Don't care in SMT—Building flexible yet efficient abstraction/refinement solvers

Andreas Bauer[*], Martin Leucker[**], Christian Schallhart[**], Michael Tautschnig[**]

[*]Computer Sciences Laboratory, Australian National University
[**] Institut für Informatik, Technische Universität München, Germany

Abstract. This paper describes a method for combining "off-the-shelf" SAT and constraint solvers for building an efficient *Satisfiability Modulo Theories* (SMT) solver for a wide range of theories. Our method follows the abstraction/refinement approach to simplify the implementation of custom SMT solvers. The expected performance penalty by *not* using an interweaved combination of SAT and theory solvers is reduced by *generalising* a Boolean solution of an SMT problem first via assigning *don't care* to as many variables as possible. We then use the generalised solution to determine a thereby smaller constraint set to be handed over to the constraint solver for a background theory. We show that for many benchmarks and real-world problems, this optimisation results in considerably smaller and less complex constraint problems.

The presented approach is particularly useful for assembling a practically viable SMT solver quickly, when neither a suitable SMT solver nor a corresponding incremental theory solver is available. We have implemented our approach in the ABSOLVER framework and applied the resulting solver successfully to an industrial case-study: The verification problems arising in verifying an electronic car steering control system impose non-linear arithmetic constraints, which do not fall into the domain of any other available solver.

1 Introduction

Satisfiability modulo theories (SMT) is the problem of deciding whether a formula in quantifier-free first-order logic is satisfiable with respect to a given *background* theory. For example, one is interested whether the formula $\phi \equiv (i \geq 0) \wedge (\neg(2i + j < 10) \vee (i + j < 5))$ is satisfiable in the theory of integers. In recent years, research on SMT has attracted a lot of attention. SMT solvers for dedicated theories have been developed, such as Yices [21], MathSAT [5], or CVC [1]. The growing efficiency of these solvers in their respective domains is witnessed in the annual SMT competition (`http://www.smtcomp.org`).

Amongst others, SMT has its applications in the area of model checking and abstraction [12], (symbolic) test case generation [19], or in the verification of hybrid control systems [3, 20], to name just a few common examples. Especially for the latter, however, one is often faced with the task of having to solve problems with respect to theories that are not (yet) supported by existing SMT solvers, although *constraint solvers* for the required theories are available. These powerful constraint solvers have been developed for dedicated theories, such as general linear arithmetic over integer and real numbers [24]. In contrast to SMT solvers, such constraint solvers only accept a conjunction rather than an arbitrary Boolean combination of atoms.

In this paper, we propose a method for combining off-the-shelf Boolean satisfiability (SAT) and constraint solvers without altering them to assemble SMT solvers for a wide range of different theories with a minimal engineering overhead, yet with a reasonable practical performance. The existing approaches to solve SMT problems can be subdivided into three main categories. In the *translation approach* [23], given an SMT instance, the entire problem is encoded as an equi-satisfiable pure SAT instance such that a solution to the SAT problem translates into a solution of the original SMT instance. For example, if the above mentioned ϕ is solved over the 16 bit integers, then it is straightforward to formulate ϕ's constraints in terms of bits yielding a purely propositional formula. With the advent of highly efficient SAT solvers (cf. [8, 15, 16]) this approach turned out quite successful—at least for certain background theories, see for example [10, 18]. However, such a translation involves a non-obvious interplay between the SAT solver and the encoding, where the structure of the underlying problem is difficult to reflect in the encoding. In the *abstraction/refinement approach* [22], one represents each occurring theory constraint with a Boolean variable. By substituting these Boolean variables for their respective constraints, an abstract SAT problem is produced and solved first. This determines the set of constraints to be satisfied. If such a Boolean *representative variable* has been set to true, then the corresponding constraint is selected, and respectively, if a Boolean representative variable has been assigned false, then the negation of the corresponding constraint is added to the constraint set. Finally, this constraint set is passed on to a dedicated solver for the background theory of the problem. If the solver finds a solution, then the original SMT problem has been solved, and a solution has been determined. On the other hand, if the theory solver fails, then the Boolean abstraction is refined, a new solution for the abstract SAT instance is computed and the process continues. In the *online solving approach* [9], both the abstract Boolean problem and the theory constraints are solved simultaneously, i. e., whenever a Boolean variable which represents a constraint is assigned, the corresponding constraint or its negation is added to the set of constraints to be satisfied. This set is checked for satisfiability immediately and consequently conflicts can be detected at an early stage of the search process and can be pruned from the remaining search space. This approach allows for building highly efficient SMT solvers and is followed by most modern tools. However, it requires a tight interaction between the SAT solver and the constraint solver: the SAT solver must call the constraint solver whenever a new constraint is added and therefore, the solver should be able to handle this growing constraint set efficiently. Furthermore, when the SAT solver backtracks, the constraint solver must follow the backtracking step, and remove the corresponding constraints from the incrementally growing set. Such a tight interaction complicates the integration of existing constraint solvers since they need an interface supporting backtracking, similar to the one described in [9]. Thus, when building custom SMT solvers using off-the-shelf constraint solvers that do not support backtracking, this approach is often impractical, especially in presence of limited development resources.

Foremost for this reason, our framework, ABSOLVER [3], which allows the integration of efficient SAT and constraint solvers to build-up custom SMT solvers, follows the abstraction/refinement approach. As this method proved to be inferior to the online solving approach, we employ a simple yet surprisingly efficacious optimisation to the abstraction/refinement scheme: once a SAT solver has determined a solution to the Boolean abstraction of an SMT problem, we first *generalise* this solution, before generating and solving the underlying constraint problem. This yields fewer and smaller constraint problems than the traditional approach. More specifically, we use a simple greedy-algorithm to find a minimal assignment (but not necessarily of minimal weight) which still satisfies the Boolean abstraction, i. e., each completion of the assignment must still satisfy the Boolean abstraction. Having found such a partial assignment, each variable is assigned either true, false, or *don't care*. For each representative variable being assigned true, we add the corresponding constraint to the constraint set. Respectively, for each representative variable being assigned false, we add the negation of the constraint. All other representative variables, i. e., all variables being assigned *don't care*, are ignored. Thus, the smaller the assignment, the smaller the constraint set to be handed to the corresponding constraint solver. Furthermore, if such a smaller assignment is found to be conflicting by the theory solvers, a set of possible Boolean solutions is invalidated by a single assignment. The size of this set is exponential in the number of *don't care*s.

Our generalisation of a SAT solver's solution is based on the efficient computation of a *minimal* solution of a given conjunctive normal form (CNF) formula. Our approach is thus similar in spirit to the so-called MINSAT problem and its variations [4, 6, 11], which, however, are known to be NP-complete [6]. These complexity theoretic results imply that we cannot hope to find any generally efficient algorithm and therefore, we need to resort to heuristic approaches which (as our benchmarks in this paper indicate) work well in most practically relevant cases.

We have implemented the suggested optimisation within our ABSOLVER framework. Even though we have to admit that our approach does not reach the performance of other participants of the SMT-COMP in their respective domains, our solver has been successfully applied to an industrial case-study involving non-linear constraints which are not supported by other solvers (see Sec. 4). Using ABSOLVER, we were able to verify properties of a car's electronic steering control system whose behaviour was given by a MATLAB/Simulink model. Such models typically capture the dynamics of the closed control loop, involving the actual system and part of its environment. This loop can then often, as it was in our case, only be expressed in terms of a non-linear equation system.

2 Abstraction and refinement for SMT

In this section, we develop the framework in which we describe our approach. Since we are faced with formulas which involve variables ranging over different domains, we use a *typed* setting.

Domains, variables, assignments. Let Σ be a finite set of *types* and $\mathcal{D} = (\mathbb{D}_\sigma)_{(\sigma\in\Sigma)}$ a family of respective *domains*. Furthermore, let $\mathcal{V} = (V_\sigma)_{(\sigma\in\Sigma)}$ be a family of finite sets of *variables* of the respective type. Abusing notation, we also denote by $\mathcal{D}$ the union $\bigcup_{\sigma\in\Sigma}\mathbb{D}_\sigma$ and by $\mathcal{V}$ the union $\bigcup_{\sigma\in\Sigma} V_\sigma$. We also call the elements of $\mathcal{D}$ *values*.

$\mathbb{B}$ denotes the *Boolean* type as well as the domain $\mathbb{B} = \{tt, ff\}$. We always assume $\mathbb{B} \in \Sigma$ and we mostly consider the reals $\mathbb{R}$ and integers $\mathbb{Z}$ as additional types.

To represent partial assignments with total mappings, we introduce $?$ to denote the *don't care* value and let $\mathcal{D}^? = \{?\} \uplus (\mathbb{D}_\sigma)_{(\sigma\in\Sigma)}$ be the family of domains enriched with *don't care*.

An *assignment* is a mapping $\tau : \mathcal{V} \to \mathcal{D}^?$ assigning to all variables either a value of the corresponding domain or ?. We call τ *complete*, iff $\tau(v) \neq\ ?$ for all $v \in \mathcal{V}$. To establish an *information preorder*, we set $? \prec d$ for all $d \in \mathcal{D}$, ordering ? below all domain values and leaving these values unordered. Let $\preceq$ denote the reflexive closure of $\prec$. The information preorder extends to assignments by $\tau \preceq \tau'$ iff for all $v \in \mathcal{V} \quad \tau(v) \preceq \tau'(v)$. Thus, τ is smaller than τ' w. r. t. $\prec$, if reassigning ? to a number of variables in τ' results in τ.

The *weight* $|\tau|$ of an assignment τ is the number of values different from ?, i.e., $|\tau| = |\{\tau(v) \neq\ ? \mid v \in \mathcal{V}\}|$. Dually, we define the *freedom* of τ, denoted by $|\tau|_?$, as the number of *don't care*s in its range: $|\tau|_? = |\{\tau(v) =\ ? \mid v \in \mathcal{V}\}|$.

The set of assignments *generated* by τ, denoted by $\langle\tau\rangle$, is given by a set of assignments τ' with $\tau \preceq \tau'$. Similarly, the set of complete assignments generated by τ, denoted by $\overline{\langle\tau\rangle}$, is given by the set of complete assignments τ' with $\tau \preceq \tau'$.

Remark 1. *The number of complete assignments generated by an assignment τ is exponential in its freedom: $|\overline{\langle\tau\rangle}| = 2^{|\tau|_?}$.*

Formulas. Let $\mathcal{F} = (F_\sigma)_{\sigma\in\Sigma}$ be a family of ranked function symbols and $\mathcal{P} = (P_\sigma)_{\sigma\in\Sigma}$ a family of ranked predicate symbols. The set of (typed) *terms* is inductively defined: First, every variable of $\mathcal{V}_\sigma$ is a term of type σ, and second, if $f \in F_\sigma$ of rank n is a function symbol of type σ and $a_1, \ldots, a_n$ are terms of type σ, then $f(a_1, \ldots, a_n)$ is a term of type σ.

The set of (typed) *atoms* is defined as follows: If $p \in P_\sigma$ of rank n is a predicate symbol of type σ and $a_1, \ldots, a_n$ are terms of type σ, then $p(a_1, \ldots, a_n)$ is an atom of type σ. Note that the above definition does not allow terms and atoms which involve two or more types. Each such atom represents a *constraint* formulated in the background theory of the respective type.

A *literal* is a possibly negated atom, a *clause* is a disjunction of literals, and a formula in *conjunctive normal form* (CNF) is a conjunction of clauses. Thus, a formula ϕ in CNF, as considered subsequently, has the form $\phi \equiv \bigwedge_{i\in I} \bigvee_{j\in J_i} (\neg)p_{ij}(a_1, \ldots, a_{n_{ij}})$.

Finally, for a formula ϕ, we use $\mathcal{V}_\sigma(\phi)$ to denote the variables of type σ occurring in ϕ.

Example 1. *As a running example, we use the following formula ϕ consisting of four clauses over the variables $\mathcal{V}_\mathbb{Z}(\phi) = \{i, j, k, l\}$ and $\mathcal{V}_\mathbb{B}(\phi) = \{x, y\}$:*

$$\phi \equiv \{(i \geq 0) \vee y\} \wedge \{\neg(2i + j < 10) \vee (i + j < 5)\} \wedge \{x \vee \neg(j \geq 0)\} \wedge \{(k + (4 - k) + 2l \geq 7)\}$$

Solutions. A *complete solution* of ϕ is a complete assignment to the variables in $\mathcal{V}$, such that ϕ evaluates to *tt* in the usual sense. For example, we can define τ as an assignment for ϕ (as shown in Ex. 1) with $\tau(i) = 3$, $\tau(j) = 1$, $\tau(k) = 0$, $\tau(l) = 2$, $\tau(x) = tt$, and $\tau(y) = ff$. This assignment *satisfies* all clauses and assigns values other than ? to all variables. It is therefore called a *complete solution* of ϕ. For a given formula ϕ, the *SMT problem* is to decide whether there is a complete solution for ϕ.

In general, an assignment τ is a *solution* of ϕ iff every complete assignment τ' with $\tau \preceq \tau'$ (i.e. every $\tau' \in \overline{\langle\tau\rangle}$) is a solution of ϕ. For example, an assignment τ with $\tau(i) = 3$, $\tau(j) = 1$, $\tau(k) =\ ?$, $\tau(l) = 2$, $\tau(x) = tt$, and $\tau(y) = ff$ is also a solution for formula ϕ of Ex. 1 since the value of k can be set arbitrarily.

The assignment τ is called a *minimal solution* iff τ is a solution of ϕ and minimal w. r. t. $\preceq$: Thus, if any further variable in τ is assigned ?, then there would be a τ' with $\tau \preceq \tau'$ which does not satisfy ϕ. A solution τ is a solution of *minimal weight* iff it is a solution and for all solutions τ' we have $|\tau| \leq |\tau'|$.

For example, the τ above is not minimal, since τ' with $\tau' \preceq \tau$ by setting $\tau'(i) = 3$, $\tau'(j) = 1$, and $\tau'(l) = 2$ and assigning ? to all remaining variables is also a solution of ϕ. τ' is not only a minimal but also a solution of minimal weight for ϕ since every solution for ϕ must at least assign values to i, j, and l to satisfy the second and the fourth clause, respectively.

2.1 Deciding SMT by abstraction and concretisation

We integrate a Boolean SAT solver as well as constraint solvers for the occurring background theories into a combined SMT solver. Thereby, we require the constraint solvers to decide the satisfiability of conjunctions of possibly negated constraints. Thus, our goal is to reduce the SMT problem to Boolean SAT problems and constraint solving problems. We follow the well-known idea of solving first a Boolean abstraction of ϕ yielding a constraint problem for each type at hand.

Boolean abstraction. Given a formula ϕ in CNF, its *Boolean abstraction* $\mathsf{abst}(\phi)$ is defined as follows: Every atom $p_{ij}(a_1, \ldots, a_{n_{ij}})$ is replaced by a new *representative* Boolean variable p_{ij} which does not occur otherwise in ϕ. Thus, $\psi := \mathsf{abst}(\phi)$ is of the form $\psi \equiv \bigwedge_{i \in I} \bigvee_{j \in J_i} (\neg) p_{ij}$. The representative Boolean variables of a Boolean abstraction $\mathsf{abst}(\phi)$ are denoted by the set $\mathcal{V}_{\mathbb{B}}^R(\mathsf{abst}(\phi)) \subseteq \mathcal{V}_{\mathbb{B}}(\mathsf{abst}(\phi))$. Since all representative variables do not occur otherwise in ϕ, we have $\mathcal{V}_{\mathbb{B}}^R(\mathsf{abst}(\phi)) \cap \mathcal{V}(\phi) = \emptyset$.

Example 2. *The Boolean abstraction of ϕ shown in Ex. 1 is given as* $\mathsf{abst}(\phi) \equiv \{v_1 \vee y\} \wedge \{\neg v_2 \vee v_3\} \wedge \{x \vee \neg v_4\} \wedge \{v_5\}$ *with* $\mathcal{V}_{\mathbb{B}}^R(\mathsf{abst}(\phi)) = \{v_1, \ldots, v_5\}$. *Here, we use v_1 as a representative Boolean variable for the atom $(i \geq 0)$, and v_2 as representative $(2i + j < 10)$, and so forth.*

Abstract solutions. Let ϕ be a formula and $\psi := \mathsf{abst}(\phi)$ its Boolean abstraction. Every complete assignment to the variables of ϕ yields a truth value for the atoms of ϕ. As the atoms are mapped to Boolean variables in ψ, this yields a complete assignment for the variables of ψ. More formally, every assignment τ to the variables in ϕ induces an assignment $\nu := \mathsf{abst}(\tau)$ to the Boolean variables in ψ by $\nu(p_{ij}) := (p_{ij}(a_1, \ldots, a_{n_{ij}}))[\tau]$ where $(p_{ij}(a_1, \ldots, a_{n_{ij}}))[\tau]$ denotes the truth value of the atom $p_{ij}(a_1, \ldots, a_{n_{ij}})$ under assignment τ (if some a_i is assigned ?, then p_{ij} is assigned ? as well). We have immediately:

Remark 2. *Let τ be a (complete) solution of ϕ. Then $\mathsf{abst}(\tau)$ is a (complete) solution of* $\mathsf{abst}(\phi)$.

Concretisation. Let $\mathsf{conc}(\phi, \nu) := \{\tau : \mathcal{V}(\phi) \to \mathcal{D}^? \mid \mathsf{abst}(\tau) = \nu\}$ be the set of all *concretisations* of ν with respect to ϕ. As a consequence of Remark 2, the satisfiability of ϕ can be checked by first searching for a complete solution ν of $\mathsf{abst}(\phi)$ and then checking whether there is a $\tau \in \mathsf{conc}(\phi, \nu)$ which satisfies ϕ. While the first problem is an ordinary Boolean SAT problem, the second problem is a constraint problem, i.e., one has to check whether $\mathsf{constr}(\phi, \nu) \equiv \bigwedge_{\nu(p_{ij}) = \mathit{tt}} p_{ij}(a_1, \ldots, a_{n_{ij}}) \wedge \bigwedge_{\nu(p_{ij}) = \mathit{ff}} \neg p_{ij}(a_1, \ldots, a_{n_{ij}})$ is satisfiable. This suggests the abstraction/refinement approach for checking satisfiability of ϕ, i. e., to search for an abstract complete solution ν for $\mathsf{abst}(\phi)$ and to then search for a complete solution for $\mathsf{constr}(\phi, \nu)$. We summarise this procedure in the following lemma:

Lemma 1. *ϕ is satisfiable iff there is a complete solution ν of* $\mathsf{abst}(\phi)$ *and* $\mathsf{constr}(\phi, \nu)$ *is satisfiable.*

Note that the application of this lemma requires each invoked constraint solver to be able to handle negated atoms.

2.2 Generalisation

We adapt the approach in order to reduce the number of calls to the constraint solvers and such that the individually processed constraint sets involve fewer constraints—ultimately yielding a much better overall performance.

The simple yet efficacious idea is to *generalise* a given solution obtained by a SAT solver before considering the constraint problem. Given a complete solution ν for $\mathsf{abst}(\phi)$, we will obtain a minimal solution $\nu' \preceq \nu$ and replace ν with ν' in all subsequent steps.

For a not necessarily complete solution ν', the constraint set $\mathsf{constr}(\phi, \nu')$ is exactly defined as for a complete solution. Note, however, all constrains $p_{ij}(a_1, \ldots, a_{n_{ij}})$ with $\nu'(p_{ij}) = ?$ are not part of $\mathsf{constr}(\phi, \nu')$. In other words, $\mathsf{constr}(\phi, \nu')$ has $|\nu'|_?$ less atoms than $\mathsf{constr}(\phi, \nu)$ for a complete solution ν. But still, the statement of Lemma 1 holds for incomplete solutions:

Lemma 2. *ϕ is satisfiable iff there is a (possibly incomplete) solution ν' of* $\mathsf{abst}(\phi)$ *and* $\mathsf{constr}(\phi, \nu')$ *is satisfiable.*

Proof. Consider a solution τ' of $\mathsf{constr}(\phi, \nu')$. If τ' is not complete, take an arbitrary complete solution τ with $\tau' \preceq \tau$. Then we have $(p_{ij}(a_1, \ldots, a_{n_{ij}}))[\tau] = \nu'(p_{ij})$ whenever $\nu'(p_{ij}) \neq ?$, i. e., $\nu' \preceq \mathsf{abst}(\tau)$. Since ν' satisfies $\mathsf{abst}(\phi)$, $\mathsf{abst}(\tau)$ satisfies $\mathsf{abst}(\phi)$ as well and thus τ satisfies ϕ. The other direction is immediate by Lemma 1. $\square$

The next lemma shows that we can resort to incomplete solutions to prune the search space:

Lemma 3. *Let ν and ν' be solutions of* $\mathsf{abst}(\phi)$ *with $\nu' \preceq \nu$. Then satisfiability of* $\mathsf{constr}(\phi, \nu)$ *implies satisfiability of* $\mathsf{constr}(\phi, \nu')$*.*

Proof. Since $\mathsf{constr}(\phi, \nu')$ contains a subset of the constraints of $\mathsf{constr}(\phi, \nu)$, every assignment τ which satisfies $\mathsf{constr}(\phi, \nu)$ must satisfy $\mathsf{constr}(\phi, \nu')$ as well. $\square$

Therefore if ν' is a solution of $\mathsf{abst}(\phi)$ and $\mathsf{constr}(\phi, \nu')$ is *not* satisfiable, then $\mathsf{constr}(\phi, \nu)$ is not satisfiable for all ν with $\nu' \preceq \nu$. This gives rise to an efficient procedure for checking the satisfiability of a formula ϕ:

Lemma 4. *Let $\boldsymbol{\nu}'$ be a set of solutions whose elements generate all complete solutions of a formula ϕ, i. e., $\bigcup_{\nu' \in \boldsymbol{\nu}'} \overline{\langle \nu' \rangle} = \{\nu \mid \nu$ is a complete solution of* $\mathsf{abst}(\phi)\}$*. Then ϕ is satisfiable iff there exists a $\nu' \in \boldsymbol{\nu}'$ such that* $\mathsf{constr}(\phi, \nu')$ *is satisfiable.*

Note the following important facts on the approach sketched above: First, every ν' generates an exponential number of solutions with respect to its freedom $|\nu'|_?$ (Rem. 1). Furthermore, the number of atoms to check is reduced by the freedom $|\nu'|_?$ of ν'. Both reasons give an intuitive explanation for the benefit of our approach empirically confirmed in Sec. 4.

This minimisation approach suggests to find some *optimal* set $\boldsymbol{\nu}'$ of solutions to generate all complete ones. However, as even computing a single solution of minimum weight from a given one is $\mathbf{NP}$-complete and enumerating all possible solutions is $\#\mathbf{P}$-complete, it is infeasible to construct such an optimal set $\boldsymbol{\nu}'$ [6].

Thus, instead of building a set $\boldsymbol{\nu}'$ of minimal solutions at the beginning, we *minimise* each solution as generated by the SAT solver according to simple heuristics. If the obtained minimal solution does not yield a concrete solution, we use the SAT solver to produce a new solution outside the already visited search space. In the next section, we introduce the corresponding algorithm, and we discuss its efficiency in Sec. 4.

3 Solving algorithm and minimisation

We now present ABSOLVER, which implements the abstraction/refinement approach *with generalisation*, following the ideas that were laid out in the previous section. We start with the main loop of our ABSOLVER framework and subsequently discuss the minimisation algorithm which is used to generalise the arising Boolean solutions.

3.1 Main loop

ABSOLVER's main procedure solve for deciding an SMT problem is shown in Alg. 1.The procedure takes a formula ϕ as input and returns a solution τ iff ϕ is satisfiable. To do so, in line 2, a Boolean abstraction ϕ' is computed before entering the main loop. Subsequently, solve adds further clauses to ϕ' whenever it discovers unsatisfiable conjunctions of (possibly negated) constraints. In the main loop, we first compute a solution ν to the Boolean abstraction ϕ' (line 4). If no such solution exists (line 5), then there exists no solution to the original SMT instance ϕ and the procedure returns *ff* (line 6).

Otherwise, following the ideas of Section 2.2, the Boolean solution ν is generalised by reducing the weight $|\nu|$ of ν (line 8). This minimisation algorithm (minimisation) is discussed in Section 3.2. Using the now generalised solution ν to the Boolean abstraction, we construct the corresponding constraint $\mathsf{constr}(\phi, \nu)$ and use a constraint solver to search for a concrete solution τ (line 9). If a solution τ exists (line 10), then τ is indeed a solution to the original problem ϕ and accordingly, the algorithm returns τ as the solution.

If no such τ exists, an unsatisfiable subset of the literals of $\mathsf{constr}(\phi, \nu)$ is constructed by conflicts and added as a conflict clause to ϕ' (line 13). In our implementation, conflicts returns those literals which are reported to be mutually inconsistent by the employed constraint solver. If the constraint solver does not return such an unsatisfiable core, $\mathsf{conflicts}(\tau)$ returns all literals of $\mathsf{constr}(\phi, \nu)$ and consequently, all of them are added into the new conflict clause.

3.2 Minimisation

Let us now turn our attention to the generalisation algorithm minimisation shown in Alg. 2. It starts with a complete Boolean assignment ν as returned by the function boolean_solver, which we have to minimise. minimisation takes a Boolean formula ϕ' and an assignment ν which must *satisfy* ϕ' initially. The procedure maintains a set of variables V which are subsequently considered for being assigned $?$. At first, V is initialised to the set of all variables $\mathcal{V}_{\mathbb{B}}(\phi')$ of ϕ' (line 2).

ALG. 1 ABSOLVER's solving algorithm.

```
1:  proc solve($\phi$)
2:    $\phi' :=$ abst($\phi$)
3:    while $tt$ do
4:      $\nu :=$ boolean_solver($\phi'$)
5:      if $\nu$ = fail then
6:        return $ff$
7:      end if
8:      $\nu :=$ minimisation($\phi', \nu$)
9:      $\tau :=$ constraint_solver(constr($\phi, \nu$))
10:     if $\tau \neq$ fail then
11:       return $\tau$
12:     end if
13:     $\phi' := \phi' \wedge \neg$(conflicts($\tau$))
14:   end while
```

ALG. 2 Iterative minimisation algorithm.

```
1:  proc minimisation($\phi'$, $\nu$)
2:    $V := \mathcal{V}_{\mathbb{B}}(\phi')$
3:    while $tt$ do
4:      for all clauses $C_i$ of $\phi'$ do
5:        $L :=$ satisfying_literals($C_i, \nu$)
6:        if $L = \{v\}$ or $L = \{\neg v\}$ then
7:          $\phi' :=$ remove_clause($C_i, \phi'$)
8:          $V :=$ remove_variable($v, V$)
9:        end if
10:     end for
11:     if $V = \emptyset$ then
12:       return $\nu$
13:     end if
14:     $v :=$ select_variable($V$)
15:     assign $v$ in $\nu$ to ?
16:     $V :=$ remove_variable($v, V$)
17:   end while
```

Then, a loop is entered in which in each iteration at least one variable is removed from V. This loop has two parts: In lines 4–10, the clauses which are only satisfied by a single literal (line 6) are removed (line 7) from ϕ' and the corresponding variable v from V (line 8): As when a constraint is satisfied by a single literal, the corresponding variable cannot be assigned ?. If no candidate variable remains in V (line 11), the algorithm returns the resulting assignment ν. Otherwise, all variables in V can be selected to be assigned ?. Thus, the algorithm chooses a variable $v \in V$ with select_variable (line 14) according to heuristics discussed below and reassigns ? to v (line 15). This v is then removed from V (line 16)—and a new iteration starts. Note that the number of iterations is bounded by the number of variables.

Selection heuristics. Presumably the choice of the variable to be assigned ? (implemented by select_variable) plays a crucial role in the efficiency of the overall decision procedure. Therefore, we experimented with the following three different heuristics: *Input-order rule:* In the simplest form, variables are chosen according to the structure of the input formula. *Purity-frequency rule:* Pure literals are those which occur in a given formula either only negative, or only positive. In this case, select_variable always prefers a pure variable over a non-pure one. *Representative rule:* Applying this heuristic, variables that represent constraints of the background theory are preferably assigned ?. Observe that minimisation runs with the proposed selection heuristics in polynomial time with respect to the size of ϕ.

It is easy to construct test cases which strongly discriminate between these variants, as well as test cases where the heuristics do not apply. Interestingly enough, in the benchmarks described in the next section, which are taken from the SMT-LIB, the heuristics performed roughly equal. The measured differences in performance were only on a marginal scale, indicating that either way good (or, bad) candidates for elimination were found.

Note that the minimisation algorithm is easily integrated into other abstraction/refinement solvers as a subsequent step *after* the Boolean part of an SMT problem has been solved by an arbitrary SAT solver, as shown in Alg. 1.Moreover, it would be possible (and, arguably, sometimes even more efficient) to modify the internals of a SAT solver in order to obtain a generalisation directly. However, this requires more development effort and ties the SMT solver to a particular version of a particular tool. Additionally, most of today's competitive SAT solvers make use of highly integrated algorithms, such that making modifications to them, even small ones, becomes a non-trivial and error-prone task. Consequently, having a separate generalisation algorithm gives us the flexibility we need, and eases implementation.

4 Implementation and benchmarks

This section briefly discusses implementation details of ABSOLVER and gives three kinds of benchmarks showing the efficiency of our approach. First, we show the speed-up of using the generalisation approach by comparing ABSOLVER without and with generalisation on existing benchmarks. Second, we compare ABSOLVER with third-party SMT solvers that follow both an iterative approach and an abstraction/refinement approach, showing that our approach yields an inferior but still estimable solver. Most interestingly, we report that we indeed easily obtained an SMT solver for non-linear arithmetic constraints that helped us to verify a car's electronic steering control system.

ABSOLVER as originally introduced in [3], is a C++ framework that, once combined with the appropriate solvers, can be either used as a stand-alone tool, or integrated in terms of a system library, e. g., to extend other constraint-handling systems. In the discussion that follows, we refer to ABSOLVER as the framework in its original form, and ABSOLVERDC as the framework that has now been extended with the iterative minimisation algorithm described above. Currently, ABSOLVER interfaces with LSAT [2], grasp [14] and (z)Chaff [15], although in this paper, only the latter was used to run benchmarks. The concretisation is handled by specialised solvers offered by the COIN-OR library [13]. Basically, the COIN-OR library is a collection of dedicated, and more or less independently developed constraint solvers, covering, e. g., linear arithmetic, or non-linear arithmetic, each with a different solver.

An input problem to ABSOLVER (and, therefore, to ABSOLVERDC) then consists of a standard DIMACS [7] format SAT problem, where the background constraints are expressed in a custom language, encoded in the DIMACS comments. This way, the abstract part of an ABSOLVER problem is already understood by any standard SAT solver, but naturally "wrapper" code has to be written for processing the solver's return set correctly. Part of the solver "wrapper" is also the iterative minimisation algorithm for the SAT solver, i. e., each assignment produced by the SAT solver is first generalised, before the concrete solution is determined. Moreover, the "wrapper" is also responsible for evaluating the return values of the constraint solver, and for adding the negated abstract solution back to the input clause, if necessary. This design facilitates a loose integration of the individual solver. However, we expect some constant penalty on all benchmarks, because the "wrapper" has to do type or character marshalling of input and return values to solvers, rather than accessing a solver's data structures directly in terms of, say, pointers to memory locations.

The benchmarks presented in the following sections have been executed using a timeout of two hours, and a memory limit of 1.2 GB on a 3.2 GHz Intel Xeon system, equipped with 2 GB of RAM. All test cases are taken from the QF_LIA suite that is part of the SMT-LIB benchmarks [17].

4.1 ABSOLVER vs. ABSOLVERDC

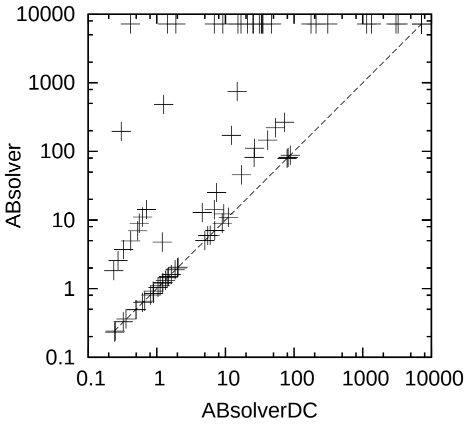
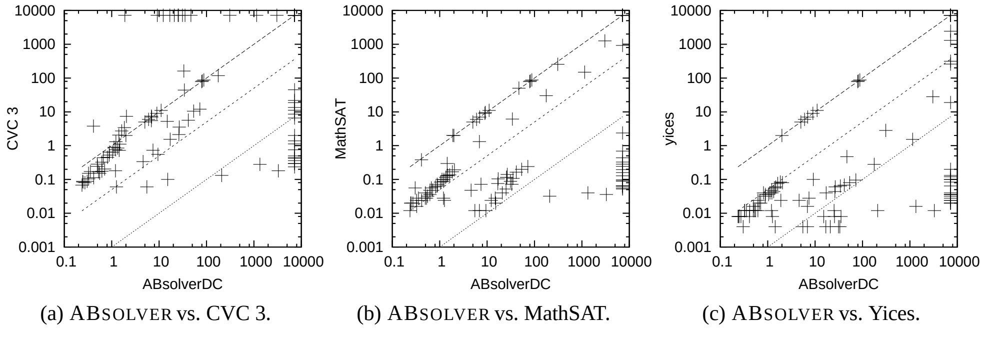


FIG. 1: With and without *don't care*s.

A direct comparison between ABSOLVER and ABSOLVERDC is shown in Fig. 1. Each test case is represented by a cross in the diagram, where the x-coordinate reflects the runtime of ABSOLVERDC, and the y-coordinate the runtime of ABSOLVER. Consequently, when ABSOLVERDC outperforms ABSOLVER, the corresponding cross is located within the upper left area of the diagram. Both, the x- and y-axis show the runtime in seconds, based on a logarithmic scale. Marks at the upper and rightmost end of the diagram denote timeouts of ABSOLVER and ABSOLVERDC, respectively. Fig. 1 indicates that, in all test cases, ABSOLVERDC is at least as efficient as ABSOLVER, and even outperforms ABSOLVER in roughly one quarter of the test cases by more than an order of magnitude. Those runs, in turn, exhibit speed ups of more than three orders of magnitude. Note that more than 20 test cases resulted in timeouts of ABSOLVER, whereas ABSOLVERDC was still able to solve these efficiently.

4.2 Comparison with other solvers

In Fig. 2, ABSOLVERDC is compared to CVC 3, MathSAT, and Yices. Let us use the same type of diagram as for the comparison between ABSOLVERDC and ABSOLVER above, i. e., for each test run, a cross is added in a square such that the x- and y-coordinate reflect the runtime of ABSOLVERDC and the other solver on a logarithmic scale, respectively. Not surprisingly, other solvers which employ an iterative approach, still perform better in these test runs than ABSOLVERDC does. However, ABSOLVERDC shows a comparatively stable and reliable performance compared to these solvers. In fact, due to the optimisations in place, ABSOLVERDC is able to solve most test runs in additional time which is only greater by a constant factor. As shown in Fig. 2a, ABSOLVERDC is comparable to CVC 3, since most test runs are clustered around the diagonal line, and since both tools are able to solve some test cases which cannot be solved by the respective competitor. Fig. 2b, and 2c show that ABSOLVERDC is clearly slower than MathSAT and Yices. However, 60% of all benchmarks are solved by ABSOLVERDC within a runtime which is only larger by a constant factor. This



(a) ABSOLVER vs. CVC 3. (b) ABSOLVER vs. MathSAT. (c) ABSOLVER vs. Yices.

FIG. 2: A detailed comparison.

is indicated by the diagonal lines, as due to the logarithmic scale of the diagrams a constant factor translates to diagonal corridors. The corridors represent factors of 20, and 100 in Fig. 2a-c, respectively. Note that part of this overhead is due to the text/file-based interface to the underlying solver.

4.3 Industrial case-study with non-linear arithmetic constraints

The ABSOLVER framework was originally developed to handle general mixed arithmetic and Boolean constraints as arising in the verification of MATLAB/Simulink models [3]. To the best of our knowledge, no pre-existing tools supported the occurring non-linear constraints imposed by these models. Consequently, we integrated a specialised non-linear constraint solver, as provided by the COIN-OR library, into ABSOLVER.

We have employed successfully ABSOLVER in verifying a number of properties of a car's steering control system. The continuous dynamics of the controller and its environment had been modelled using MATLAB/Simulink, where the environment consisted of non-linear functions modelling the physical behaviour of the car. An automated conversion (using a custom tool-chain) resulted in 976 CNF-clauses, and 24 (non-) linear expressions representing the constraints. Currently, ABSOLVER in its original version is able to solve the imposed constraint problem in 17 seconds. On the other hand, our optimised solver ABSOLVERDC, was able to solve the same problem in only 9 seconds, giving a speed-up of roughly 50%. In both cases the employed theory solvers were COIN [13] (for the linear part), zChaff [15] (for the Boolean part), and IPOPT [24] (for the non-linear part).

5 Conclusions

In this paper, we presented a simple yet surprisingly efficacious optimisation to the abstraction/refinement approach in SMT solving. Starting with our ABSOLVER framework as presented in [3], we were able to improve the performance of the solver substantially by *generalising* a SAT solver's solution, before generating and solving the underlying constraint problem. This yields fewer and smaller constraint problems than the traditional approach. Our experiments confirm that the optimisation improves the traditional abstraction/refinement approach and pushes the ABSOLVER framework in a practically applicable range.

In many fundamental domains, specialised SMT solvers exist and ABSOLVER cannot compete with these solvers. However, to build an SMT solver with our framework, it is sufficient to integrate a SAT solver and non-incremental theory solvers *as black boxes*.

Therefore, ABSOLVER provides a useful trade-off point between research and development effort on the one hand side, and the domain of solvable problems on the other: With a minimum engineering effort, we were able to build a solver for non-linear arithmetic SMT problems and to successfully apply this solver in verifying a car's electronic steering control system—no other solver was able to process these non-linear constraints before. As such our framework somewhat closes the gap between more advanced SMT solvers being developed in research, and currently arising industrial problems which are often based upon hitherto unsupported theories.

Acknowledgements. Thanks to Jinbo Huang, NICTA, for comments on an earlier version of this paper.

References

[1] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Computer Aided Verification (CAV)*, pages 515–518, 2004.

[2] A. Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AIO-R)*, pages 49–63, 2005.

[3] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Design, Automation and Test in Europe (DATE)*, pages 924–929, 2007.

[4] A. Belov and Z. Stachniak. Substitutional definition of satisfiability in classical propositional logic. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 31–45, 2005.

[5] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 317–333, 2005.

[6] J. P. Delgrande and A. Gupta. The complexity of minimum partial truth assignments and implication in negation-free formulae. *Ann. Math. Artif. Intell.*, 18(1):51–67, 1996.

[7] Satisfiability: Suggested format. Technical report, 1993.

[8] N. Een and N. Sörensson. An extensible sat-solver. In *Theory and Application of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[9] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification (CAV)*, pages 175–188, 2004.

[10] R. B. Jones and D. L. Dill. Automatic verification of pipelined microprocessors control. In *Computer Aided Verification (CAV)*, pages 68–80, 1994.

[11] L. M. Kirousis and P. G. Kolaitis. The complexity of minimal satisfiability problems. *Inf. Comput.*, 187(1):20–39, 2003.

[12] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Computer Aided Verification (CAV)*, pages 424–437, 2006.

[13] R. Lougee-Heimer. The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. *IBM J. Res. Dev.*, 47(1):57–66, 2003.

[14] J. P. Marques-Silva and K. A. Sakallah. GRASP—A New Search Algorithm for Satisfiability. In *Int. Conf. Computer-Aided Design (ICCAD)*, pages 220–227, 1996.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535, 2001.

[16] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[17] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Dep. of Comp. Sci., University of Iowa, 2006. `www.SMT-LIB.org`.

[18] Y. Rodeh and O. Strichman. Building small equality graphs for deciding equailty logic with uninterpreted functions. *Informantion and Computation*, 204(1):26–59, 2006.

[19] J.-W. Roorda and K. Claessen. SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In *Computer Aided Verification (CAV)*, pages 175–189, 2006.

[20] J. Rushby. Harnessing disruptive innovation in formal verification. In *Software Engineering and Formal Methods (SEFM)*, pages 21–30, 2006.

[21] J. Rushby. Tutorial: Automated formal methods with PVS, SAL, and Yices. In *Software Engineering and Formal Methods (SEFM)*, page 262, 2006.

[22] H. M. Sheini and K. A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Application of Satisfiability Testing (SAT)*, pages 241–256, 2005.

[23] H.M. Sheini and K.A. Sakallah. From Propositional Satisfiability to Satisfiability Modulo Theories. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 1–9, 2006.

[24] A. Wächter and L. T. Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. *SIAM Journal on Optimization*, 16(1):1–31, 2005.