Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants

Pascal Fontaine, Jean-Yves Marion, Stephan Merz,
Leonor Prensa Nieto, and Alwen Tiu

LORIA – INRIA Lorraine – Université de Nancy

Abstract. Formal system development needs expressive specification languages, but also calls for highly automated tools. These two goals are not easy to reconcile, especially if one also aims at high assurances for correctness. In this paper, we describe a combination of Isabelle/HOL with a proof-producing SMT (Satisfiability Modulo Theories) solver that contains a SAT engine and a decision procedure for quantifier-free first-order logic with equality. As a result, a user benefits from the expressiveness of Isabelle/HOL when modeling a system, but obtains much better automation for those fragments of the proofs that fall within the scope of the (automatic) SMT solver. Soundness is not compromised because all proofs are submitted to the trusted kernel of Isabelle for certification. This architecture is straightforward to extend for other interactive proof assistants and proof-producing reasoners.

1 Introduction

Deductive tools for system verification can be classified according to the axes of *expressiveness*, *degree of automation* and *guarantees of soundness*. An ideal tool would score high everywhere: expressive input languages such as higher-order logic or set theory allow a user to write natural and concise models, automatic verification takes care of a large fraction of the proof obligations, and the assurance of soundness gives confidence in the result. In practice, these goals are in conflict. For example, interactive proof assistants encode rich logics, which are at the basis of highly expressive (and user-extensible) modeling languages. Their verification environment is usually built around a small trusted code base, ensuring that theorems can only be produced from explicitly stated axioms and proof rules. At the other end of the spectrum one finds automatic verification tools, including model checkers and decision procedures. These tools come with fixed input languages in which to express the models, and they implement fully automatic verification algorithms tailored for these languages. Using sophisticated optimizations, they aim to scale up to large problems; however, it is all too easy to inadvertently introduce bugs that compromise soundness.

It is clearly desirable to combine interactive and automatic verification tools in order to benefit from their respective strengths. Proof assistants often provide a back door for using automated tools in the form of trusted *oracles*: it suffices to translate the formulas to prove into the input language of the automatic reasoner and to invoke it. If the proof succeeds, the proof assistant will accept the formula as a theorem. However, this mechanism makes the oracle a part of the trusted code base, and therefore weakens the guarantees of soundness. Even if one may be inclined to trust the external reasoner, the translation function can be non-trivial, for example when translating from higher-order to first-order logic; moreover, the translation will often undergo much less testing than the external reasoner itself.

One way to avoid this problem is to make the external reasoner produce proof traces that can be checked independently. Usually, checking a proof is a much simpler problem than finding it in the first place, so the checker can be accepted as part of the trusted code base. Even more, proof checking can be implemented relatively easily within an interactive proof assistant so that the size of the trusted kernel does not augment beyond what users of the proof assistant accept anyway. The combined tool offers the full expressiveness of the proof assistant, but provides the automation of the external reasoner over its domain, without compromising soundness guarantees.

An alternative would be to verify the algorithm of the automatic prover within a proof assistant and to extract an implementation whose soundness is guaranteed, without the need of checking individual proofs. (Note that code extraction or interpretation becomes part of the trusted code base.) It is not clear yet that this approach can produce implementations whose efficiency can compete with reasoners implemented as, say, highly optimized C programs. Mahboubi [14] describes ongoing work with the aim of implementing cylindrical algebraic decomposition in Coq.

In this paper we describe an implementation of proof certification for a decision procedure for the quantifier-free first-order language of uninterpreted function and predicate symbols implemented in haRVey [7] within Isabelle/HOL [23], the encoding of higher-order logic in Isabelle. The SMT (Satisfiability Modulo Theories) solver haRVey combines a SAT solver with decision procedures. In a nutshell, the SAT solver maintains a Boolean abstraction of the input formula. Whenever a propositional model for this abstraction is found, it is submitted to the decision procedure(s). If the model is found to be incompatible with a theory, a *conflict clause* is produced in order to exclude a class of models. This process continues until either a model is found, in which case the input formula is satisfiable, or until the SAT solver determines the Boolean abstraction to be unsatisfiable. Because the SAT solver plays a central role in haRVey, we first introduce in Sect. 3 proof reconstruction in Isabelle for SAT solvers. In Sect. 4 we describe how haRVey has been extended to produce proof traces and how we implement proof reconstruction for these traces (Sect. 5). The overall approach generalizes to other theories implemented in SMT solvers, including fragments of arithmetic and set-theoretical constructions.

Related work. We are not aware of any existing combination of SMT solvers and proof assistants, but the use of proof certification for tool combination is widely accepted. For example, an interface between Coq and the rewriting system ELAN [21] lets ELAN compute proof objects (as λ-terms) that are submitted to Coq, and a similar approach has been implemented for Coq and the first-order theorem provers Bliksem [4] and Zenon. Because explicit proof objects can be huge, Necula and Lee [18] propose techniques to compress them. In contrast, we do not compute full proof objects but just “hints” that guide Isabelle during proof reconstruction. Meng et al. describe a combination of Isabelle and resolution-based first-order theorem provers [16], and a similar approach underlies the combination of Gandalf and HOL within the Prosper project [13]. The work on the TRAMP system reported by Meier [15] is used in the Omega system [25], and it appears to be closely related to ours because the target logic is similar; also, our "proof hints" can be understood as a form of proof planning.

2 Motivation for tool integration

Our motivation for combining interactive proof assistants and SMT solvers comes from case studies that we performed for the verification of distributed algorithms, including a framework for clock synchronization protocols [3, 26]. These case studies were carried out in Isabelle/HOL, and this formalism allowed us to write easily understandable system specifications. When it came to verification, we would typically instantiate the higher-order abstractions in a few initial proof steps, leaving us with first-order verification conditions. Many of these subgoals would fall within the domain of automatic decision procedures. A typical example is provided by the following lemma that appears within the context of clock synchronization:

lemma *bounded-drift*:
  assumes $s \le t$ **and** *correct* $p\ t$ **and** *correct* $q\ t$
    and *rbound1* C **and** *rbound2* C **and** *rbound1* D **and** *rbound2* D
  shows $|C\ p\ t - D\ q\ t| \le |C\ p\ s - D\ q\ s| + 2 \times \rho \times (t-s)$

The lemma establishes a bound on the drift between two ρ-bounded clocks C and D for processors p and q that are supposed non-faulty (*correct*) at time t. It relies on the following definition of ρ-boundedness:

$$\begin{aligned} \mathit{rbound1}\ C &\overset{\Delta}{=} \forall p,s,t.\ \mathit{correct}\ p\ t \wedge s \le t \longrightarrow C\ p\ t - C\ p\ s \le (1+\rho)\times(t-s) \\ \mathit{rbound2}\ C &\overset{\Delta}{=} \forall p,s,t.\ \mathit{correct}\ p\ t \wedge s \le t \longrightarrow (1-\rho)\times(t-s) \le C\ p\ t - C\ p\ s \end{aligned}$$

The Isabelle proof of this lemma in [26] requires a series of intermediate lemmas, which were rather tedious to prove. In particular, Isabelle's built-in tactic for linear arithmetic is unable to prove the lemma, even after manual instantiation of the quantifiers. This is mainly due to the appearance of the subterm $\rho \times (t-s)$, which falls outside the scope of linear arithmetic. In contrast, it is not hard to see that the lemma is correct, and CVC-Lite [2] was able to prove it automatically. CVC-Lite is an SMT solver whose core consists of a combination of decision procedures for fragments of first-order logic; other tools in this category include MathSAT [5], ICS [10] and Yices.

As a first step towards tool combination, we tried an oracle-style integration and implemented ML functions that translate a fragment of Isabelle/HOL to the input languages of SMT solvers. The recent emergence of the common SMT-LIB input format [24] turned out to be very helpful, because the same translations worked for many different tools. By using SMT solvers as oracles, we could concentrate on the high-level structure of the verification and leave tedious details such as the above lemma to the external tools.

However, we were also quickly reminded of the dangers with oracle-style integration: a simple typo in the translation functions was enough to corrupt soundness. The translation from a higher-order setting to a (multi-sorted) first-order language is non-trivial. In short, it was all too easy to introduce bugs in the translation, which suggested to us that we should investigate techniques of proof certification.

3 Proof reconstruction for propositional logic

SAT solvers decide the satisfiability problem for propositional logic, and they are an essential component of SMT solvers. Given a propositional formula, a SAT solver either computes a satisfying valuation or reports that the formula is unsatisfiable. Modern SAT solvers implement the DPLL algorithm [6] due to Davis, Putnam, Logemann, and Loveland, enhanced by optimizations such as conflict analysis and non-chronological backtracking, good branching heuristics, and efficient data structures [29]. These solvers expect the input to be presented as a set (i.e., conjunction) of clauses, which are disjunctions of literals. In preparation for using a SAT solver, we must convert arbitrary propositional formulas into conjunctions of clauses, preserving satisfiability.

A naive conversion to conjunctive normal form (CNF) simply distributes disjunctions over conjunctions. However, this could result in a conjunction whose size is exponential in the size of the original formula. For example, the formula

$$(a_1 \wedge b_1) \vee \ldots \vee (a_n \wedge b_n)$$

gives rise to 2^n conjuncts. For our purposes, we do not need to produce an equivalent CNF formula, but only have to preserve (un)satisfiability, and it is well known that a conversion of linear complexity is possible in this case. The classical technique, due to Tseitin [1, 27], is to introduce new Boolean variables to represent complex subformulas. In the above example, we would introduce additional variables $x_1, \ldots, x_n$ and obtain the clauses

$$x_1 \vee \ldots \vee x_n,\ \neg x_i \vee a_i,\ \neg x_i \vee b_i,\ x_i \vee \neg a_i \vee \neg b_i \quad (i = 1, \ldots, n).$$

The first clause represents the original formula, whereas the remaining clauses arise from “definitional” equivalences $x_i = a_i \wedge b_i$. This idea can be implemented in Isabelle by a tactic that repeatedly applies the theorem

$$(A \wedge B) \vee C \ = \ (\exists x. (x = A \wedge B) \wedge (x \vee C))$$

in order to obtain a quantified Boolean formula $\exists \mathbf{x}. c_1 \wedge \ldots \wedge c_m$ that is equivalent to the original formula. The clauses c_1, …, c_m are then passed on to the SAT solver.

SAT solvers try to compute a satisfying assignment of truth values to atoms by repeatedly applying two basic operations [17]: Boolean constraint propagation determines the values of Boolean variables that appear in *unit clauses*, i.e. clauses that contain a single unassigned literal. Second, truth values are guessed for variables whose value has not yet been determined. In case these guesses are found to be incompatible with the input clauses, the search backtracks, remembering the unsuccessful guesses as a *learned clause* that is added to the original set of clauses in order to help direct the search.

Problem	# clauses	SAT time	Total time
MSC007-1.008	204	0.208	11.546
PUZ015-2.006	184	0.005	2.435
PUZ016-2.005	117	0.003	1.158
PUZ030-2	63	0.002	0.485
PUZ033-1	13	0.003	0.078
SYN090-1.008	65	0.002	0.492
SYN093-1.002	26	0.005	0.133
SYN094-1.005	82	0.005	0.742

Table 1. Running time for SAT proof reconstruction.

In a theorem-proving context, we show a formula to be valid by establishing the unsatisfiability of its negation, and we are therefore mostly interested in verdicts of unsatisfiability. As explained in [30], SAT solvers such as MiniSAT [9] or zChaff [29] can produce justifications of unsatisfiability verdicts as lists of binary resolution steps. Each step operates on two clauses $c_1 \equiv a_1 \vee \ldots \vee a_k$ and $c_2 \equiv b_1 \vee \ldots \vee b_l$ that contain a complementary literal (say, $b_1 \equiv \overline{a_1}$) to produce the clause $a_2 \vee \ldots \vee a_k \vee b_2 \vee \ldots \vee b_l$; hence, a step can be represented as a triple of integers identifying the two participating clauses and the propositional variable to resolve on. The proof ends with establishing the empty clause, which is trivially unsatisfiable.

The proof trace produced by the SAT solver is passed to Isabelle, where it is used to guide a proof of the unsatisfiability of the formula $\exists \mathbf{x}. c_1 \wedge \ldots \wedge c_m$ obtained by the CNF transformation. The unsatisfiability of this latter formula is easily reduced to the proof of the sequent $[\![c_1; \ldots; c_m]\!] \Longrightarrow \mathit{False}$, i.e. to deriving a contradiction from the hypotheses $c_1, \ldots, c_m$. At this point, the representation of the clauses c_i in Isabelle becomes important. A naive representation of clauses as disjunctions of literals in Isabelle/HOL requires associativity and commutativity of disjunction to be applied prior to each resolution step so that the complementary literal appears, say, as the first disjunct. This complication can be circumvented when clauses are encoded as sequents, observing that the clause $a_1 \vee \ldots \vee a_k$ can be represented as the sequent $[\![\overline{a_1}; \ldots; \overline{a_k}]\!] \Longrightarrow \mathit{False}$ where $\overline{a_i}$ denotes the complement of the literal a_i. With this representation, binary resolution essentially becomes an application of the cut rule. More precisely, given two clauses $c_1 \equiv [\![a_1; \ldots; a_k]\!] \Longrightarrow \mathit{False}$ and $c_2 \equiv [\![b_1; \ldots; b_l]\!] \Longrightarrow \mathit{False}$ in sequent representation such that, say, $b_j \equiv \overline{a_i}$, we deduce from c_1 the equivalent sequent

$$c_1' \equiv [\![a_1; \ldots; a_{i-1}; a_{i+1}; \ldots; a_k]\!] \Longrightarrow \overline{a_i}$$

and then join the two sequents using a primitive operation provided by Isabelle to obtain the sequent representation of the resolvent, i.e.

$$[\![a_1; \ldots; a_{i-1}; a_{i+1}; \ldots; a_k; b_1; \ldots; b_{j-1}; b_{j+1}; \ldots; b_l]\!] \Longrightarrow \mathit{False}.$$

We have tested our method with proofs generated by MiniSAT and by zChaff, and it is now available as the `sat` and `satx` tactics (the latter based on the definitional CNF conversion described above) in the Isabelle 2005 standard distribution. Table 1 shows

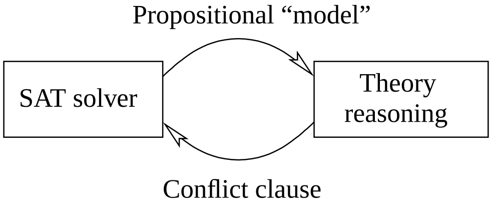

Fig. 1. Cooperation between a SAT solver and a theory reasoner.

experimental results for several examples taken from the TPTP benchmark, based on the solver zChaff. We can successfully check proofs for problems of a few hundred clauses and that require about 10000 binary resolutions. As for the execution time (given in seconds, measured on a Pentium-IV with 1.6 GHz and 512 MB main memory under Linux), "SAT time" refers to the running time of the SAT solver alone whereas "Total time" includes the time taken by Isabelle to reconstruct the proof. One can see that proof checking by Isabelle takes at least two orders of magnitude longer than it takes zChaff to determine unsatisfiability and to produce the proof. This mainly comes from the underlying representation of formulas and theorems in Isabelle, which accommodates arbitrary higher-order syntax, and is not optimized for propositional logic. On the other hand, the default automated tactics offered by Isabelle cannot solve any but the smallest problems of Tab. 1.

Weber [28] has independently suggested a way to perform proof reconstruction in Isabelle from proof traces obtained from SAT solvers. His approach is based on rewriting entire sets of clauses, whereas our sequent representation allows us to operate on comparatively small objects, and our implementation is about an order of magnitude faster for most of the examples of Tab. 1.

4 Proof traces from SMT solvers

The integration of SAT solving with Isabelle is essential for supporting SMT solvers that handle more expressive, though still quantifier-free, languages. Roughly, SMT solvers are SAT solvers working together with theory reasoners, as illustrated in Fig. 1. The information exchanged at the interface are conflict clauses of the theory reasoner, introduced in Sections 4.1 and 4.2. These clauses also contain the essence of a formal proof: the conjunction of the clauses implies the unsatisfiability of the goal formula by purely propositional reasoning. The conflict clauses themselves are proved by laws of equational logic (reflexivity, symmetry, transitivity, and congruence), and in Sect. 4.3 we address the generation of these proofs from the data structures of the underlying decision procedure.

4.1 SAT solvers beyond Boolean logic

Assume that we wish to decide the satisfiability of the formula

$$x = y \wedge \big(f(x) \neq f(y) \vee (\neg p(x) \wedge p(z))\big). \tag{1}$$

We first construct a Boolean abstraction by consistently replacing first-order atoms by Boolean variables. For our example, we obtain the propositional formula

$$p_1 \wedge \big(\neg p_2 \vee (\neg p_3 \wedge p_4)\big) \tag{2}$$

where the Boolean variables p_1, p_2, p_3 and p_4 correspond to the first-order atoms $x = y$, $f(x) = f(y)$, $p(x)$ and $p(z)$. This Boolean abstraction has two (sets of) models that respectively satisfy the literals $\{p_1, \neg p_2\}$ and $\{p_1, \neg p_3, p_4\}$. The first abstract model (i.e. the one that makes p_1 true and p_2 false) does not correspond to a model for the original formula (1), because it is not possible to have a model that would make $x = y$ true and $f(x) = f(y)$ false. The second abstract model corresponds to a concrete one, since $\{x = y, \neg p(x), p(z)\}$ is satisfiable. In general, a formula is satisfiable if and only if there exists a model for the Boolean abstraction of the formula that corresponds to a satisfiable set of literals. Formula (1) is indeed satisfiable.

Notice that this process of first building a Boolean abstraction to extract an abstract model, and then checking the corresponding sets of first-order literals, allows the theory reasoner to operate on sets of literals only. The Boolean structure of formulas is managed efficiently by the SAT solver.

Now if in Formula (1) we replace $p(z)$ by $p(y)$, we obtain the unsatisfiable formula

$$x = y \wedge \big(f(x) \neq f(y) \vee (\neg p(x) \wedge p(y))\big).$$

Its Boolean abstraction is still (2) but p_4 now represents $p(y)$. The models for the abstraction do not correspond to models for the original formula, since the sets of literals $\{x = y, f(x) \neq f(y)\}$ and $\{x = y, \neg p(x), p(y)\}$ are both unsatisfiable. To reduce the satisfiability problem to a purely propositional one, it is sufficient to add conjunctively to (2) *conflict clauses* that express the unsatisfiability of the abstract models in the first-order theory. In our example, we obtain the conflict clauses $\neg p_1 \vee p_2$ and $\neg p_1 \vee p_3 \vee \neg p_4$, corresponding to the valid formulas $x \neq y \vee f(x) = f(y)$ and $x \neq y \vee p(x) \vee \neg p(y)$.

To summarize, the cooperation between the SAT solver and the decision procedure for sets of literals is depicted in Fig. 1. The SAT solver produces models for the Boolean abstraction (that are not necessarily models for the original formula). If the sets of first-order literals that correspond to those models are unsatisfiable, they are rejected by the theory reasoning module, and the Boolean abstraction is *refined* by a conflict clause. For a satisfiable input, an abstract model corresponding to a satisfiable set of first-order literals will eventually be found. For an unsatisfiable input, the successive refinements with conflict clauses will eventually produce an unsatisfiable propositional formula.

4.2 Improving efficiency

In practice, the Boolean abstraction of a given formula will have many models. It is therefore important to find conflict clauses that eliminate not just one, but many abstract models.

The first ingredient to remove several abstract models simultaneously is to extract *partial models* from the propositional abstraction rather than full models. A partial model assigns a truth value to a subset of the propositional variables used in the abstraction, such that every interpretation that extends this partial model is a (full) model.

A partial model assigning n variables for a formula using m variables represents 2^{m-n} full models. Adding a conflict clause to reject a partial model allows us to reject a large number of full models. In [11] we introduced a simple technique to efficiently compute a minimal partial model from a full model for a set of clauses.

Second, the set of literals L corresponding to an abstract (partial) model can still be huge. On the contrary, the very reason for which this set is unsatisfiable is often quite small: it can be expressed as a small subset of L that is unsatisfiable with respect to the theory, the remaining literals being irrelevant. Generating conflict clauses that correspond to small unsatisfiable subsets will, in practice, contribute to an efficient cooperation of the SAT solver with the theory reasoner. The theory reasoner should therefore be able, given an unsatisfiable set of literals, to detect those literals that were really useful to conclude that the set is unsatisfiable. The congruence closure algorithm described in Sect. 4.3 has been designed for this purpose.

Because proof reconstruction essentially relies on the conflict clauses produced by the theory reasoner, it also benefits from this effort to compute small conflict clauses.

4.3 Congruence closure

A congruence closure algorithm decides the satisfiability of a set of ground first-order logic literals in the theory of uninterpreted predicates and functions. It does so by constructing equivalence classes of terms. Two terms belong to the same class if and only if the equalities from the input force the terms to be equal: disequalities play no role in building equivalence classes. A set of literals may be unsatisfiable for two reasons. First, if it contains a pair of complementary literals built from the same predicate such that the corresponding arguments are in the same congruence class as in $\{a = b, p(a,b), \neg p(b,a)\}$. Second, if there is a disequality between two terms in a single congruence class: for instance the set $\{a = b, f(a) \neq f(b)\}$ is unsatisfiable.

Many implementations of congruence closure exist, notably the Nelson-Oppen algorithm [20], and the algorithm due to Downey, Sethi and Tarjan (DST for short) [8]. The simple Nelson-Oppen algorithm has a complexity of $O\left(n^2\right)$ where n is the total number of nodes in the tree or DAG representations of the set of literals. The DST algorithm is more complicated but is of complexity $O(n \log n)$, as long as enter and query operations on a hash table are assumed to be constant in time. haRVey implements a variant of DST: its complexity is $O(n \log n)$, and terms are represented as DAGs for maximal sharing of subterms. This algorithm is described in detail in [11].

Very abstractly, the congruence closure algorithms work on a partition of a set of terms. This set must be closed under the subterm relation. Initially each term is alone in its own class. The partition of terms is successively updated to take into account a set of equalities. When an equality $t = t'$ is given to the algorithm, the classes for terms t and t' are merged. Any class merge may produce the merge of further classes because of the congruence rule

$$\frac{t_1 = t_1' \quad \cdots \quad t_n = t_n'}{f(t_1, \dots t_n) = f(t_1', \dots t_n')} \tag{3}$$

For instance, assume x and y belong to two classes that are merged. Then, if $f(x)$ and $f(y)$ belong to two different classes, those two classes should also be merged. Implementations of congruence closure algorithms rely on efficient data structures to represent classes of terms, and on indexing techniques to quickly find the classes that have to be merged because of the congruence rule.

As an example, consider the set of terms

$$\big\{a, b, f(a), g(a), g(b), g(g(a)), f(g(b)), g(f(a))\big\}.$$

This set is closed under the subterm relation. Assume that we wish to compute the equivalence classes of this set of terms for the equalities $a = f(a)$, $f(a) = f(g(b))$, $f(g(b)) = g(f(a))$ and $g(b) = g(g(a))$. Initially every term is in its own class. Processing the equality $a = f(a)$ merges the classes for a and $f(a)$. Because of congruence, the classes for $g(a)$ and $g(f(a))$ will also be merged. Taking into account the equality $f(a) = f(g(b))$ merges the classes for the two terms, without inducing any further merging operations. At this point, the partition of terms is

$$\big\{\{a, f(a), f(g(b))\}, \{b\}, \{g(a), g(f(a))\}, \{g(b)\}, \{g(g(a))\}\big\}.$$

Now, processing the equality $f(g(b)) = g(f(a))$ merges the classes for those two terms, that is, the classes for a and $g(a)$. This entails, by congruence, that $g(a)$ and $g(g(a))$ are equal. Processing the last equality $g(b) = g(g(a))$ results in all terms except for b forming a single class.

Notice that only the congruence axiom is applied explicitly: the data structure (i.e. a partition of terms) makes implicit the equivalence properties of equality, i.e. the laws of reflexivity, symmetry, and transitivity. Two classes are merged because two terms are found to be equal, either because a literal in the input equates them, or by propagation according to the congruence rule. If we want to store that information for later use, we can store the pair of terms that are responsible for a merge, together with its reason. This information is enough to reconstruct, for any two terms of a class, a small set of equations that entail their equality.

Back to the previous example, we can draw a graph that summarizes the successive merges. The nodes of the graph are just the terms handled by the algorithm. Each time two classes are merged because of an equation in the input (for instance $a = f(a)$), we draw a plain edge between the left- and right-hand side terms of the equation, and label the edge by the equation. If two classes are merged because of an application of the congruence rule (for instance $g(a)$ and $g(f(a))$), we draw a dashed edge. The full merge-history graph for the congruence closure algorithm applied to our example appears in Fig. 2.

It is easy to verify that merge-history graphs enjoy the following properties:

- the equality of two terms is entailed by a set of equations (i.e. the two terms are in the same class), if and only if there is a path between the corresponding nodes in the merge-history graph;
- there is a unique path between any two terms in the same class;
- the equality between two terms in the same class follows by reflexivity, symmetry, and transitivity of equality from the conjunction of the edge labels along the path between the two terms;

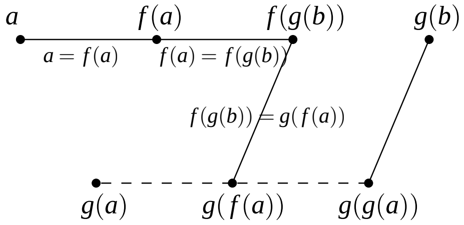

Fig. 2. Merge-history graph.

– two terms connected by a dashed edge have the same topmost symbol, and the corresponding subterms are in the same classes. The equality between those two terms follows, by congruence only, from equalities between direct subterms.

As a consequence, it is easy to use a merge-history graph to decompose[1] the justification of the equality of two terms into elementary steps that involve either only congruence or only reflexivity, symmetry, and transitivity of equality.

Assume that the algorithm concludes the unsatisfiability of a set containing the equalities $a = f(a)$, $f(a) = f(g(b))$, $f(g(b)) = g(f(a))$ and $g(b) = g(g(a))$ and the disequality $a \neq g(b)$, possibly among many other literals. It does so by building the classes of terms according to the equalities in the input, and then discovering a conflict with the disequality $a \neq g(b)$. At this point, the algorithm uses the merge-history graph to produce a minimal unsatisfiable subset of the input literals and outputs a justification of the unsatisfiability of this set that will be used for proof reconstruction.

The idea of using representations similar to merge-history graphs to extract small conflict sets has appeared before [19, 12, 22], but we are not aware of a previous use of these graphs to justify *a posteriori* the equality of terms by elementary proof steps.

5 Proof Reconstruction for Congruence Closure

In this section we describe our implementation of the interface between Isabelle and haRVey, with the focus on proof reconstruction for the congruence closure reasoning part of haRVey, as it is described in Section 4. This interface is implemented as a proof method called `rv` in Isabelle, i.e., as an ML program.

The idea behind the interface is not to use haRVey to give a complete proof for a given goal, rather, it is used to provide a list of intermediate lemmas, namely the conflict clauses described in Section 4, to guide proof search in Isabelle. More precisely, given a goal formula F, the interface performs the following steps:

1. Convert the negated goal ($\neg F$) to SMT-LIB format and give it to haRVey.
2. If $\neg F$ is unsatisfiable, haRVey produces a list of formulas $C_1, \ldots, C_n$ (the conflict clauses) along with a proof trace for each C_i. If $\neg F$ is satisfiable, the interface displays the model found by haRVey and aborts.

[1] Decomposition terminates due to the inductive construction of merge-history graphs from the two elementary merge operations.

3. Construct a proof for each conflict clause C_i in Isabelle, based on the justification output by haRVey.
4. Construct a proof for the sequent $[\![\neg F; C_1; \cdots ; C_n]\!] \Longrightarrow \mathit{False}$.
5. Apply modus ponens to the formulas obtained in (3) and (4) to get $\neg F \Longrightarrow \mathit{False}$, and hence prove F.

Step (5) is implemented straightforwardly in Isabelle using resolution. Step (4) applies the SAT interface described in Section 3. We now describe the proof reconstruction for each conflict clause C_i.

The haRVey prover produces a compact proof trace for each conflict clause, summarizing the kind of reasoning needed to prove the clause. These proof traces consist of lists of sequents labeled with hints how they can be proved, as follows:

```
TRANS: <sequent>
CONGR: <sequent>
PRED : <sequent>
INEQ : <sequent>
```

and end with the line

```
CONFL: <formula>
```

The formula following the keyword `CONFL` is the conflict clause. We shall look at the overall structure of the proof trace, before explaining in details the meaning of the other keywords. Implicit in the proof format is the (backward) resolution proof for deriving the conflict clause. More precisely, suppose that the list of sequents preceding the conflict clause are

$$\begin{array}{l}
l_1 : [\![C_{11}; \cdots ; C_{1k_1}]\!] \Longrightarrow B_1 \\
\vdots \qquad\quad \vdots \\
l_n : [\![C_{n1}; \cdots ; C_{nk_n}]\!] \Longrightarrow B_n
\end{array}$$

where each label l_i is either `TRANS`, `CONGR`, `PRED` or `INEQ`. The first sequent is always a statement of a contradiction, i.e., B_1 is *False*. The assumptions C_{ij} in the sequent i satisfy the following requirement: each of them either appears in negated form in the conflict clause, or it is the conclusion of a later sequent, i.e. it is B_k, for some $k > i$. The conflict clause is therefore proved by contradiction, as the result of resolving its negation with all the intermediate sequents above until *False* is derived. The corresponding inference in Isabelle looks something like:

$$\frac{[\![\neg C; C_{n1}; \cdots ; C_{nk_n}]\!] \Longrightarrow B_n \quad \cdots \quad [\![\neg C; C_{11}; \cdots ; C_{1k_1}]\!] \Longrightarrow \mathit{False}}{C}$$

where C is the conflict clause. This is a valid inference because each C_{ij} is either justified by $\neg C$, or it is B_k for some $k > i$. In the implementation, this inference scheme is realized by a series of resolution steps between sequent i, for several (possibly all) $i > 1$, with the first sequent.

We shall now turn to the proofs of the intermediate sequents. The keywords preceding the sequents indicate the kind of reasoning needed to prove the sequent. The

`INEQ:` $\llbracket a = g\ b;\ \mathbf{a} \neq \mathbf{g\ b} \rrbracket \Longrightarrow \mathit{False}$

`TRANS:` $\llbracket \mathbf{a} = \mathbf{f\ a};\ \mathbf{f\ a} = \mathbf{f\ (g\ b)};\ \mathbf{f\ (g\ b)} = \mathbf{g\ (f\ a)};$
$\quad g\ (f\ a) = g\ (g\ a);\ \mathbf{g\ b} = \mathbf{g\ (g\ a)} \rrbracket \Longrightarrow a = g\ b$

`CONGR:` $f\ a = g\ a \Longrightarrow g\ (f\ a) = g\ (g\ a)$

`TRANS:` $\llbracket g\ (f\ a) = g\ a;\ \mathbf{f\ (g\ b)} = \mathbf{g\ (f\ a)};\ \mathbf{f\ a} = \mathbf{f\ (g\ b)} \rrbracket \Longrightarrow f\ a = g\ a$

`CONGR:` $\mathbf{a} = \mathbf{f\ a} \Longrightarrow g\ (f\ a) = g\ a$

`CONFL:` $a = g\ b \vee a \neq f\ a \vee f\ a \neq f\ (g\ b) \vee f\ (g\ b) \neq g\ (f\ a) \vee g\ b \neq g\ (g\ a)$

Fig. 3. Proof trace for a conflict clause.

keyword `PRED` indicates that the sequent can be proved using one substitution and followed by proof-by-contradiction. That is, the sequent in this case is of the form:

$$\llbracket s = t; P\ s; \neg(P\ t) \rrbracket \Longrightarrow \mathit{False}.$$

The keyword `INEQ` indicates that the sequent contains a contradictory pair of equalities:

$$\llbracket s = t; s \neq t \rrbracket \Longrightarrow \mathit{False}.$$

Proof reconstruction for both cases are easily done in Isabelle using substitution and proof by contradiction.

The keyword `TRANS` means that the sequent is provable by using the reflexivity, symmetry, and transitivity of equality alone. We have implemented a special tactic in Isabelle to do this type of equality reasoning. We could have used the built-in simplifier tactics (based on rewriting) but these may not terminate in case of equalities that result in looping rewrite rules.

The label `CONGR` indicates that the sequent is provable by using the congruence rule (3). As in the case with `TRANS`, we could use Isabelle's built-in rewriting engine, but faster proofs are obtained using a custom-built tactic. Because terms are represented in curried notation in Isabelle/HOL, we only need to rely on a single axiom scheme, independently of the arity of the function symbol:

$$\llbracket f = g;\ x = y \rrbracket \Longrightarrow f\ x = g\ y.$$

Proof construction proceeds recursively from the last argument of a function application: to prove $f\ x_1 \cdots x_n = g\ y_1 \cdots y_n$, first show $x_n = y_n$ and then recursively construct a proof for $f\ x_1 \cdots x_{n-1} = g\ y_1 \cdots y_{n-1}$.

Example. Given the formula (cf. Fig. 2)

$$a = f\ a \wedge f\ a = f\ (g\ b) \wedge f\ (g\ b) = g\ (f\ a) \wedge g\ b = g\ (g\ a) \Longrightarrow a = g\ b,$$

haRVey produces one conflict clause, which is just the formula itself, but in CNF. It also produces a proof trace for the conflict clause, which appears in Fig. 3. For better readability, we have presented in boldface letters the (dis)equations that come from the conflict clause (the last line of the proof trace). The remaining (dis)equations appear as conclusions of sequents below in the proof trace. It is straightforward to construct a refutation proof from the above sequents.

Formula	Size		# confl.	Times (s)	
	nodes	atoms	clauses	haRVey	Isabelle
SEQ004-size5	18795	6967	143	0.41	115.68
SEQ011-size2	7355	3471	73	0.02	9.69
SEQ015-size2	331	47	20	0.02	3.10
SEQ020-size2	7963	3775	74	0.02	7.16
SEQ032-size2	255	43	20	0.01	2.66
SEQ042-size2	947	293	49	0.09	11.17
SEQ050-size2	779	213	105	0.11	32.42

Table 2. Running time for proof reconstruction for congruence closure.

Benchmark. We have tested our interface to haRVey with proof reconstruction with a number of example formulas. The running times needed to solve these problems using the `rv` tactic are given in Tab. 2. The benchmarks were run on a machine with a 1.5 GHz Intel Pentium-IV processor and 1024 MB memory under Linux. For each formula, we indicate the number of nodes in the dag representation of the formula, the number of distinct atoms that occur in the formula, and the number of conflict clauses produced by haRVey. We also indicate the times taken by haRVey to refute the formula and output the proof trace, and by Isabelle to parse the proof trace and check the proof.

For all these examples, the running time it took for haRVey to find a refutation is negligible (less than a second). For formulas of small size, the number of conflict clauses produced is up to 20 clauses. In those cases, proof reconstruction succeeds within one to five seconds. For larger test cases, we make use of some of the benchmark problems used in the SMT 2005 competition. Note that "small problems" in the competition are actually quite large formulas, in comparison to the kind of lemmas shown in Sect. 2. We see that the times taken for proof reconstruction in Isabelle are again more than two orders of magnitude larger than the running times of haRVey, and that they depend mostly on the number of conflict clauses produced (remember also that each conflict clause is justified by a number of low-level reasoning steps).

None of these examples succumbs to Isabelle's existing automatic proof methods. Isabelle 2005 contains a preliminary implementation, without proof reconstruction, of the combination of resolution-based theorem provers and Isabelle described by Meng et al. [16], and we have not succeeded in using this implementation to prove the examples of Tab. 2: for the larger examples, the first-order prover did not complete within 5 minutes. For the smaller examples, Isabelle was unable to parse the result of the prover, which also took orders of magnitude longer than haRVey. This experiment seems to indicate to us that the combination with an SMT solver can be useful for certain problems.

6 Conclusion

We have proposed a technique for combining interactive proof assistants and proof-producing SMT solvers. Because proofs are certified by the trusted kernel of the interactive prover, theorems established in this way come with the same soundness guarantees as those theorems established interactively. The combination with an efficient external reasoner allows us to significantly raise the degree of automation while retaining the expressiveness of the input language for specification. Our current implementation combines Isabelle/HOL with the fragment of haRVey that handles quantifier-free first-order logic with uninterpreted function and predicate symbols. However, the overall approach extends to other interactive provers and to other decidable fragments of first-order logic. In particular, we plan to address linear arithmetic along the same lines by making haRVey output compact proof traces that can be replayed within Isabelle/HOL.

On the implementation level, we observe that the time Isabelle takes to replay a proof trace significantly exceeds the time taken by haRVey to find the proof, although basically no proof search is required. We believe that a significant part of this run-time penalty comes from the overhead incurred by the support for higher-order abstract syntax, but more investigation will be necessary into this matter. It also remains to be seen whether efficiency of proof reconstruction is a big issue for those verification conditions that we expect to see in practical applications (where we are mostly interested in stronger theories). Also, proof reconstruction can be done off-line, whereas an oracle-style combination should be sufficient for interactive proof.

On a conceptual level, we propose to study and identify uniform formats for proof traces for SMT solvers, akin to the SMT-LIB input format, to enable comparisons between different solvers and to standardize the interface towards interactive proof assistants (and, in fact, independent proof checkers).

Acknowledgements. We are grateful to Kamal Kant Gupta, who contributed to the syntactic translation from Isabelle to the SMT format, and to Tjark Weber for his help with integrating and maintaining our code for SAT proofs within the Isabelle distribution.

References

1. M. Baaz, U. Egly, and A. Leitsch. Normal form transformations. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 5, pages 273–333. Elsevier Science B.V., 2001.
2. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, volume 3114 of *LNCS*, pages 515–518. Springer, Apr. 2004.
3. D. Barsotti, L. Prensa-Nieto, and A. Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. In *Proc. of the Fifth Workshop on Automated Verification of Critical Systems (AVOCS)*, ENTCS, 2005. to appear.
4. M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *J. Autom. Reasoning*, 29(3-4):253–275, 2002.
5. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. The MathSAT 3 System. In *CADE*, volume 3632 of *LNCS*, pages 315–321, Tallinn, Estonia, 2005. Springer.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.
7. D. Déharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code. In *Software Engineering and Formal Methods (SEFM)*, pages 220–228. IEEE Comp. Soc., Sept. 2003.

8. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
10. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *LNCS*, pages 246–249. Springer, 2001.
11. P. Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Université de Liège, Belgium, Sept. 2004.
12. P. Fontaine and E. P. Gribomont. Using BDDs with combinations of theories. In M. Baaz and A. Voronkov, editors, *LPAR*, volume 2514 of *LNCS*, pages 190–201. Springer, 2002.
13. J. Hurd. Integrating Gandalf and HOL. In *Theorem Proving in Higher-Order Logics (TPHOLs'99)*, volume 1690 of *LNCS*, pages 311–322, Nice, France, 1999. Springer.
14. A. Mahboubi. Programming and certifying the CAD algorithm inside the coq system. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl, Germany, 2005.
15. A. Meier. TRAMP: Transformation of machine-found proofs into ND-proofs at the assertion level. In D. McAllester, editor, *CADE*, volume 1831 of *LNCS*, pages 460–464, Pittsburgh, PA, 2000. Springer.
16. J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, to appear.
17. D. G. Mitchell. A SAT solver primer. *EATCS Bulletin*, 85:112–133, 2005.
18. G. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Logics in Computer Science (LICS'98)*, pages 93–104. IEEE Press, 1998.
19. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.
20. G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.
21. Q. H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *J. Autom. Reason.*, 29(3-4):309–336, 2002.
22. R. Nieuwenhuis and A. Oliveras. Union-find and congruence closure algorithms that produce proofs. In C. Tinelli and S. Ranise, editors, *PDPAR*, 2004.
23. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.
24. S. Ranise and C. Tinelli. The SMT-LIB standard : Version 1.1, Mar. 2005.
25. J. H. Siekmann and many others. Proof development with OMEGA. In *CADE*, pages 144–149, 2002.
26. A. Tiu. Formalization of a generalized protocol for clock synchronization in Isabelle/HOL. Archive of Formal Proofs: `http://afp.sourceforge.net`, 2005.
27. G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, volume II, pages 115–125. 1970.
28. T. Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. In J. Hurd, E. Smith, and A. Darbari, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005), Emerging Trends*, pages 180–189. Oxford Univ. Comp. Lab., Prog. Res. Group, 2005. Report PRG-RR-05-02.
29. L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In A. Voronkov, editor, *CADE*, volume 2392 of *LNCS*, pages 295–313. Springer, 2002.
30. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker. In *Design, Automation and Test in Europe (DATE 2003)*, pages 10880–85, Munich, Germany, 2003. IEEE Comp. Soc.