. These properties can also be seen as steps towards a completeness proof, however, this is outside the scope of this paper. These are:

1. A pair of complementary literals is deleted in the resolvent obtained from resolving a given pair of clauses (Theorem 2).
2. All non-complementary pair of literals that are unequal are retained in the resolvent (Theorem 3).
3. For a given pair of clauses, if there are no duplicate literals within each clause, then for a literal that exists in both the clauses of the pair, only one copy of the literal is retained in the resolvent (Theorem 4).

For Theorem 2 to ensure that only a single pair of complementary literals is deleted we need to assume that there is a unique complementary pair ($UniqueCompPair$). The theorem will not hold in this form for the case with multiple complementary pairs.

Theorem 2. *A pair of complementary literals is deleted.*

$$\forall c_1\ c_2 \cdot\ Wf\ c_1\ \supset\ Wf\ c_2\ \supset\ UniqueCompPair\ c_1\ c_2\ \supset \\ \forall \ell_1\ \ell_2 \cdot\ (\ell_1 \in c_1)\ \supset\ (\ell_2 \in c_2)\ \supset\ (\ell_1 + \ell_2 = 0) \supset \\ (\ell_1 \notin (c_1 \bowtie c_2)) \wedge (\ell_2\ \notin (c_1 \bowtie c_2))$$

In the following theorem, $NoCompLit\ \ell\ c$ asserts that the clause c contains no literal that is complementary to the given literal ℓ.

Theorem 3. *All non-complementary, unequal literals are retained.*

$$\forall c_1\ c_2 \cdot\ Wf\ c_1\ \supset\ Wf\ c_2\ \supset \\ \forall \ell_1\ \ell_2 \cdot\ (\ell_1 \in c_1)\ \supset\ (\ell_2 \in c_2) \supset \\ (NoCompLit\ \ell_1\ c_2) \supset (NoCompLit\ \ell_2\ c_1)\ \supset \\ (\ell_1 \neq \ell_2)\ \supset\ (\ell_1 \in (c_1 \bowtie c_2)) \wedge (\ell_2\ \in (c_1 \bowtie c_2))$$

Our last correctness theorem is about factoring. We show that for equal literals in a given pair of clauses only one is copied in the resolvent.

Theorem 4. *Only one copy of equal literals is retained (factoring).*

$$\forall c_1\ c_2 \cdot\ Wf\ c_1\ \supset\ Wf\ c_2\ \supset \\ \forall \ell_1\ \ell_2 \cdot\ (\ell_1 \in c_1)\ \supset\ (\ell_2 \in c_2)\ \supset\ (\ell_1 = \ell_2) \supset \\ ((\ell_1 \in (c_1 \bowtie c_2)) \wedge (count\ \ell_1\ (c_1 \bowtie c_2) = 1))$$

3.5 Preservation of Well-Formedness

Our implementation of the resolution function works correctly if the input clauses are well-formed. This implies that we prove that when we use the resolution function on a pair of well-formed clauses where there is only a single pair of literals to be resolved, we guarantee that the resolvent will be well-formed. This is shown in theorem below.

Theorem 5. *The resolvent of a pair of well-formed clauses is well-formed as well.*

$$\forall c_1\ c_2 \cdot\ Wf\ c_1 \supset\ Wf\ c_2 \supset\ UniqueCompPair\ c_1\ c_2 \supset Wf\ (c_1 \bowtie c_2)$$

Note that we assume the existence of a unique complementary pair of literals between the clauses c_1 and c_2 because the well-formedness only matters when the resolution function is applied on well-formed proof traces (i.e., one complementary pair of literals between any pair of clauses resolved).

3.6 Glue Code and Program Extraction

For the complete checker, we need to wrap a couple of auxiliary functions in Coq around the resolution function. These include $findAndResolve$ which starts the checking process by first obtaining the clause identifiers from the proof trace file, and then invoking $findClause$ to collect all the clauses for each row in the proof part of the proof trace

file. A function called $checkResolution$ recursively calls the function $findAndResolve$ to apply the resolution function $\bowtie$ on each proof chain.

The top-level function in OCaml $checkUnsat$ shown in Sect. 3.1 relies on the function $resolve$. This function (implemented in OCaml) first computes the number of proof steps from the chains (by counting the number of lines with an '*'), and then obtains the chains themselves and stores them in a table. This table is passed as an argument together with the number of proof steps and an empty table (to store resolvents) to the function $checkResolution$ which calculates the resolvents for each step. Once the resolvent is obtained for the last row, its value is queried from the updated resolvent table and the value is returned as the final resolvent. These functions are implemented in OCaml directly because they handle file I/O, a feature not possible to implement inside Coq. An important observation is that the design of these OCaml functions though trivial is still necessary for using the core of the checker which is proven correct inside Coq.

We extract the OCaml code using the built-in extraction API in Coq. By default the extracted code would be implemented in terms of Coq datatypes. But this causes the implementation to be very inefficient at run time. A well-known technique [3] is to replace the Coq datatypes with equivalent OCaml datatypes. This is easily done by providing a mapping (between types) as an option when we do extraction. An important consequence of extraction is that only some datatypes, and data structures get mapped to OCaml's; the key logical functionality is unmodified. The decision for making changes in data types and data structures is a standard procedure used in any large-scale Coq related work such as the CompCert project [12]. For optimization purposes we thus made the following replacements:

1. Coq Booleans by OCaml Booleans.
2. Coq integers ($\mathbb{Z}$) by OCaml `int`.
3. Coq lists by OCaml lists.
4. Coq finite map by OCaml's finite map.
5. The combination of app and rev on lists in the function $union$, and $auxunion$ was replaced by the tail-recursive List.rev_append in OCaml.

Replacing Coq's $\mathbb{Z}$ with OCaml integers gave a performance boost by a factor of 7-10. The largest integer (literal) we can denote depends on the choice of a 32-bit or a 64-bit OCaml `int`. The current mapping is done on a 32-bit signed integer; if SHRUTI encounters an integer greater than ± 2 billion (approx) it aborts with an error message. Making minor adjustments by replacing the Coq finite maps by OCaml ones and using tail recursive functions gave a further 20% improvement.

The Coq formalization consists of eight main function definitions amounting to 114 lines (not counting blank lines and comments), and the proofs of five main theorems shown in the paper and four more that are about maps (not shown here due to space limitations). The entire proof development is organized in several modules and is built interactively using the primitive inference rules of higher-order logic. The extracted code in OCaml is approximately 320 lines and the glue code implemented in OCaml is nearly 200 lines, including comments and print statements. The size of the extracted code is slightly larger than the original development in Coq because the Coq extractor produces code related to the libraries (integer, lists, and finite maps) used in our definitions. However, the actual size of the extracted code is not significant since it has been

Table 1: Comparison of our results with HOL 4 and Tracecheck.

No.	Benchmark	HOL 4			SHRUTI			Tracecheck	
		Resolutions	*Time*	*inf/sec*	*Resolutions*	*Time*	*inf/s*	*Time*	*inf/s*
1.	een-tip-uns-numsv-t5.B	89136	4.61	19335	122816	0.86	142809	0.36	341155
2.	een-pico-prop01-75	205807	5.70	36106	246430	1.67	147562	0.48	513395
3.	een-pico-prop05-50	1804983	58.41	30901	2804173	20.76	135075	8.11	345767
4.	hoons-vbmc-lucky7	3460518	59.65	58013	4359478	35.18	123919	12.95	336639
5.	ibm-2002-26r-k45	1448	24.76	58	1105	0.004	276250	0.04	27625
6.	ibm-2004-26-k25	1020	11.78	86	1132	0.004	283000	0.04	28300
7.	ibm-2004-3_02_1-k95	69454	5.03	13807	114794	0.71	161681	0.35	327982
8.	ibm-2004-6_02_3-k100	111415	7.04	15825	126873	0.90	140970	0.40	317182
9.	ibm-2002-07r-k100	141501	2.82	50177	255159	1.62	157505	0.54	472516
10.	ibm-2004-1_11-k25	534002	13.88	38472	255544	1.77	144375	0.75	340725
11.	ibm-2004-2_14-k45	988995	31.16	31739	701430	5.42	129415	1.85	379151
12.	ibm-2004-2_02_1-k100	1589429	24.17	65760	1009393	7.42	136036	3.02	334236
13.	ibm-2004-3_11-k60	z?	z?	-	13982558	133.05	105092	59.27	235912
14.	manol-pipe-g6bi	82890	2.12	39099	245222	1.59	154227	0.50	490444
15.	manol-pipe-c9nidw_s	700084	26.79	26132	265931	1.81	146923	0.54	492464
16.	manol-pipe-c10id_s	36682	11.23	3266	395897	2.60	152268	0.82	482801
17.	manol-pipe-c10nidw_s	z?	z?	-	458042	3.06	149686	1.21	381701
18.	manol-pipe-g7nidw	325509	8.82	36905	788790	5.40	146072	1.98	398378
19.	manol-pipe-c9	198446	3.15	62998	863749	6.29	137320	2.50	345499
20.	manol-pipe-f6bi	104401	5.07	20591	1058871	7.89	134204	2.97	356522
21.	manol-pipe-c7b_i	806583	13.76	58617	4666001	38.03	122692	15.54	300257
22.	manol-pipe-c7b	824716	14.31	57632	4901713	42.31	115852	18.00	272317
23.	manol-pipe-g10id	775605	23.21	33416	6092862	50.82	119891	21.08	289035
24.	manol-pipe-g10b	2719959	52.90	51416	7827637	64.69	121002	26.85	291532
25.	manol-pipe-f7idw	956072	35.17	27184	7665865	68.14	112501	30.74	249377
26.	manol-pipe-g10bidw	4107275	125.82	32644	14776611	134.92	109521	68.13	216888

produced automatically using the extraction utility in Coq, which we believe to be correct much in the same way as we believe that the OCaml compiler and the underlying hardware are both correct.

4 Experimental Results

We evaluated SHRUTI on a set of industrial benchmarks from the SAT Races of 2006 and 2008 and the SAT Competition of 2007, and compared it to the Amjad and Weber's checkers that run inside the provers [28], and to the uncertified checker Tracecheck. We present our results on a sample of the SAT Race Benchmarks in Table 1. The results for SHRUTI shown in the table are for validating proof traces obtained from the PicoSAT solver. Our experiments were carried out on a server running Red Hat on a dual-core 3 GHz, Intel Xeon CPU with 28GB memory. Times shown for all the three checkers in the table are the total times including time spent on actual resolution checking, file I/O and garbage collection.

The HOL 4 and Isabelle checkers [28] were also evaluated on the SAT Race Benchmarks. The Isabelle-based version reported segmentation faults on most of the prob-

lems [27], but results for the HOL 4 implementation are summarized along with ours in Table 1. The symbol z? denotes that the underlying zChaff solver timed out after an hour. Since we were unable to get the HOL 4 implementation working on our system, it was run on a (comparable) AMD dual-core 3.2 GHz processor running Ubuntu with 4GB of memory. Amjad reported that the version of the checker he has used on these benchmarks is much faster than the one published in [28]. Since Amjad's work is based on proof traces obtained from ZVerify, the uncertified checker for zChaff, the actual proof traces checked by the HOL 4 implementation differ substantially from those checked by SHRUTI. We thus compare the speed in terms of resolution steps (i.e., inferences) checked per second, and observe that SHRUTI is 1.5 to 32 times faster than HOL 4. In addition, as a proof of concept we also validated the proof traces from zChaff by translating them to PicoSAT's trace format. The performance of SHRUTI in terms of inferences per second on the translated proof traces (from zChaff to PicoSAT) was similar to the performance of SHRUTI when it checked PicoSAT's traces obtained directly from the PicoSAT solver—something that is to be expected. We also compare our timings with that obtained from the uncertified checker Tracecheck; here, SHRUTI is about 2.5 times slower, on exactly the same proof traces.

We noticed that OCaml's native code compilation produces efficient binaries but the default settings for automatic garbage collection were not useful, and for large proof traces it ended up consuming (and thereby delaying the overall computation) as much as 60% of the total time. By increasing the initial size of major heap and making the garbage collection less eager, we reduced the computation times of our checker by almost an order of magnitude on proof traces with over one million inferences.

5 Related Work

Recent work on checking the result of SAT solvers can be traced to the work of Zhang and Malik [29] and Goldberg and Novikov [10], with additional insights provided in recent work [2,25]. The work closest to ours is that by Amjad and Weber, which we have already discussed throughout the paper. Bulwahn et al. [6] also have advocated the use of a checker, and experimented with the idea of reflective theorem proving in Isabelle, suggesting that it can be used for designing a SAT checker. However, no performance results were given. Shankar [23] proposed an approach generally based on a verified SAT solver, for checking a variety of checkers.

Marić [14], presented a formalization in Isabelle of SAT solving algorithms that are used in modern day SAT solvers. An important difference is that while we have formalized a SAT checker and *extracted* an executable code from the formalization itself, Marić formalizes a SAT solver (at the abstract level of state machines) and then implements the verified algorithm in the SAT solver *off-line*.

An alternative line of work involves the formal development of SAT solvers. Lescuyer and Conchon [13] have formalized a simplified SAT solver in Coq and extracted an executable. However, they have not formalized several of the key techniques used in modern SAT solvers, and have not reported performance results on any industrial benchmarks. The work of of Smith and Westfold [24] involves the formal synthesis of a SAT solver from a high level description. Albeit ambitious, this work does not include the most effective techniques used in modern SAT solvers.

There has also been interest in the area of certifying SMT solvers. M. Moskal recently provided an efficient certification technique for SMT solvers [15] using term-rewriting systems. The soundness of the proof checker is guaranteed through a formalization using inference rules provided in a term-rewriting formalism.

6 Conclusion

In this paper we presented a methodology for performing efficient yet formally certified SAT solving. The key feature of our approach is that we can combine a formally designed and verified proof checker with industrial-strength SAT solvers such as PicoSAT and zChaff to achieve industrial-strength certified SAT solving. We used the Coq proof-assistant for the formal development, but relied on its program extraction mechanism to obtain an OCaml program which was used as a standalone executable to check the outcome of the solvers. Any proof generating SAT solver that supports the PicoSAT's proof format can be plugged directly into our checker; different formats require only a simple proof translation step.

On the one hand, our checker provides much higher assurance compared to uncertified checkers such as Tracecheck and on the other it enhances usability and performance over certified checkers implemented inside provers such as HOL 4 and Isabelle. In this regard our approach provides an arguably optimal middle ground between the two extremes. We believe that such verified result checkers can be developed for other problem classes as well, and that this is a viable approach to verified software development. We are investigating on optimizing the overall performance of our checker even further so that the slight difference to uncertified checkers can be further minimized. We are also investigating checking SMT proofs.

Acknowledgements. We thank H. Herbelin, Y. Bertot, P. Letouzey, and other people on the Coq mailing list who helped us with Coq questions. J. Harrison gave useful suggestions on the soundness proof. We also thank T. Weber and H. Amjad for answering our questions on their work and also carrying out industrial benchmark evaluation on their checker. This work was partially funded by EPSRC Grant EP/E012973/1, and EU Grants ICT/217069 and IST/033709.

References

1. C. M. Angelo, L. Claesen, and H. De Man. Degrees of formality in shallow embedding hardware description languages in HOL. In *Proc. 6th Intl. Workshop Higher Order Logic Theorem Proving and its Applications*, *LNCS* 780, pp. 89–100. Springer, 1994.
2. P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.
3. Y. Bertot and P. Castéran. Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions, 2004.
4. A. Biere. PicoSAT essentials. *J. Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.
5. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking, in *Advances in Computers*, Academic Press, 2003.
6. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *Proc. 21st Intl. Conf. Theorem Proving in Higher Order Logic*, *LNCS* 5170, pp. 134–149. Springer, 2008.

7. T. Coquand and G. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
8. T. Coquand and C. Paulin. Inductively defined types. In *Proc. Intl. Conf. Computer Logic (COLOG-88)*, *LNCS* 417, pp. 50–66. Springer, 1990.
9. N. Een and N. Sorensson. An extensible sat-solver. In *Proc. 6th Intl. Conf. Theory and Applications of Satisfiability Testing*, *LNCS* 2919, pp. 502–518. Springer, 2003.
10. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proc. Design, Automation and Test in Europe*, pp. 10886–10891. IEEE, 2003.
11. J. Hammarberg and S. Nadjm-Tehrani. Formal verification of fault tolerance in safety-critical reconfigurable modules. *J. Software Tools for Technology Transfer*, 7(3):268–279, 2005.
12. X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and its uses for Verifying Program Transformations. *J. Automated Reasoning*, 41(1):1–31, 2008.
13. S. Lescuyer and S. Conchon. A reflexive formalization of a SAT solver in Coq. In *Proc. Emerging Trends of TPHOLs*, 2008.
14. F. Marić. Formalization and implementation of modern SAT solvers. *J. Automated Reasoning*, 43(1):81–119, 2009.
15. M. Moskal. Rocket-fast proof checking for SMT solvers. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 4963, pp. 486–500. Springer, 2008.
16. M. Penicka. Formal approach to railway applications. In *Formal Methods and Hybrid Real-Time Systems*, *LNCS* 4700, pp. 504–520. Springer, 2007.
17. R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pp. 129–156. North-Holland, 1992.
18. J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.
19. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.
20. SAT 2007 Competition. http://www.cril.univ-artois.fr/SAT07/results/globalbysolver.php?idev=11&det=1.
21. SAT 2007 Competition - Phase 2. http://users.soe.ucsc.edu/~avg/ProofChecker/cert-poster-sat07.pdf.
22. D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1-2):411–440, 1993.
23. N. Shankar. Trust and Automation in Verification Tools. In *Proc. 6th Intl. Symposium Automated Technology for Verification and Analysis*, *LNCS* 5311, pp. 4–17. Springer, 2008.
24. D. R. Smith and S. J. Westfold. Synthesis of propositional satisfiability solvers. Technical report, Kestrel Institute, April 2008.
25. A. Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In *Proc. 10th Intl. Conf. Theory and Applications of Satisfiability Testing*, *LNCS* 4501, pp. 328–333. Springer, 2003.
26. T. Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. *6th Intl. Workshop Implementation of Logics*, Phnom Penh 2006.
27. T. Weber and H. Amjad. Private communication.
28. T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.
29. 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*, pp. 10880–10885. IEEE, 2003.