SATISFIABILITY SOLVING FOR SOFTWARE VERIFICATION

David Déharbe
DIMAp/UFRN (Natal, Brazil)

Silvio Ranise
INRIA-Lorraine (Nancy, France)

ABSTRACT

Many approaches to software verification require to check the satisfiability of (possibly quantified) first-order formulae in theories modeling user-defined data types, the memory model used by the programming language, and so on. For such verification techniques, it is of crucial importance to have satisfiability solvers which are both predictable and flexible, i.e. capable of automatically discharging the largest possible number of proof obligations coming from the widest range of verification problems. In this paper, we describe our approach to build predictable and flexible satisfiability solvers by combining (an extension of) resolution theorem proving, arithmetic reasoning, Boolean solving, and some transformations on the proof obligations (such as definition unfolding or theory reduction). We show the viability of the approach by describing the experimental results obtained with an implementation of the proposed techniques on a set of proof obligations extracted from various software verification problems, in particular the certification of auto-generated aerospace code.

CONTEXT AND MOTIVATION

Many approaches to software verification, ranging from applications of Hoare logic to software model checking (see e.g., [1]) and, more recently, to program analysis,[1] require to discharge some proof obligations, i.e. checking that some formula (usually of first-order logic with equality) is satisfiable in a given theory modeling the user-defined data types of the software system under scrutiny, the memory model used by the programming language, its type system, and so on. For such verification techniques, it is of crucial importance to have *satisfiability solvers* which are both *predictable* and *flexible*, i.e. capable of automatically

[1]`http://www.mit.edu/~vkuncak/projects/jahob`

discharging the largest possible number of proof obligations coming from the widest range of verification problems. Indeed, this task is far from simple.

We identify two key challenges to build predictable and flexible satisfiability solvers:

- to provide satisfiability procedures for the theories most commonly used in software verification that capture interesting classes of software properties so to have the highest possible degree of predictability (i.e. the guarantee of termination with a correct answer), and
- to widen the scope of applicability of the available satisfiability procedures so to be able to discharge proof obligations which are usually expressed in extensions of the supported (decidable) theories (see [2] for an extensive discussion on this issue).

In this paper, we describe our approach to build satisfiability solvers for software verification by tackling the above challenges. A careful integration of the following three techniques is at the core of our approach:

- Superposition theorem proving (see e.g., [3]) and satisfiability checking for Linear Arithmetics to reason about the data flow of software systems;
- Boolean solving to handle their control flow;
- Two heuristics to reduce—*a priori*—the search space of the satisfiability checking problem.

SATISFIABILITY OF QUANTIFIER-FREE FORMULAE

Let T be a theory with decidable satisfiability problem, i.e. it is decidable to check whether an arbitrary conjunction of literals of T is satisfiable. The *Satisfiability Module Theory (SMT) problem* is the problem of checking if a quantifier-free first-order formula ϕ of T is satisfiable, i.e. whether $T \wedge \phi$ is satisfiable. An *SMT algorithm* is a decision procedure for the SMT problem.

Satisfiability Modulo Theory checking

In the following, β^p is used to denote a propositional assignment; π is used to denote a conjunction of ground first order literals, and π^p its boolean abstraction; in general, we use the superscript "$...^p$" to denote boolean formulas or assignments. We represent propositional assignments β^p indifferently as sets of propositional literals $\{l_i\}_i$ or as conjunctions of propositional literals $\bigwedge_i l_i$; in both cases the intended meaning is that a positive literal v (resp. a negative literal $\neg v$) denotes that the variable v is assigned to true (resp. false).

Figure 1 depicts $\mathsf{Bool} + T$, the SMT algorithm underlying (with different variants) several systems such as Mathsat [4], DPLL(T) [5], TSAT [6], ICS [7],

```
function Bool + $T$($\phi$: quantifier-free formula)
1     $\phi^p \longleftarrow fol2prop(\phi)$;
2     $A^p \longleftarrow fol2prop(Atm(\phi))$;
3     while Bool-satisfiable($\phi^p$) do
4         $\beta^p \longleftarrow pick\_total\_assign(\phi^p)$;
5         $(\rho, \pi) \longleftarrow T\text{-}satisfiable(prop2fol(\beta^p))$;
6         if ($\rho ==$ sat) then return sat;
7         $\phi^p \longleftarrow \phi^p \wedge \neg fol2prop(\pi)$;
8     end while;
9     return unsat;
end
```

Figure 1: A simplified view of traditional SMT algorithms

CVC-Lite [8], and **haRVey**[2]. For the sake of simplicity, we only give a naive formulation, based on the enumeration of total assignments and we ignore more realistic representations based on DPLL-style assignment enumeration (for this the interested reader is pointed to one of the papers above). The algorithm relies on the existence of a bijection between the atoms of ϕ and a suitable set of propositional letters. In particular, we call *Atm* the set of ground atoms in ϕ and P_{Atm} be a set of propositional letters s.t. the cardinality of *Atm* is equal to that of P_{Atm} and $Atm \cap P_{Atm} = \emptyset$. Let *atm2prop* be a bijective function from *Atm* to P_{Atm}. *fol2prop* is a mapping from the quantifier-free formula ϕ to a propositional formula ϕ^p as the homomorphic extension of *atm2prop* to ϕ; *prop2fol* is the inverse of *fol2prop*; its result is a conjunction of ground first-order literals. Basically, the truth assignments for (the propositional abstraction of) ϕ are enumerated, and checked against T. The procedure either concludes satisfiability if one such model is found, and returns with failure otherwise. The ancillary functions in Figure 1 are assumed to satisfy the following requirements:

- *pick_total_assign* returns a total assignment to the propositional variables in ϕ^p. So, β^p is a conjunction of propositional literals;
- *T-satisfiable*(β) detects if β is satisfiable in T. If so, it returns (sat, $\emptyset$); otherwise, it returns (unsat, π), where π is satisfiable in T and $\pi \subseteq \beta$ is called a *theory conflict set*.

It is easy to show that the algorithm terminates, and it is correct and complete (see e.g., [9] for more details). Termination holds since there exists only a finite number of possible assignments, and none is considered more than once: the set of propositional assignments to ϕ^p is monotonically decreasing, since each iteration excludes at least one of them. Correctness follows from the basic definitions of truth assignment and interpretation: sat is returned only if β^p

[2]See `http://combination.cs.uiowa.edu/smtlib` for more details and pointers to these systems

propositionally satisfies ϕ, and there exists an interpretation refining it. Completeness follows from the fact that we never eliminate a consistent assignment.

Superposition-based satisfiability procedures

To be able to execute the algorithm in Figure 1, we are left with the problem of building the function *T-satisfiable*. In order to do this, we adopt the rewriting approach described in [10] which uses superposition as a framework to synthesize satisfiability procedures for various theories of interests for verification.

The superposition calculus (see [3] for an overview) checks the satisfiability of arbitrary sets of first-order clauses. It consists of a set of rules which are derived from resolution and which are especially designed for the efficient treatment of equality. Although equality dramatically enlarges the search space of resolution based provers, the effectiveness of superposition lies in some powerful criteria (such as term ordering) to prune the search space while maintaining completeness. Any fair application of the rules of the calculus to an unsatisfiable set of clauses derives $\perp$ (also called the empty clause).[3] Roughly, a saturation prover (such as the E prover [12]) amounts to a clever mechanization of the exhaustive application of the rules of the calculus to any finite set of first-order clauses (indeed, a lot of sophistication is required to obtain efficient implementations). In general, the process of applying the rules of the calculus to a set of clauses may not terminate since first-order logic is undecidable. If for a class $\mathcal{C}$ of clauses, we are able to prove that this process terminates, then we are entitled to conclude that the calculus is a satisfiability procedure for $\mathcal{C}$ given its refutation completeness. The methodology to build satisfiability procedures described in [10] is based on this simple observation and it is organized in two phases.

Let $Ax(\mathcal{T})$ be a finite set of equational clauses axiomatizing $\mathcal{T}$ and β a conjunction of ground literals.[4] The first phase amounts to flattening all ground literals in β. A flat literal is either an equality of the form $f(c_1, ..., c_n) = c_{n+1}$ or the negation of an equality of the form $c_1 \neq c_2$ (where $c_1, ..., c_{n+1}$ are constants and f is an n-ary function symbol).[5] Flattening is done by extending the signature with "fresh" constant symbols for all the distinct non-constant sub-terms in β. It is easy to see that flattening preserves satisfiability. Let β' be the result of flattening β. The second phase consists of exhaustively applying the rules of the superposition calculus to the clauses in $Ax(\mathcal{T}) \wedge \beta'$. As shown in [10], the second phase terminates for many interesting theories such as the theory of lists, arrays, sets, and their combination. We illustrate the approach

[3]Fairness means that if some inference is possible, it will be performed at some step unless one of the parent clauses gets simplified or deleted (see, e.g. [11] for a formal definition).

[4]Here, we are assuming that the theory $\mathcal{T}$ can be finitely presented by a set of first-order clauses. As shown in [10], many theories of practical interest admits such an axiomatization.

[5]To check satisfiability, predicate applications can be turned into function application as follows. For p an n-ary predicate symbol is in the input set of literals, the literal $p(t_1, ..., t_n)$ is translated to $f_p(t_1, ..., t_n) = \mathtt{tt}$ where f_p is a "fresh" n-ary function symbol and $\mathtt{tt}$ is a distinct constant. Similarly, $\neg p(t_1, ..., t_n)$ is translated to $f_p(t_1, ..., t_n) \neq \mathtt{tt}$. So, via this satisfiability preserving transformation, flattening can be applied also to non-equational literals.

on a simple example. Example Let us consider the theory $\mathcal{A}$ of arrays. The signature $\Sigma_{\mathcal{A}}$ contains the binary function symbol read, the ternary function symbol write (abbreviated below with rd and wr, respectively), and a finite set of constant symbols (written in small letters). The theory $\mathcal{A}$ is axiomatized by the following set $Ax(\mathcal{A})$ of axioms:

$$\forall A, I, E.(\mathsf{rd}(\mathsf{wr}(A, I, E), I) = E) \tag{1}$$

$$\forall A, I, J, E.(I \neq J \Rightarrow \mathsf{rd}(\mathsf{wr}(A, I, E), J) = \mathsf{rd}(A, J)), \tag{2}$$

where capitalised letters are implicitly universally quantified variables. Let β be the following conjunction of literals:

$$a = \mathsf{wr}(\mathsf{wr}(a, i, \mathsf{rd}(a, j)), j, \mathsf{rd}(a, i)) \wedge \mathsf{rd}(a, i) \neq \mathsf{rd}(a, j).$$

Flattening β yields the conjunction of the following (ground) literals:

$$\begin{aligned} c_1 &= \mathsf{rd}(a, i) & (3)\\ c_2 &= \mathsf{rd}(a, j) & (4)\\ c_3 &= \mathsf{wr}(a, i, c_2) & (5)\\ c_4 &= \mathsf{wr}(c_3, j, c_1) & (6)\\ a &= c_4 & (7)\\ c_1 &\neq c_2 & (8) \end{aligned}$$

where c_1, c_2, c_3, c_4 are "fresh" constants. The second phase amounts to exhaustively apply the rules of the superposition calculus given in [3] to $Ax(\mathcal{A}) \wedge \beta'$. For this example, standard rewriting and the following simplified version of the calculus is sufficient:

$$\frac{l' = r' \quad l = r}{(l[r'] = r)\sigma}\ \textit{Superposition} \qquad \frac{l' = r' \quad l \neq r}{(l[r'] \neq r)\sigma}\ \textit{Paramodulation} \qquad \frac{s \neq t}{\bot}\ \textit{Reflection}$$

where σ is the most general unifier between l' and a sub-term of l, for *Superposition* and *Paramodulation*, and the terms s and t must be unifiable, for *Reflection* (we have omitted the provisos about ordering constraints for simplicity).

A *Superposition* between (6) and axiom (1) yields $c_1 = \mathsf{rd}(c_4, j)$, which can be rewritten to $c_1 = \mathsf{rd}(a, j)$ by (7). A *Superposition* between $c_1 = \mathsf{rd}(a, j)$ and (4) gives $c_1 = c_2$, which by *Paramodulation* with (8) yields $c_1 \neq c_1$. By applying *Reflection* to $c_1 \neq c_1$, we immediately obtain $\bot$, which proves that β is $\mathcal{A}$-unsatisfiable.

In order to implement T-*satisfiable*(β), it is sufficient to implement flattening (which can be easily done), and then run a superposition theorem prover on $Ax(T) \wedge \beta'$. Furthermore, all superposition theorem provers are capable of returning a proof of the unsatisfiability of a set of clauses. By analyzing such a proof, it is possible to identify the sub-set of (unit) clauses in β' which are used to derive the empty clause. This offers an elegant and uniform way to obtain the theory conflict set π. Since the proof found is not necessarily the one using fewest assumptions, the returned set of literals is not guaranteed to be minimal but it is usually a good over-approximation.

It is important to emphasize the *flexibility* offered by this approach with respect to specialized decision procedures. In fact, when a decision procedure for a new theory is needed, it is only necessary to feed the superposition theorem prover with the axioms of the theory and the literals to be proved (un-)satisfiable, whereas with specialized decision procedures a major design and coding effort should be undertook.

The Nelson-Oppen combination method

The price to pay for using superposition to build satisfiability procedures in a flexible and uniform way is a restriction in the class of theories which can be handled. In fact, it is well-known that superposition theorem proving has difficulties in handling (decidable fragments of) arithmetics. The technique described above inherits this limitation so that it can efficiently handle (equational) theories which are axiomatized by a finite set of clauses. In order to overcome this limitation, we can use the Nelson and Oppen (N&O) combination schema [13] to combine saturation theorem proving and arithmetic reasoning (for details and preliminary experiments, see [14]).

The N&O combination method enables us to solve the problem of checking the satisfiability of a conjunction Φ of quantifier-free literals in the union of two signature-disjoint theories T_1 and T_2 for which two satisfiability procedures are available. Since the literals in Φ may be built over symbols in T_1 or in T_2, we need to *purify* them by introducing fresh constants to name sub-terms. This process leaves us with a conjunction $\Phi_1 \wedge \Phi_2$ which is equisatisfiable to Φ where Φ_i contains only literals with symbols of T_i, for $i = 1, 2$.[6] In this way, literals in Φ_i can be *dispatched* to the available decision procedure for T_i.

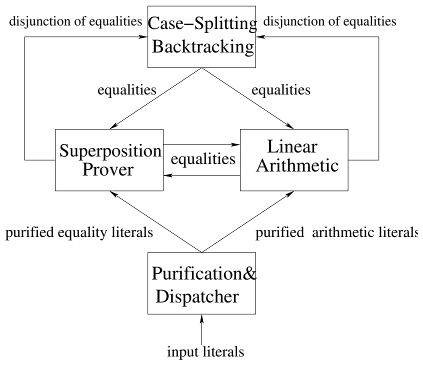
To show the correctness of the N&O method (see e.g., [15]), the theories T_1 and T_2 must be stably-infinite. Roughly, a theory is *stably infinite* if any satisfiable quantifier-free formula is satisfiable in a model having an infinite cardinality. All theories considered in this paper (the theory of equality, the theory of arrays, and the theory of Linear Arithmetic) are stably infinite.

An efficient implementation of the N&O method is based on the availability of satisfiability procedures with (at least) the following properties (see [16] for an in depth discussion on this and other efficiency issues):[7]

Deduction completeness. It must be capable of efficiently detecting elementary clauses (i.e. a clause whose literals are equalities between constants which occur in purified literals belonging to both theories) which are implied by the input conjunction of literals.

[6]Notice that flat literals are purified.

[7]Incrementality and resettability are two other common requirements in this context. Unfortunately, it is difficult to modify modern state-of-the-art superposition provers so to make them incremental and resettable. So, we do not discuss here these issues. However, see [17] for a possible way to overcome this problem.



Case–Splitting Backtracking
disjunction of equalities
disjunction of equalities
equalities
equalities
Superposition Prover
equalities
Linear Arithmetic
purified equality literals
purified arithmetic literals
Purification& Dispatcher
input literals

Figure 2: The Nelson-Oppen Combination Method

The N&O method for satisfiability procedures satisfying the requirements above is depicted in Figure 2 when T_1 is the theory of equality for which the superposition calculus is known to be a satisfiability procedure (see e.g., [10]) and T_2 is Linear Arithmetic (LA) for which various satisfiability procedures are available (see e.g., [18]). Such a combination method simply consists of exchanging elementary clauses between the two procedures until either unsatisfiability is reported for one of the two component theories or no more elementary clauses can be exchanged. In the first case, we report the unsatisfiability of the input formula; in the second case, we report its satisfiability. Only finitely many elementary clauses can be constructed by using the constants of both Φ_1 and Φ_2, and so the N&O method terminates.

It is sufficient to exchange only elementary equalities when combining convex theories. A theory is *convex* if for any conjunction of equalities Γ, a disjunction D of equalities is entailed by Γ if and only if some disjunct of D is entailed by Γ. Examples of convex theories are the theory of equality, the theory of lists, and the theory of Linear Arithmetic over the Rationals ($LA(\mathcal{R})$). Since both procedures are assumed to be deduction complete, the combination method only needs to pass around elementary equalities between the procedures as soon as they discover them.

When combining at least one non-convex theory such as the theory of arrays or the theory of Linear Arithmetic over the Integers ($LA(\mathcal{I})$), the combination method is more complex since the procedures should exchange elementary clauses. Although the procedures are capable of deriving the entailed elementary clauses, their processing is problematic since they are only capable of handling conjunctions of literals. The standard solution is to case-split on the derived elementary clauses and then consider each disjunct in turn by using a backtracking procedure.

Although surprising (since superposition is not known to be complete for consequence finding, i.e. we are not guaranteed that a clause which is a logical consequence of a set of clauses will be eventually derived by applying the rules of the calculus), satisfiability procedures obtained by superposition are deduction complete, i.e. derive sufficiently many elementary clauses to be efficiently combined *à la* N&O with other procedures (see [17] for details). We illustrate the combination schema with a simple example. Example Let us consider the theory T obtained as the union of LA($\mathcal{R}$) and the theory $\mathcal{A}$ of arrays (cf. Example). A deduction complete satisfiability procedure for $\mathcal{A}$ can be built by superposition [17]. A deduction complete satisfiability procedure for LA($\mathcal{R}$) can be built by modifying the Fourier-Motzkin elimination method (see e.g., [18] for details). Now, consider the problem of checking the satisfiability (in T) of the following conjunction of literals (which are already purified):

$$\mathsf{rd}(b,i) = e_2 \wedge \mathsf{wr}(a,i,e_1) = b \wedge \tag{9}$$

$$e_1 + e_2 \neq 2e_2 \tag{10}$$

(notice that e_1 and e_2 occur both in literals of LA($\mathcal{R}$), cf. (10), and of $\mathcal{A}$, cf. (9). First, (9) is sent to the superposition prover which terminates without deriving the empty clause. However, it is capable of performing the following derivation (again we consider the simplified superposition calculus introduced in Example):

$\mathsf{rd}(b,i) = e_1$	by *Superposition* between (1) and the second literal of (9)
$e_1 = e_2$	by rewriting the previous with the first literal of (9)

Then, the derived elementary (unit) clause $e_1 = e_2$ is sent with (10) to the satisfiability procedure for LA($\mathcal{R}$) which immediately reports unsatisfiability. The N&O combination schema reports the unsatisfiability of the conjunction of (9) and (10).

SATISFIABILITY OF ARBITRARY FIRST-ORDER FORMULAE

We now consider two extensions of the SMT problem, introduced at the beginning of the previous section. First, we consider a theory T whose satisfiability problem is not necessarily known to be decidable. This is quite common in software verification where theories are frequently obtained by modularly composing and extending small theories. To be concrete, let T be obtained by extending LA with finitely axiomatizable theories (which themselves may be obtained as combinations/extensions of others) whose signatures are not necessarily disjoint with that of LA. We describe a semi-decision procedure for the satisfiability problem of T based on an extension of the N&O method, where the superposition prover generates instances of the axioms of the theories and then sends such facts (hence, a superset of the elementary clauses) to the satisfiability procedure for LA. To make this approach practical, we briefly present two heuristics which we have found particularly useful to reduce the search space of the superposition prover: definition unfolding and reduction of large theories [19]. The second extension of the SMT problem we consider here is checking the satisfiability of first-order formulae containing quantifiers. We describe a technique which allows us to reduce the satisfiability problem of quantified formulae in a theory T to the satisfiability problem of quantifier-free formulae in an extension of the original theory T'.

An Extension of the Nelson-Oppen Combination schema

First of all, let us consider the situation of a theory T which does not contain LA and it is finitely axiomatized. Recall that we use a superposition prover to build satisfiability procedures for theories which can be finitely axiomatized. Since superposition can handle (at least in theory) any sets of clauses, we can use the SMT algorithm in Figure 1 (at least) as a semi-decision algorithm whenever we consider a theory T which is axiomatized by a finite set of formulae, which are easy to translate to Conjunctive Normal Form (CNF).

Now, let us consider the situation in which T is an extension of LA by a finite set Ax of axioms. There are two sub-cases two consider. First, Ax does not contain symbols of LA. In this case, T can be seen as the disjoint combination of LA and the theory axiomatized by Ax. If this last is stably-infinite then the N&O schema of Figure 2 is a satisfiability procedure for T; otherwise, it is only a semi-decision procedure. The second, more complex, case to consider is when the axioms of Ax contain symbols of LA. In this situation, we need to extend the schema of Figure 2, since exchanging elementary clauses is no more sufficient, even in simple situations. The key idea is to use the superposition prover as a mechanism to find ground instances of the quantified axioms which can be suitably used by the procedure for LA to detect unsatisfiability. As a consequence, the N&O schema must be modified in two ways: (i) the whole set of flat literals (i.e. also the arithmetic literals) is sent to the superposition prover and (ii) the superposition prover must send to the procedure for LA (and/or the module for case-splitting, cf. Figure 2) also the ground arithmetic facts which it has derived. We illustrate the technique by means of an example. Example Let us consider the theory T obtained as the extension of LA($\mathcal{R}$) with the following two axioms:

$$\forall U, V.(0 \leq U \quad \Rightarrow \quad \texttt{uir}(V, U) \leq U)) \tag{11}$$
$$\forall U, V.(0 \leq U \quad \Rightarrow \quad 0 \leq \texttt{uir}(V, U)) \tag{12}$$

where uir is a function which returns a random number constrained to be in the range satisfying the constraints above. Let us consider the problem of checking the T-satisfiability of the following formula in Disjunctive Normal Form (DNF):[8]

$$\begin{array}{ll}
0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq (5-1) \wedge 0 \not\leq 0 & \vee \\
0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq (5-1) \wedge 0 \not\leq \texttt{pv51} & \vee \\
0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq (5-1) \wedge \texttt{pv51} \not\leq (5-1) & \vee \\
0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq (5-1) \wedge 0 \not\leq \texttt{uir}(1, ((135300-1)-0)) & \vee \\
0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq (5-1) \wedge \texttt{uir}(1, ((135300-1)-0)) \not\leq (135300-1) &
\end{array} \tag{13}$$

[8]This proof obligation corresponds to `cl5_nebula_array_0020` in the benchmark set Array.

Indeed, if each disjunct of (13) is T-unsatisfiable then (13) is T-unsatisfiable. The first three disjuncts are obviously unsatisfiable: the first is LA($\mathcal{R}$)-unsatisfiable because of $0 \not\leq 0$, the second is Boolean-unsatisfiable because of $0 \leq \texttt{pv51}$ and $0 \not\leq \texttt{pv51}$, and the third is also Boolean-unsatisfiable because of $\texttt{pv51} \leq (5-1)$ and $\texttt{pv51} \not\leq (5-1)$. It is easy to see that (13) can be simplified to

$$\begin{array}{ll} 0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq 4 \wedge 0 \not\leq \texttt{uir}(1,135299) & \vee \\ 0 \leq \texttt{pv51} \wedge \texttt{pv51} \leq 4 \wedge \texttt{uir}(1,135299) \not\leq 135299 & \end{array}$$

while building its Boolean abstraction by evaluating ground terms. We are now left with the problem of checking the T-unsatisfiability of two conjunctions of literals which are not trivially unsatisfiable. Let us consider each case separately.

1. Each literal in the conjunction is turned into a unit clause and it is sent to the superposition prover together with the CNFs of (11) and (12), i.e.

$$0 \not\leq U \quad \vee \quad \texttt{uir}(V,U) \leq U \tag{14}$$

$$0 \not\leq U \quad \vee \quad 0 \leq \texttt{uir}(V,U) \tag{15}$$

   where U and V are implicitly universally quantified variables. Among many other facts, superposition derives (by resolution between (15) and the last literal) that $0 \not\leq 135299$, which is then sent to the satisfiability procedure for LA($\mathcal{R}$) so that unsatisfiability is immediately detected. Hence, we are entitled to conclude the T-unsatisfiability of this disjunct.

2. This case is similar to the previous. The only difference is that superposition derives $0 \not\leq 135299$ by resolution on (14) and the last literal.

Since the last two disjunct are T-unsatisfiable, we can conclude the T-unsatisfiability of (13).

Heuristics: definition unfolding and large theories

In order to make the extension of the N&O schema efficient, it is crucial to reduce the search space of the superposition prover, so that the least possible number of ground instances of the axioms in Ax should be considered. Even when the theory T does not contain LA, it is important to reduce the search space of the superposition prover in order to augment its predictability. A similar observation has already been done in [19] when using resolution-based theorem provers for software verification. We have found definition unfolding and the reduction of large theories particularly useful in this respect.

Definition unfolding.

Expanding definitions is a crucial concern for automated theorem proving. Definitions represent concepts in a theory and are an important structuring mechanism. For example, the subset relationship $\subseteq$ is defined in terms of set membership $\in$. How a theorem prover handles such definitions can have a significant effect on its performance. Resolution-based theorem provers are often very weak on problems involving definitions because all formulae must be preliminary translated to CNF. So, for example, it is not possible to replace $x \subseteq y$ with $\forall e.(e \in x \Rightarrow e \in y)$ in a clause since this would introduce a new quantifier and the quantifiers have already been eliminated in the translation to CNF. A further problem which frequently arises in software verification is given by predicates defined by case-analysis on the values of their arguments; a large number of clauses are created by the translation to CNF so that the search space of the prover is significantly larger.

To overcome these difficulties and to make the prover more predictable, we have adapted the technique of definition unfolding of [20]. Let ϕ be a formula to be checked for satisfiability in the theory T, obtained as the extension of LA by a set Ax of axioms which contain (non-recursive) predicate definitions. We replace all the occurrences of predicate applications by suitable instances of the body of the definition. Let ϕ' be the formula obtained by this transformation and Ax' be obtained from Ax by deleting all predicate definitions. It is easy to see that ϕ is satisfiable in T iff ϕ' is satisfiable in T', obtained by extending LA with the axioms in Ax'.

If the body of the definition contains a quantifier, then there are two cases to be considered. If the occurrence of the predicate application has a positive (resp., negative) polarity and the quantifier is existential (resp., universal), then it can be Skolemized by replacing its bound variables with Skolem constants. Hence, we obtain a ground formula on which the SMT algorithm of Figure 1 can be directly invoked. Otherwise, i.e. the occurrence of the predicate application has a negative (resp., positive) polarity and the quantifier is universal (resp., existential), then we have two choices: (i) do not expand the definition so to avoid introducing new quantifiers in the formula and hope that superposition will be capable of coping with it or (ii) expand the definition and further processing the formula to eliminate the newly introduced quantifier as explained below before invoking the SMT algorithm. We illustrate our technique with an example not involving quantifiers. Example Let us consider the theory T obtained by extending LA($\mathcal{R}$) with the following predicate definition:

$$\forall n.(\textit{IsInt0-199}(n) \Leftrightarrow n = 0 \vee n = 1 \vee \cdots \vee n = 198 \vee n = 199) \qquad (16)$$

expressing the fact that the number n is an integer in the range between 0 and 199. We want to check the unsatisfiability in T of the following formula:

$$\textit{IsInt0-199}(x) \wedge (\neg 0 \leq x \vee \neg x \leq 199). \qquad (17)$$

Now, if we do not perform definition unfolding, (16) is translated to CNF, thereby resulting in 201 clauses: 200 of the form $U \neq k \vee \textit{IsInt0-199}(U)$ for $k \in \{0, ..., 199\}$ and one of the form $\bigvee_{k=0}^{199} U = k \vee \neg \textit{IsInt0-199}(U)$ for U an implicitly universally quantified variable. By resolution between $\textit{IsInt0-199}(x)$ of (17) and the last clause above, we get the clause $\bigvee_{k=0}^{199} x = k$, which must be passed to the the case-splitting module (cf. Figure 2) so that each one of 200 cases is considered first with $\neg 0 \leq x$ and then again with $\neg x \leq 199$ (for a grand total of 400 case splits). So, all the burden is on the case-splitting module while the available satisfiability solver of the SMT algorithm (cf. Figure 1), which is much more suited to handle large formulae, remains idle. To avoid this ineffective use of the resources, it is better to expand the definition (16) in (17) so to obtain the following formula:

$$(x = 0 \vee x = 1 \vee \cdots \vee x = 198 \vee x = 199) \wedge (\neg 0 \leq x \vee \neg x \leq 199) \tag{18}$$

which must be checked for unsatisfiability in LA($\mathcal{R}$). In this way, the 400 case splits are handle by the available satisfiability solver and only the satisfiability procedure for LA($\mathcal{R}$) can be used, thereby avoiding the overhead of invoking the superposition prover to derive suitable instances of the definition.

Reduction of large theories.

It is not unusual that software specifications comprise hundreds of axioms and the proofs of the unsatisfiability of conjectures are usually shallow and consist of analyzing a large number of cases. Resolution-based theorem provers are known to have quite impressive performances on problems consisting of few axioms and conjectures (usually extracted from mathematical problems) whose proofs are quite deep and contain simple case analyses. Hence, it is not surprising that resolution-based theorem provers do not perform well in the context of software verification (see e.g., [21] for an in-depth discussion on this issue). The problem is that provers are lost in the search space although the majority of the axioms are irrelevant to the proof under consideration.

In [19], a technique to find out an approximation of the relevant axiom set for the proof of a conjecture is presented. We use an adaptation of such a technique as a pre-processing step of the SMT algorithm of Figure 1 by introducing suitable syntactic constructs to structure the set Ax of axioms. The axiomatization is structured into *theories*. A theory T_i defines the semantics of a set of symbols S_i by means of a set of axioms Ax_i, and possibly the use of previously defined theories via an explicit importation clause.

The idea underlying the reduction algorithm is to use a directed acyclic graph whose nodes are associated to the theories; an edge from the node n_1 to the node n_2 represents the fact that the theory T_1 associated to n_1 is a superset of the theory T_2 associated to n_2, i.e. S_1 is obtained by extending S_2. So, given a formula ϕ, it is sufficient to compute the set of symbols occurring in ϕ, find which the set N_ϕ of nodes with theories containing the symbols of ϕ and to form the relevant axiom set by transitive traversal of the nodes reachable from N_ϕ. Finally, in order to incorporate the treatment of arithmetic in this framework, which is not finitely axiomatized in our case, our algorithm considers an implicit theory T_a, such that Ax_a is empty and S_a are the arithmetic symbols, and adds an implicit edge to N_a from all nodes representing theories where these symbols appear. For more details and an extensive discussion, the interested reader is referred to [19].

We evaluate the impact of these two techniques on a set of benchmarks.

```
function grounding ($\phi$: quantified formula)
    $\phi_0 \longleftarrow$ existential_closure($\phi$)
    $\phi_1 \longleftarrow$ minimize_scope($\phi_0$)
    $\phi_2 \longleftarrow$ drop_existentials($\phi_1$)
    $(\phi_3, \Delta) \longleftarrow$ rename_and_define($\phi_2$)
    return $(\phi_3, \Delta)$
end
```

Figure 3: Handling Quantifiers

Handling quantified formulae

So far, we have considered the problem of checking the (un-)satisfiability of ground formulae in a given theory. In many software verification scenarios, considering only ground formulae is too restrictive. The crux to lift the SMT algorithm of Figure 1 to handle quantified formulae is again the usage of a superposition prover to implement *T-satisfiable*.

For simplicity, here we consider a theory T axiomatized by the a set of clauses Ax. However, the technique can be straightforwardly adapted to consider LA($\mathcal{R}$). Let ϕ be a first-order formula (containing quantifiers), the idea is to transform ϕ into a ground formula ϕ_g by replacing the quantified sub-formulae of ϕ with fresh propositional letters and to add the definitions Δ of these to the axioms Ax in such a way that $Ax \wedge \phi$ is satisfiable iff $Ax \wedge \Delta \wedge \phi_g$ is. The algorithm for pre-processing is given in Figure 3. Let ϕ be a first-order formula and $v_1, ..., v_n$ its free variables. The formula returned by *existential_closure*(ϕ) is $\exists v_1, ..., v_n.\phi$. It is easy to see that ϕ is satisfiable iff *existential_closure*(ϕ) is.

Since we want to add a set Δ of "small" formulae to Ax and to preserve as much as possible the Boolean structure of the formula (so to maximally exploit the Boolean solver of the SMT algorithm in Figure 1), *minimize_scope* implements rules to move quantifiers as far inwards as possible. Roughly, we use all the rules to transform a formula into prenex form[9] but in the opposite direction [20]. For example, the formula $\forall x.(\phi \wedge \psi)$ is transformed to $(\forall x.\phi) \wedge \psi$ if the variable x does not occur in ψ and the formula $\exists x.(\phi \Rightarrow \psi)$ to $(\forall x.\phi) \Rightarrow \psi$, again if x does not occur in ψ. These transformations are equivalence preserving and terminate as they always reduce the depth of a formula starting with a quantifier. In practice, these classic (anti)prenexing rules do not always yield an optimal result, as they take into account neither the associativity and commutativity of conjunction and disjunction, nor the properties of multi-variables quantifications. Consider indeed the following :

$$\begin{aligned}\forall x, y \bullet [p_1(x) \wedge p_2(x, y) \wedge p_3] &= \forall x \bullet [\forall y \bullet [p_1(x) \wedge (p_2(x, y) \wedge p_3)]],\\ &= \forall x \bullet [p_1(x) \wedge (\forall y \bullet [p_2(x, y) \wedge p_3])],\end{aligned}$$

[9]A formula is in prenex form if it has the following structure $Q_1 x_1 ... Q_n x_n.\phi$, where Q_i is either $\forall$ or $\exists$, x_i is a variable ($i = 1, ..., n$), and ϕ is a quantifier-free formula whose free variables are $x_1, ..., x_n$.

$$\begin{aligned} &= \forall x \bullet [p_1(x) \wedge (\forall y \bullet [p_2(x,y)] \wedge p_3)], \\ &= \forall x \bullet [p_1(x) \wedge ((\forall y \bullet [p_2(x,y)]) \wedge p_3)]. \end{aligned}$$

At this point, no simplification rule can be applied, as x is a free variable in both operands of the conjunction under the scope of the outermost quantification, which still applies to p_3. However, there is a better solution:

$$\forall x, y \bullet [p_1(x) \wedge p_2(x,y) \wedge p_3] \quad = \quad p_3 \wedge \forall x \bullet [p_1(x) \wedge \forall y \bullet [p_2(x,y)]].$$

The next example illustrates the influence of the variable order in the quantification:

$$\forall x, y, z \bullet [p_1(x,z) \wedge p_2(y,z)] \quad = \quad \forall x \bullet [y \bullet [z \bullet [p_1(x,z) \wedge p_2(y,z)]]].$$

Here, no simplication rule can be applied. However, there is a better solution:

$$\forall x, y, z \bullet [p_1(x,z) \wedge p_2(y,z)] \quad = \quad \forall z \bullet [\forall x \bullet [p_1(x,z)] \wedge \forall y \bullet [p_2(y,z)]].$$

The rules implemented in the *minimize_scope* routine are aware of aforementioned properties and provides a more aggressive minimization of quantifier scope than the classical rules [20].

We are now ready to start the elimination of quantifiers. We begin by eliminating the existential quantifiers by a restricted form of Skolemization. Invoking *drop_existentials* on a first-order formula ϕ returns the formula obtained by repeatedly eliminating existential quantifiers in an outermost way. More precisely, *drop_existentials* exhaustively applies the following transformations: $\phi[\exists x.\psi]_p$ ($\phi[\forall x.\psi]_p$) is rewritten to $\phi[\psi[x/c]]_p$, where c is a fresh constant, p is the outermost position at which a positive (negative, resp.) occurrence of an existentially (universally, resp.) quantified sub-formula is in ϕ, and $\exists x.\psi$ ($\forall x.\psi$, resp.) does not contain free variables. This process terminates since each application of the rules removes an existential or a universal quantifier. We can eliminate the remaining quantifiers by substituting each quantified sub-formula ψ with a fresh propositional letter q and recording its definition $q \Leftrightarrow \psi$. The function *rename_and_define* performs this with some optimizations. More precisely, invoking *rename_and_define* on a first-order formula ϕ returns the pair (ϕ', Δ) which is obtained by exhaustively applying the following transformations: $\phi[\psi]_p$ is rewritten to $\phi[q]_p$, where q is a fresh propositional letter, p is the outermost position at which a quantified sub-formula occurrence is in ϕ, and ψ does not contain free variables. Furthermore, we add $q \Rightarrow \psi$ ($\psi \Rightarrow q$) to Δ if ψ is a positive (negative, resp.) sub-formula occurrence of ϕ. Since there are only finitely many (quantified) sub-formulae, this process obviously terminates. It is not difficult to prove that the formula ϕ is satisfiable iff $\phi_g \wedge \Delta$ is, where $(\phi_g, \Delta) =$ *rename_and_define*(ϕ) (see [9] for a formal development).

The formula ϕ_g and the theory axiomatized by $Ax \cup \Delta$ can be sent to the SMT algorithm of Figure 1.

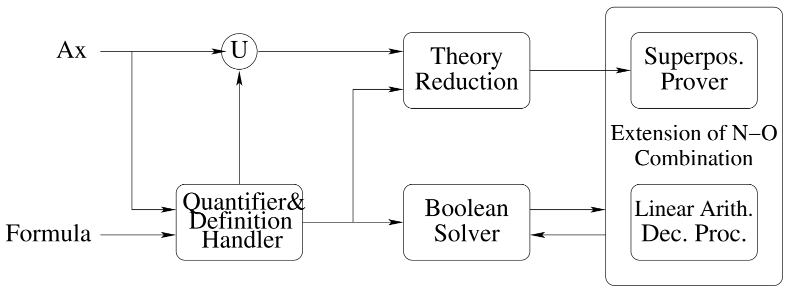


Figure 4: The Architecture of **haRVey**

IMPLEMENTATION AND EXPERIMENTS

We have implemented the techniques described above in a system called **haRVey**.[10] It makes use of several existing tools: Flotter[11] to transform the axioms of the background theory to CNF, D. Long's BDD library[12] or zChaff for Boolean reasoning, the E prover[13] for T-*satisfiable*, and the ATerm library[14] for transforming formulae and axioms as well as a basis for the communication between the various tools. **haRVey** accepts as input the (possibly structured) axiomatization extending LA($\mathcal{R}$) and the first-order formula to be proved (un-)satisfiable in a LISP-like syntax. A high-level view of **haRVey** is depicted in Figure 4. The set Ax of axioms is searched for definitions which are expanded in the input formula (related heuristics are presented below), quantified subformulae are eliminated, and the set of axioms suitably modified (as explained below). At this point, the theory is reduced and passed to the superposition prover. Finally, the ground formula obtained after unfolding and "grounding" (cf. Figure 3) is passed to the SMT algorithm of Figure 1. If arithmetic reasoning is required, the superposition prover may be combined with a satisfiability procedure for LA by using the extension of the N&O schema described in this paper.

For benchmarks, we have considered the proof obligations generated by the certification of auto-generated aerospace software [22]. For certification, the goal is not to ensure full correctness but to check that a program satisfy a certain level of safety. A typical security problem is that a program does not access out-of-bound elements in an array. Using a Hoare logic approach, one can automatically add annotations to a program, generate the corresponding proof obligations, and then discharge them. In [22], five safety properties are considered on four auto-generated aerospace programs written in C (ranging from around 400 to more than 1000 lines of code): 366 valid and 2 invalid proof obligations are obtained. The mean value of the cardinality of the set Ax of

[10]`http://www.loria.fr/equipes/cassis/softwares/haRVey`
[11]`http://spass.mpi-sb.mpg.de`
[12]`http://www-2.cs.cmu.edu/~modelcheck/bdd.html`
[13]`http://www4.informatik.tu-muenchen.de/schulz/WORK/eprover.html`
[14]`http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary`

Table 1: Experimental Results.

	haRVey (%)	NASA (%)
Array	100	96.4
Init	94.4	76.8
In-use	78.9	68.4
Symm	100	50.0
Norm	70.8	51.8

axioms is 79.

The experiments have been carried out on a Pentium IV 2Ghz running Linux with 256 Mb of RAM and 1Gb of disk space. We have set a time-out of 30 seconds per proof obligation. The results are depicted in Table 1. The first column lists the five safety properties used to generate the proof obligations. The second (third) column records the percentage of proof obligations successfully discharged by **haRVey** (the system developed by NASA and described in [22], respectively).[15] In the cases of Array and Symm, we successfully discharge all proof obligations, for Init our system scores definitely better than NASA, while for In-use and Norm **haRVey** is still better but not as good as we would like. The problem with In-use is that corresponding proof obligations are large and the system frequently times out. For Norm, the problem is that only a partial axiomatization of the operator `sum`, which takes a vector and returns the sum of its values, is supplied. This is so because it is not possible to have a complete axiomatization for this operator in first-order logic as explained in [22].

To evaluate the impact of the proposed heuristics, we have repeated our experiments in the following configurations of the system. First, we have disabled theory reduction: we have drastically reduced our success rates of 60%. Second, we have prevented the expansions of definitions: we have reduced our success rates of only 3%. Finally, to evaluate the effectiveness of our integration of LA in the system, we have eliminated all the axioms regarding LA from the benchmarks in Array and we have re-rerun the system using our extension of the N&O schema on the resulting proof obligations. **haRVey** is still capable of discharging all modified proof obligations.

We believe that these results are encouraging and confirm the viability of our approach for effective automated software verification tasks such as those arising in the engineering of areospatial systems. Other successful applications of the techniques described in this paper can be found in [9, 23, 14].

References

[1] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN 2001*, volume 2057 of *LNCS*, pages 103–

[15]We use the results obtained on the 366 proof obligations and not those obtained after applying simplifications. See again [22] for details.

122, 2001.

[2] R.S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[3] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning.* 2001.

[4] M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, P.v.Rossum, S.Schulz, and R.Sebastiani. The mathsat 3 system. In *Conference on Automated Deduction (CADE-20)*, 2005.

[5] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'04)*, LNCS. Springer, 2004.

[6] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, Massimo Idini, and Marco Maratea. TSAT++: an Open Platform for Satisfiability Modulo Theories. In *Proc. 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'04)*, 2004.

[7] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.

[8] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th Intl. Conf. Computer Aided Verification (CAV 2004)*, 2004.

[9] D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In IEEE Comp. Soc. Press, editor, *Proc. of the Int. Conf. on Software Engineering and Formal Methods (SEFM03)*, 2003.

[10] A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Info. and Comp.*, 183(2):140–164, June 2003.

[11] M. Rusinowitch. Theorem-proving with Resolution and Superposition. *JSC*, 11(1&2):21–50, January/February 1991.

[12] Stephan Schulz. E – a brainiac theorem prover. *AI Communications*, 2002. See also the web page `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`.

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

[14] D. Déharbe, A. Imine, and S. Ranise. Abstraction–Driven Verification of Array Programs. In *In Proc. of AISC'04*, LNCS, 2004.

[15] S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In *In Proc. of ICTAC'04*, LNCS, 2004. Available at `http://www.loria.fr/~ranise/pubs/long-ictac04.ps.gz`.

[16] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Lab., 2003.

[17] H. Kirchner, S. Ranise, C. Ringeissen, and D. K. Tran. On Superposition-Based Satisfiability Procedures and their Combination. Submitted, available from `http://www.loria.fr/~ranise`.

[18] P. van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Ann. of Math. and Art. Intell.*, 5:303–319, 1992.

[19] W. Reif and G. Schellhorn. *Automated Deduction—A Basis for Applications*, volume 1, chapter Theorem Proving in Large Theories. Kluwer Academic Pub., 1998.

[20] Andreas Nonnengart and Christoph Weidenbach. *Handbook of Automated Reasoning*, chapter Computing Small Clause Normal Forms. Elsevier Science, 2001.

[21] J. Schumann. Automated Theorem Proving in High-Quality Software Design. *Intelletics and Computational Logic*, 19, 2000. Applied Logic Series.

[22] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In *Proc. of Int. Joint Conf. On Automated Reasoning (IJCAR'04)*, volume 3097 of *LNCS*, 2004.

[23] Jean-Franois Couchot, Frédéric Dadeau, David Déharbe, Alain Giorgetti, and Silvio Ranise. Proving and debugging set-based specifications. *Electronic Notes in Theoretical Computer Science*, 95:189–208, 2004. Proc. of the 6th Workshop on Formal Methods.