Verifying SAT and SMT in Coq for a fully automated decision procedure

Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, Benjamin Wener

Verifying SAT and SMT in Coq for a fully automated decision procedure

M. Armand[1], G. Faure[2], B. Grégoire[1], C. Keller[2], L. Théry[1], and B. Werner[2]

[1] INRIA Sophia-Antipolis
{Michael.Armand,Benjamin.Gregoire,Laurent.Thery}@sophia.inria.fr
[2] INRIA Saclay–Île-de-France at École Polytechnique {Germain.Faure,Chantal.Keller,
Benjamin.Werner}@inria.fr

Abstract Enjoying the power of SAT and SMT solvers in the Coq proof assistant without compromising soundness requires more than a yes/no answer from them. SAT and SMT solvers should also return a proof witness that can be checked by an external tool. We propose a fully certified checker for such witnesses written in Coq. It can currently check witnesses from the SAT solvers ZChaff and MiniSat and from the SMT solver VeriT. Experiments highlight the efficiency of this checker. On top of it, new reflexive Coq tactics have been built that can decide a subset of Coq's logic by calling external provers and carefully checking their answers.

1 Introduction

Several developments highlight that *interactive theorem provers* are now ready to be used to formalize highly non-trivial mathematical theories [27,24] and to specify and study industrial scale software [31]. Unfortunately, the high confidence of these formalizations often requires a *tedious interaction* with the prover: contrary to paper proofs, all the "trivial cases" have to be detailed. Thus, the ability to scale up is significantly determined by the capability of automating trivial proofs.

A priori, one could expect that progress in automated theorem proving would directly benefit proof assistants. In particular, the spectacular advances in solving the Boolean SATisfiability Problem [23] have placed SAT solvers at the heart of many practical applications using automated theorem provers [33]. Encouraged by this success, the DPLL procedure [16,15] on which most of the off-the-shelf SAT solvers (e.g. ZChaff [38], MiniSat [19]) are based was largely enhanced and generalized to more expressive logics. This is the case for Satisfiability Modulo Theory solving that considers non-purely propositional extensions of the SAT Problem in which atomic propositions belong to a variety of theories: linear and non linear arithmetic, equality, bit vectors and others. This led to competitive tools like Z3 [35], CVC3 [9], Yices [18], VeriT [13], Simplify [17], Alt-Ergo [14] among others. These tools are used, for instance, in static checkers, verification systems, model checkers and unit test generators. We refer the reader to [8,21,7,39] for some of such applications.

As automatic provers are becoming more and more powerful, their trusted bases have been growing as well. Formally proving a state of the art prover has revealed to be

very difficult [32]. Another approach is then to adopt *result certification*. For this, the provers should not only return a yes/no answer but also produce a *proof witness* that can be checked by an external tool. The main contribution of the paper is to propose a modular and effective checker for SAT and SMT proof witnesses written in Coq and *fully certified*. The benefit of having such a certified checker inside Coq is double. First, *the reliability of automatic provers* can be increased by checking their results *a posteriori*. Second, the automation within Coq *without compromising soundness* can also be enhanced by the introduction of new tactics that delegate goals to external provers whose answers are checked. The source code of the checker and information on its usage can be found online [1].

The work presented in this paper extends the work presented in [5]. The format of certificates has been rethought to be more modular and then makes it easier the addition of new theories. Moreover, we enhanced the automation of Coq by defining new decision procedures.

The paper is organized as follows. In Section 2, proof witnesses for SAT and SMT and the computational reflection on which our checker is based are introduced. Section 3 presents a deep embedding of SAT and SMT terms. Section 4 details the architecture of our checker and shows how the initial kernel that deals with resolution has been extended in order to handle theories. Section 5 explains how the checker has been integrated inside Coq into a tactic. Finally, in Section 6, we evaluate the efficiency of our integration and compare it to other similar attempts.

2 Proof Witnesses and Computational Reflection

2.1 SAT Solvers

SAT solvers deal with propositional formulas given in Conjunctive Normal Form (CNF). They decide whether there is an assignment of the variables that satisfies the formula or not. To set the notations, we recall the basic definitions that are involved. We consider a countable set X of boolean variables. In the following, we use the variables x, y to denote elements of X. Two special constants $\top$ and $\bot$ represent *true* and *false* respectively. V is the set of *positive literals* defined by $X \cup \{\top, \bot\}$. We use the variable v to denote an element of V. The involutive negation of domain V is denoted $v \mapsto \bar{v}$. Its codomain, denoted $\bar{V}$, is called the set of *negative literals*. A *literal* l is either a positive or a negative literal. A clause is a disjunction of literals, denoted $l_1 \vee \cdots \vee l_n$ if it is nonempty and $\square$ otherwise. We use the variables C, D to denote clauses. Clauses are considered up to associativity and commutativity of $\vee$. Furthermore, a variable v and its negation $\bar{v}$ cannot appear in the same clause. A formula in CNF is represented by a *finite set of clauses* $\mathcal{S}$, seen as their conjunction.

Given a valuation ρ mapping a unique boolean to each variable, literals, clauses and sets of clauses are interpreted into booleans as usual. Their interpretation are written $[l]_\rho$, $[C]_\rho$, $[\mathcal{S}]_\rho$ respectively. A set of clauses $\mathcal{S}$ is satisfiable if and only if there exists a valuation ρ such that $[\mathcal{S}]_\rho = \top$. Conversely, $\mathcal{S}$ is unsatisfiable if and only if for all valuation ρ, $[\mathcal{S}]_\rho = \bot$.

We recall that the refutationally complete resolution rule is defined by:

$$\frac{v \vee C \qquad \bar{v} \vee D}{C \vee D}$$

This rule is derivable in classical logic. v is called the *resolution variable*. A tree of resolution rules that have a "comb"-like topology is called a *resolution chain*.

Modern SAT solvers rely on variants of the DPLL algorithm which can be customized to generate a proof witness [37]. The witness is:

- either an assignment of the variables to $\top$ and $\bot$ that satisfies all the clauses, if the set of clauses is satisfiable;
- or a proof by resolution of the empty clause, if the formula is unsatisfiable.

Note that in both cases, the proof witness is not necessarily unique. Here are two examples of proof witnesses.

Example 1. Consider the following set of clauses: $\mathcal{S} = \{x \vee y, x \vee \bar{y} \vee z, \bar{x} \vee z\}$. $\mathcal{S}$ is satisfiable. A proof witness is $\{x \mapsto \top, y \mapsto \bot, z \mapsto \top\}$.

Example 2. Consider the set of clauses $\mathcal{S}' = \{x \vee y, x \vee \bar{y} \vee z, \bar{x} \vee z, \bar{z}\}$. $\mathcal{S}'$ is unsatisfiable. A proof witness is:

$$\dfrac{\dfrac{x \vee y \qquad \dfrac{x \vee \bar{y} \vee z \qquad \bar{z}}{x \vee \bar{y}}}{x} \qquad \dfrac{\bar{x} \vee z \qquad \bar{z}}{\bar{x}}}{\square}$$

From the point of view of result certification, the case of unsatisfiability is the more challenging. Checking such a witness consists in inspecting each application of the resolution rule, making sure that we obtain the empty clause at the end. We explain in Section 4.2 how to preprocess such witnesses in order to obtain smaller certificates.

Even if the two witness producing SAT solvers ZChaff and MiniSat differ on the output format, both give assignments in case of satisfiability and resolution trees in case of unsatisfiability.

2.2 SMT Solvers

SMT solvers decide an extension of the SAT problem in which positive literals are not only boolean variables but also atomic propositions of some first-order theory. Given a signature Σ containing simple types, and function and predicate symbols with their types, a *theory* $\mathcal{T}$ is a set of formulas of type *bool* written using this signature, variables and logical connectives. Those formulas are called *theory lemmas*. In this paper, we only deal with quantifiers free formulas (all non-boolean variables are implicitly existentially quantified) We illustrate this with two examples frequently considered by SMT solvers: congruence closure and linear arithmetic.

Example 3. The theory of congruence closure is defined upon the signature containing:

- one single type, written U_0;
- function symbols of type $U_0 \to \cdots \to U_0$;
- predicate symbols of type $U_0 \to \cdots \to U_0 \to \mathit{Bool}$ among which a particular predicate symbol $=$ of type $U_0 \to U_0 \to \mathit{Bool}$.

and includes all the formulas that are true using the rules of uninterpreted functions with equality, namely the rules of reflexivity, symmetry, transitivity and congruence.

Example 4. The theory of linear arithmetic is defined upon the signature containing:

- one single type, written I;
- four function symbols:
  $0 : I, \qquad S : I \to I, \qquad + : I \to I \to I, \qquad - : I \to I \to I$
- three predicate symbols:
  $= : I \to I \to \mathit{Bool}, \qquad < : I \to I \to \mathit{Bool}, \qquad \leq : I \to I \to \mathit{Bool}$

and includes all the formulas whose standard interpretation in $\mathbb{Z}$ are true.

The set of *atoms* A of a theory $\mathcal{T}$ is the set of all the formulas of type *bool* written using only the signature of $\mathcal{T}$ and variables (but no logical connectives). In the following, a denotes an element of A. The set of *positive literals* V is extended with atoms and becomes $X \cup \{\top, \bot\} \cup A$. The set of *negative literals* $\bar{V}$ is extended as well. The definitions of clause and resolution rule are unchanged.

The interpretation of sets of clauses $[\mathcal{S}]_{T_t,T_v,T_f}$ requires three functions:

- T_t maps simple types to values;
- T_v maps variables to values;
- T_f maps function and predicates symbols of the theory of congruence to concrete functions and predicates according to their types.

A set of clauses $\mathcal{S}$ is satisfiable if and only if there exist functions T_t, T_v and T_f such that $[\mathcal{S}]_{T_t,T_v,T_f} = \top$. Conversely, $\mathcal{S}$ is unsatisfiable if and only if for all functions T_t, T_v and T_f, $[\mathcal{S}]_{T_t,T_v,T_f} = \bot$.

The standard architecture for SMT solvers is an interaction between a SAT solver and decision procedures for the theories [37]. It is well-suited to generate proof witnesses. Indeed, the witness is:

- *if the set of clauses is satisfiable:* either an assignment of the booleans and theory variables
- *if the formula is unsatisfiable:* a resolution tree of the empty clause where leaves are not only clauses from $\mathcal{S}$ but also theory lemmas

Here are two examples of proof witnesses in the theory of congruence closure, given in Example 3.

Example 5. Consider the set of clauses $\mathcal{S} = \{f(x) \neq f(y), y = z, f(x) = f(f(z))\}$ where a is a term of type U. The set $\mathcal{S}$ is satisfiable. A proof witness is $\{x \mapsto f(a), y \mapsto a, z \mapsto a\}$.

Example 6. Consider $\mathcal{S}' = \{f(x) \neq f(y), y = z, f(x) = f(f(z)), x = y\}$. The set $\mathcal{S}'$ is unsatisfiable. A proof witness is

$$\dfrac{\dfrac{x \neq y \vee f(x) = f(y) \qquad x = y}{f(x) = f(y)} \qquad f(x) \neq f(y)}{\square}$$

Note that the leaves are either elements of $\mathcal{S}'$ or congruence closure lemmas. The only congruence closure lemma here is $x \neq y \vee f(x) = f(y)$. Also, the conflict occurs between the clause $x = y$ and the clause $f(x) \neq f(y)$. As the other clauses of $\mathcal{S}'$ do not contribute to the conflict, they do not appear in the proof witness.

To our knowledge, mainly three existing SMT solvers can deliver such informative proof witnesses: Z3, CVC3 and VeriT (some other solvers give proof witnesses, but they are less informative). Even if they differ on the output format, the three of them give resolution trees with theory lemmas in case of unsatisfiability. They also give witnesses for satisfiability. In our experiments we have concentrated on VeriT. In early stages of the development, its open source status gave us the opportunity to have control of the proof traces.

2.3 Computational Reflection in Coq

Coq [10] is a proof assistant based on type theory. It implements the calculus of inductive constructions [10]. In Coq, term equality is done up to conversion. This means that objects may have some computational content. In particular, it is possible to write programs that evaluate within the logic of Coq. The idea of computational reflection, as proposed in [4], is to turn the proof search that is traditionally performed by tactics into an internal computation. To give a more concrete example, let us explain how Coq manages to prove ring equalities automatically. Consider the equality $(x + y) - x = y$ where x and y are two variables in $\mathbb{Z}$. A dedicated data structure in Coq represents ring expressions. It is composed of the operators `mult` for multiplication, `plus` for addition, `minus` for subtraction, and `var` for variables. Variables are indexed by integers. Associated with the data structure, there are two functions that are computable inside the logic. First, the interpretation function $[\ ]_{\rho,\phi}$ that maps the abstract data structure to a concrete domain. The function ρ explains the mapping of the operators, while ϕ handles variables. So we have for example:

$$[\texttt{(minus (add (var 0) (var 1)) (var 0))}]_{to\mathbb{Z},\{0\to x,1\to y\}} \textit{ evaluates to } (x + y) - x$$

where $to\mathbb{Z}$ is the interpretation for the integer numbers. Second, the normalization function `normalize` computes a normal form. So, we have:

$$\texttt{normalize (minus (add (var 0) (var 1)) (var 0))} \quad \textit{evaluates to} \quad \texttt{(var 1)}$$

Its soundness lemma `normalize_sound` states that terms with equal normalizations have equal interpretations:

$$\forall \rho\, \phi\, e_1\, e_2, \texttt{normalize}\ e_1 = \texttt{normalize}\ e_2 \to [e_1]_{\rho,\phi} = [e_2]_{\rho,\phi}$$

With these two functions and the soundness lemma, it is possible to give directly the proof of our initial equality:

```
(normalize_sound  toℤ  {0 → x, 1 → y}
      (minus (add (var 0) (var 1)) (var 0))  (var 1)
      (refl_equal (normalize (var 1))))
```

where `refl_equal` corresponds to the reflexivity of equality. It is then the task of the proof checker of Coq to verify that this proof term is valid.

Our application of computational reflection follows the exact same path. First, we are going to define our own data structures to represent variables, literals, clauses and theories. Second, we are going to encode the certificate checker as a computational function inside the logic of Coq and prove its soundness. Finally, the proof of a Coq statement provable by an external solver will be reduced to a minimum: the application of the soundness of the certificate checker with one of its arguments being the certificate returned by the solver.

Checking certificates requires a fair amount of CPU. The efficiency of the evaluator inside Coq is then crucial for us to be able to handle non-trivial examples. Thanks to an efficient virtual machine [25], the evaluation mechanism in Coq is as fast as Ocaml bytecode evaluation. Also, recent extensions [5] have added native 31-bit machine integers and imperative persistent arrays to the virtual machine. These two features are crucial in order to get a good complexity behavior of our certificate checker.

3 Embedding SAT and SMT terms

To be easily manipulable by our checker, proof witnesses are preprocessed into certificates. Before presenting the general format for certificates, we present the efficient deep embedding of sets of clauses it relies on. Sections 3.1 to 3.4 are dedicated to this representation, and Sections 3.5 and 3.6 interpret them into Coq terms.

3.1 Representing Variables and Literals

On any particular instance, the set of variables X is finite. Let n be its cardinal. We represent each variable by a distinct 31-bit integer ranging from 2 to $n+1$. The integer 0 and 1 are used to represent $\top$ and $\bot$ respectively. The injection of positive and negative literals into the whole set of literals is defined in the following way:

$$\text{if } v \in V \text{ is represented by } i, \text{ then } \begin{cases} \text{the literal } v \text{ is represented by } 2i \\ \text{the literal } \bar{v} \text{ is represented by } 2i+1 \end{cases}$$

Thanks to this encoding, we can benefit from the efficiency of the operations of the native 31-bit integers and most common operations on literals can be implemented directly by bitwise operations.

Because of our encoding, we can represent problems with less than $2^{30}-2$ variables only. This is not a strong limitation since the biggest SAT problems coming from industrial benchmarks consider around 500,000 variables, which is far smaller than our upper bound.

3.2 Representation of Clauses

We have two different representations of clauses : a representation R_1, used for storage, and a representation R_2, used for computation. A clause in R_1 is a list of integers. A clause in R_2 is an array t of length $n+2$ where n is the number of variables. The elements of the array are in the set $\{\oplus, \ominus, \odot\}$. The idea is the following one: for any $v \in V$, if i represents v, $t[i]$ is set to:

$$\begin{cases} \oplus \text{ if } v \text{ appears in the clause} \\ \ominus \text{ if } \bar{v} \text{ appears in the clause} \\ \odot \text{ if neither } v \text{ nor } \bar{v} \text{ appear in the clause} \end{cases}$$

This is well-defined since v and $\bar{v}$ cannot appear at the same time in the clause as explained in Section 2.1. Since clauses are considered up to commutativity and associativity of $\vee$, the representation in R_2 is canonical. Let us consider an example.

Example 7. Suppose $V = \{x, y, z\}$ and x is represented by 2, y by 3, and z by 4. The clause $\top \vee x \vee \bar{z}$ is encoded in R_1 as $[0; 4; 9]$ and in R_2 as $\boxed{\oplus}\boxed{\odot}\boxed{\oplus}\boxed{\odot}\boxed{\ominus}$.

Going from one representation to another is linear in n.

3.3 Representation of Sets of Clauses

To check a proof by resolution, we start from an initial set of clauses. Deduced clauses are then progressively added to this set until the empty clause is reached. For industrial problems, the number of clauses can be quite large. It is then crucial for the basic operations of accessing and adding a clause to be very efficient. For this reason, we use an imperative persistent array to hold clauses. The type of a set of clauses is then an **array of R_2 clauses**.

3.4 Representation of Atoms

To be efficient, we need to represent atoms in such a way that it does not interfere with the checking of the propositional part of a witness: when a resolution is computed, the meaning of the atoms is useless. To do so, variables presented in Section 3.1 are still considered as variables during SAT checking, but represent atoms during theory checking.

Starting from 2, an integer is attributed to each atom appearing in the proof witness. These integers now stand for the propositional variables. As before, they are injected into positive literals.

For checking the theory lemmas, we obviously need to know the atom associated to an integer. To do so, the set of clauses comes with an array mapping the atom represented by i to the cell number i. As we must keep this array as small as possible, common sub-atoms are globally shared using a technique inspired by hash-consing [20]. Example 8 details the representation of $\mathcal{S}$ for Example 6.

Example 8. In Example 6, the array of atoms that is computed is:
$\boxed{f(x)}\boxed{\mathcal{H}(0) = f(y)}\boxed{y = z}\boxed{\mathcal{H}(0) = f(f(z))}$ where $\mathcal{H}(i)$ represents the atom in cell number i. In that context, the clause $f(x) \neq f(y)$ is represented by the integer 3.

In addition to facilitating propositional checking, hashing atoms also speeds up comparison of atoms, since we only need to compare hashes. This is crucial for instance in congruence closure (see Section 4.3).

3.5 Interpretation Functions for Variables, Literals and Clauses

In order to state and prove the soundness of the checker, we interpret sets of clauses in terms of boolean expressions. With our representation, a valuation ρ is a function that maps 31-bit integers into booleans. The definition of interpreted functions $\bullet \mapsto [\bullet]_\rho$ for literals, clauses and sets is straightforward. We interpret a variable v by:

$$[v]_\rho = \top \text{ if } v = 0 \qquad [v]_\rho = \bot \text{ if } v = 1 \qquad [v]_\rho = \rho(v) \text{ if } v \geq 2$$

a literal l by:

$$[l]_\rho = \mathsf{if}\ \mathsf{even}\, l\ \mathsf{then}\ [l/2]_\rho\ \mathsf{else}\ \neg[l/2]_\rho.$$

a clause c by the disjunction of the interpretation of its literals, and a set of clauses C by the conjunction of the interpretation of its clauses.

3.6 Typechecking and Atom Interpretation

As opposed to what is usually done in computational reflection like in *ring* where the interpretation requires just a single type, here we need an interpretation for atoms that accommodates different types (integers, booleans, user-defined types, ...) at the same time. Then, we must add typechecking to our framework, and use it to define a specific interpretation for theory variables (such as x and y in Example 8), uninterpreted functions (such as f in Example 8), and then atoms. We define a syntactic representation of types and the array mapping atoms to integers now comes with three other arrays:

- an array T_t mapping the syntactic representation of a user-defined type to its Coq type;
- an array T_v mapping a theory variable to its interpretation;
- an array T_f mapping an uninterpreted function to its interpretation.

Using T_t, we can interpret syntactic types as:

$$\begin{cases} [\texttt{Bool}]_{T_t} = \texttt{bool} \\ [\texttt{I}]_{T_t} = \texttt{Z} \\ [\texttt{U}_i]_{T_t} = T_t.(i) \end{cases}$$

Using $T := (T_t, T_v, T_f)$, we can now define the interpretation of an atom. The result has the same dependent type as the cells of T_v and T_f. We denote its elements $(s|u)$, where s is a syntactic type and u is of type $[s]_{T_t}$. Here is its definition:

$$\begin{cases} [v]_T = T_v.(v) & \text{if } v \text{ is a variable} \\ [f(t_1,\ldots,t_n)]_T = (s, u\ u_1\ \ldots\ u_n) & \text{if for } i = 1,\ldots,n, [t_i]_T = (s_i|u_i) \text{ and} \\ & T_f.(f) = (s_1 \to \cdots \to s_n \to s|u) \\ [f(t_1,\ldots,t_n)]_T = (\texttt{Bool}, \top) & \text{if this function application is ill-typed} \\ [a = b]_T = (\texttt{Bool}, \texttt{eqb}\ u_a\ u_b) & \text{if } [a]_T = (s|u_a) \text{ and } [b]_T = (s|u_b) \\ [a = b]_T = (\texttt{Bool}, \top) & \text{if this equality is ill-typed} \end{cases}$$

And ρ as used in Section 3.5 can now be defined as:

$$\begin{cases} \rho(v) = u \text{ if } a \text{ is the atom associated to } v, \text{ and } [a]_T = (\texttt{Bool}|u) \\ \rho(v) = \top \text{ otherwise} \end{cases}$$

4 A Modular Coq Checker for SAT and SMT Proof Witnesses

In this section, we present the checkers that have been developed to verify SAT and SMT proof witnesses. Checking satisfiability does not require much work: it simply consists of verifying that the interpretation of the set of clauses using the valuation given by the solver is $\top$.

In that case, the checker is defined by bringing together small checkers dedicated to each aspect of the verification of a resolution tree: checking resolution chains (Section 4.2), checking theory lemmas for theories (Section 4.3 and 4.4) or checking CNF computation (Section 4.5). This gives a lot of freedom in the definition of small checkers, since we only require that they preserve the interpretation of clauses. For instance, any already existing Coq decision procedure can be used as a black-box checker (see Section 4.4).

This modularity comes from the transformation of proof witnesses into certificates, whose format is designed for flexibility.

4.1 The Small Checkers and the interface Checker

A *small checker* is a Coq program that, given a set of clauses $\mathcal{S}$, computes a new set of clauses $\mathcal{S}'$, usually by following the proof witness. We say that a small checker is *sound* if it preserves satisfiability:

$$\text{for any interpretation } T, [\mathcal{S}]_T \Rightarrow [\mathcal{S} \cup \mathcal{S}']_T$$

Notice that, if $\mathcal{S}$ is empty, the soundness means that the small checker only produces tautologies (for instance in the case of theory lemmas).

Given some small checkers, the *interface checker* computes new clauses using the small checkers until it computes the empty clause. If the small checkers are sound, so is the interface checker: since small checkers preserve satisfiability, if the empty clause is obtained, this means that the initial set of clauses was indeed unsatisfiable.

As the interface checker stores the set of clauses in an array, an over-estimate of the length of this array is the number of initial clauses plus the number of clauses that are deduced by small checkers. We can do much better. Once a clause is not used anymore by small checkers, it can be forgotten. So, using techniques similar to register allocation in compilation [3], we compute the maximal number m of clauses that are alive at the same time. We preprocess the proof witness so to allocate a cell to each clause such that two clauses that are never alive at the same time can share the same cell. In practice, this greatly reduces the length of the array.

Example 9 shows how the proof witness of Example 2 is preprocessed into a certificate using cell allocation and our data structures.

Example 9. The proof witness of Example 2 is preprocessed into:

$$\begin{array}{lll} 0 \leftarrow x \vee y & (1) & 3 \leftarrow \bar{x} \vee z \quad (4) \\ 1 \leftarrow x \vee \bar{y} \vee z & (2) & 0 \leftarrow \mathcal{R}(1,2,0) \quad (5) \\ 2 \leftarrow \bar{z} & (3) & 1 \leftarrow \mathcal{R}(2,3,0) \quad (6) \end{array}$$

where $\mathcal{R}(i_1, \ldots, i_p)$ is the result of the resolution chain of the clauses contained in i_1, $\ldots$, i_p (see Section 4.2).

With such a certificate, the interface checker follows the following simple schema.

1. Create an array s of length m, the maximal number of clauses that are alive at the same time.
2. Follow the certificate to put the clauses into s step by step, using small checkers to compute intermediate clauses.
3. Check that the last clause that was added is the empty clause.

The soundness of this checker is proved in Coq:

Theorem 1 (Soundness of the interface checker). *Given sound small checkers, if the interface checker returns $\top$ on entries m and $\mathcal{S}$, then for any interpretation T, $[\mathcal{S}]_T = \bot$.*

4.2 A small Checker to Compute Resolution Chains

Our first example of small checker is a computer of resolution chains. It is the only small checker that is needed to check propositional unsatisfiability, and it is also used for the resolution part of SMT proof witnesses.

Because most SAT solvers use variants of DPLL, the resolution trees that are generated as proof witnesses have very special shapes. It is often the case that one of two premises of the resolution rules is a leaf. So instead of using the binary version of resolution $\mathcal{R}(C_1, C_2)$, each line of the output format for SAT witnesses is a resolution chain $\mathcal{R}(C_1, C_2, \ldots, C_m)$ where

$$\mathcal{R}(C_1, C_2, \ldots, C_m) = \mathcal{R}(\ldots(\mathcal{R}(\mathcal{R}(C_1, C_2), C_3), \ldots), C_m).$$

We can efficiently compute resolution chains using the intermediate representation R_2 of clauses. The algorithm is the direct encoding in Coq of the algorithm of zverify, the checker for ZChaff proof witnesses [38]. The soundness of this small checker has been proved in Coq:

Theorem 2 (Soundness of the algorithm for resolution chains). *Consider m clauses C_1, C_2, and C_m, represented as lists, and let C be the clause, also represented as a list, returned by the algorithm for resolution chains. Then:*

$$\textit{for any interpretation } T,\ [C_1]_T \wedge [C_2]_T \wedge \cdots \wedge [C_m]_T \Rightarrow [C]_T$$

4.3 A small Checker to Compute Congruence Closure

Nelson-Oppen's technique [36] is a solution to combine the congruence closure (described in Example 3 with theories. Congruence closure theory lemmas can be proved in Coq either by using the *congruence* tactic or by checking the detailed proof witness provided by the SMT solver VeriT. We chose to write our checker such that its works up to symmetry (i.e. $x = y$ and $y = x$ are considered to be equal). Thus, our checker for congruence closure uses certificates made of three rules:

- transitivity: $x_1 \neq x_2 \vee \cdots \vee x_{n-1} \neq x_n \vee x_1 = x_n$;
- function congruence: $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$;
- predicate congruence: $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee \neg P(x_1, \ldots, x_n) \vee P(y_1, \ldots, y_n)$.

Its soundness is proved in Coq as well.

4.4 A small Checker for Linear Arithmetic

VeriT does not give certificates for linear arithmetic. However, this is not really a problem since Coq implements a very efficient reflexive and certificate-producing decision procedure for linear arithmetic [11]: `lia`.

We have not yet written a small checker for linear arithmetic. However, we have performed some experiments and checked a wide range of lemmas coming from the SMT-LIB benchmarks using the `lia` decision procedure. The latter has proved to be very efficient: about $1,000$ non-trivial lemmas have been proved in less than 10s. We are then very confident in adding linear arithmetic in our SMT checker, which should be done soon.

4.5 A small Checker for CNF Computation

Contrary to SAT solvers, most SMT solvers deal with arbitrary formulas as input, and not only CNF formulas. In our framework, CNF computation is considered as a particular theory, described in Example 10, for which a small checker can be implemented.

Example 10. The theory of CNF computation is defined upon the signature containing:

- one single type, written F;
- function symbols $\top : F$, $\bot : F$, $\wedge : F \to F \to F$, $\vee : F \to F \to F$, $\Rightarrow : F \to F \to F$, $\Leftrightarrow : F \to F \to F$
- one predicate symbol $P : F \to \mathit{bool}$

and includes all the propositional tautologies, like $\neg P(f_1 \vee f_2) \vee P(f_1) \vee P(f_2)$. P is often omitted.

Since CNF computation does not interfere with other theories—it can be done entirely at the beginning—we chose to encode logical formulas separately from the other kinds of atoms; but this is simply a matter of clarity.

The SMT solvers VeriT and Z3 return proof witnesses which are explicit about CNF computation. We pass them as certificates to the CNF checker, whose soundness is proved in Coq.

5 Coq Tactics that Call SAT and SMT Solvers

In this section, we explain how the automation of Coq can be enhanced by introducing new tactics that delegate goals to external provers. Similar ideas were already used in several contexts, see for example [26,28]. We consider goals of the form $\forall \overrightarrow{x}, F$ where F is a *quantifier-free* decidable formula. The reification step generates a deep embedded clause C given to the external solver, and an interpretation for the signature T_0 such that $[C]_{T_0} = \neg F$.

- If the external solver returns that C is unsatisfiable, the soundness of the witness checker gives $\forall T, [C]_T = \bot$ and then $\neg F = [C]_{T_0} = \bot$ from which we conclude (decidability of F) that F is valid.
- If the external solver returns that C is satisfiable, the proof witness provides a counterexample that can be used to understand why the initial Coq goal is not provable.

We designed a tactic for the solvers ZChaff and VeriT. The corresponding proof-terms are linear in the length of the proof witness (this is an application of the computational reflection in Coq, see Section 2.3).

For the moment, only formulas that directly fit in the fragment decided by SAT and SMT provers can be solved. We leave for future work to define encodings of more elaborate Coq terms into this fragment, in order to be able to prove more goals. This could be done by relying on a previous plugin of Coq called Dp [6].

6 Results and Comparison with Other Work

6.1 Related Work

Another approach for safely integrating SAT and SMT solvers into proof assistants is to formally prove their soundness. This is the approach followed in [32]. It has the advantage to validate the algorithms at work in the prover but is sensitive to any changes, *e.g.* optimizations in the proof search. Unfortunately at the time this paper is written, we could not have access to their code.

Several SAT and SMT solvers have been safely integrated in LCF style interactive theorem provers including CVC Lite in HOL Light [34], haRVey in Isabelle/HOL [22], Z3 in HOL and Isabelle/HOL [12]. In the following, we will focus on the comparison with the integration in Isabelle/HOL and HOL of ZChaff [40] (this corresponds to the state of the art).

6.2 Experiments

All the experiments have been conducted on an Intel Quad Core processor with 2.66GHz and 4Gb RAM, running Linux. For Coq, we use the code available online [1], with the native version of Coq and Ocaml 3.11. For Isabelle/HOL, we use proof reconstructions for ZChaff written by Alwen Tiu and Tjark Weber, with Isabelle 2009-1 and

Solved ZChaff			Isabelle/HOL checker			Coq checker		
#	%	Time	#	%	Time	#	%	Time
75	49.7	64.3	57	37.7	101.	70	46.4	22.3

Table 1: SAT benchmarks

Solved VeriT			Coq checker		
#	%	Time	#	%	Time
3897	97.0	5.075	3871	96.3	1.050

Table 2: Congruence closure benchmarks

Poly/ML 5.2. We use ZChaff 2007.3.12 and the revision 1789 of the VeriT repository. We put a timeout of 300s both for producing and checking the proof witness.

We tested the combination of the interface checker with the small checker of resolution chains for ZChaff on a database of 151 industrial benchmarks from SAT Race'06 and '08. Table 1 presents the number of benchmarks solved by ZChaff, and among them, the number of proof witnesses successfully checked by Coq and Isabelle/HOL. The times are the average of the times for the 55 benchmarks on which ZChaff, Coq and Isabelle/HOL all succeeded, in seconds.

Errors in Isabelle/HOL were due to timeouts, whereas in Coq they are due to memory overflows. It thus clearly appears that Coq is faster but consumes more memory than Isabelle/HOL. Moreover, Coq can solve more cases.

We also tested the combination of the interface checker with the small checkers of resolution chains, CNF computation and congruence closure, for VeriT on a database of 4019 industrial benchmarks from SMT-LIB [2] dealing with congruence closure (QF_UF). Table 2 presents the number of benchmarks solved by VeriT, and among them, the number of proof witnesses successfully checked by Coq. The times are the medium of the times for the 3871 benchmarks on which VeriT and Coq both succeeded, in seconds.

Once more, errors in Coq were due to out of memory errors. Using VeriT, we are able to solve a large majority of the problems in very small amount of time.

Checking a certificate in Coq is faster than producing it. This is really encouraging for adding new decision procedures in Coq: we benefit from the rapidity of the tuned external prover and we do not add much time during proof checking.

7 Conclusion and Future Work

We have presented a checker for SAT and SMT proof witnesses written in Coq that is fully certified. The checker is efficient: it is able to check huge proof witnesses coming from challenging benchmarks in reasonable time, and it compares well with state of the art implementations in Isabelle/HOL. It serves as a basis for a fully automated decision procedures that is completely safe and produces small proof terms. We took a

special care in being generic: for example the same checker is used for MiniSat, ZChaff and VeriT. This relies on the generic format we use after translating proof witnesses into certificates. The technique of *proof by reflection* and native data structures have been crucial in this work. They let us design proofs that are both small and fast. We expect our implementation to be efficient enough to be integrated in a large-scale formalization program, see for example [30].

Our main ambition now is to increase the expressiveness of goals that can be proved following two directions. On the one hand, we want to deal with more theories including non-linear arithmetic, lists and bit vectors. We also want to deal with non quantifier-free formulas. On the other hand, by encoding dependent types and inductives in first-order logic, we could then apply SMT solvers on more complex goals. This idea is already present in [40] where Higher-Order Logic was encoded in propositional logic.

Acknowledgments We wish to thank Sascha Böhme and Tjark Weber for the details and source code they gave us to reproduce the Isabelle experiments. We are also grateful to Pascal Fontaine for his help about the VeriT prover.

References

1. Source code of the development, `http://www.lix.polytechnique.fr/~keller/Recherche/smtcoq.html`
2. SMT-LIB, `http://www.smtlib.org`
3. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Prentice Hall (2006)
4. Allen, S.F., Constable, R.L., Howe, D.J., Aitken, W.E.: The Semantics of Reflected Proof. In: LICS, pp. 95–105. IEEE Computer Society (1990)
5. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to sat verification. In: Kaufmann and Paulson [29], pp. 83–98
6. Ayache, N., Filliâtre, J.C.: Combining the Coq proof assistant with first-order decision procedures (March 2006), unpublished notes
7. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: CAV. LNCS, vol. 6174, pp. 119–122. Springer (2010)
8. Barnett, M., Chang, B.Y.E., DeLine, R., 0002, B.J., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. LNCS, vol. 4111, pp. 364–387. Springer (2005)
9. Barrett, C., Tinelli, C.: Cvc3. In: CAV. LNCS, vol. 4590, pp. 298–302. Springer (2007)
10. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
11. Besson, F.: Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In: TYPES. LNCS, vol. 4502, pp. 48–62. Springer (2006)
12. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann and Paulson [29], pp. 179–194
13. Bouton, T., de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: CADE. LNCS, vol. 5663, pp. 151–156. Springer (2009)
14. Conchon, S., Contejean, E., Kanig, J.: Ergo : a theorem prover for polymorphic first-order logic modulo theories (2006), `http://ergo.lri.fr/papers/ergo.ps`
15. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Communications of the ACM 5(7), 394–397 (1962)

16. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)
17. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
18. Dutertre, B., Moura, L.D.: The Yices SMT Solver. Tech. rep., SRI (2006)
19. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003)
20. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: ML. pp. 12–19. ACM (2006)
21. Filliâtre, J.C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: CAV. LNCS, vol. 4590, pp. 173–177. Springer (2007)
22. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L., Tiu, A.: Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In: TACAS. LNCS, vol. 3920, pp. 167–181. Springer (2006)
23. Franco, J., Martin, J.: A History of Satisfiability. In: Handbook of Satisfiability, pp. 3–74. IOS Press (2009)
24. Gonthier, G.: A computer-checked proof of the four colour theorem. Tech. rep., Microsoft Research Cambridge (2005)
25. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP. pp. 235–246 (2002)
26. Grégoire, B., Théry, L., Werner, B.: A computational approach to pocklington certificates in type theory. In: FLOPS. LNCS, vol. 3945, pp. 97–113. Springer (2006)
27. Harrison, J.: Formal Proof – Theory and Practice. Notices of the American Mathematical Society 55 (2008)
28. Kaliszyk, C., Wiedijk, F.: Certified Computer Algebra on Top of an Interactive Theorem Prover. In: Calculemus/MKM. LNCS, vol. 4573, pp. 94–105. Springer (2007)
29. Kaufmann, M., Paulson, L.C. (eds.): Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings, Lecture Notes in Computer Science, vol. 6172. Springer (2010)
30. Kawaguchi, M., Rondon, P.M., Jhala, R.: Dsolve: Safety verification via liquid types. In: CAV. LNCS, vol. 6174, pp. 123–126. Springer (2010)
31. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
32. Lescuyer, S., Conchon, S.: Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In: FroCos. LNCS, vol. 5749, pp. 287–303. Springer (2009)
33. Marques-Silva, J.: Practical applications of boolean satisfiability. In: Workshop on Discrete Event Systems (WODES'08), Goteborg, Sweden (2008)
34. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. Electr. Notes Theor. Comput. Sci. 144(2), 43–51 (2006)
35. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
36. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transaction on Programming Laguages and Systems 1(2), 245–257 (1979)
37. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). J. ACM 53(6), 937–977 (2006)
38. Princeton University: zChaff, `http://www.princeton.edu/~chaff/zchaff.html`
39. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: TAP. LNCS, vol. 4966, pp. 134–153. Springer (2008)
40. Weber, T.: SAT-based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Institut für Informatik, Technische Universität München, Germany (Apr 2008)