Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Don't care in SMT[⋆]

Building flexible yet efficient abstraction/refinement solvers

Andreas Bauer[1], Martin Leucker[2], Christian Schallhart[3], Michael Tautschnig[3]

[1] Australian National University & National ICT Australia (NICTA), e-mail: baueran@rsise.anu.edu.au
[2] Institut für Informatik, Technische Universität München, Germany, e-mail: leucker@in.tum.de
[3] FB Informatik, Technische Universität Darmstadt, Germany, e-mail: {schallhart,tautschnig}@forsyte.de

Received: date / Accepted: date

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.

Key words: SMT, Verification, Constraint 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 linear integer arithmetic.*

Generally, a *theory* refers to the set of statements that are deducable from given axioms via some deducability relation. In the context of this paper, we consider quantifier-free first-order theories, i.e., statements are built-up from function and predicate symbols and are interpreted over a fixed domain. Thus, depending on the considered domain (like *integer* versus *reals*), the involved function and predicate symbols, and axioms, different theories arise. The simplest perhaps being *equality logic*, as it does not offer any function symbols at all, and has only one binary predicate, the equality predicate (cf. Zantema and Groote (2003)). In the more expressive *difference logic*, which is also used for reasoning about programs (cf. Ball et al (2005)), statements use interpreted predicates of the form $t_i - t_j \leq d$, where t_i and t_j are numerical variables or compound terms and d an integer constant. In *linear integer arithmetic* used above, one can basically specify general inequalities over the domain of the integers (cf. Shostak (1981)). In that sense, a SAT solver is a theory solver for the propositional domain, where only null-ary predicate symbols (i.e., propositional variables) are allowed. While it is known that the general theory of integers is not decidable, there exist decidable fragments in it which are commonly used in the area of SMT or automated theorem proving.

In recent years, research on SMT has attracted a lot of attention. SMT solvers for dedicated theories have been developed, such as Yices (Dutertre and de Moura, 2006; Rushby, 2006b), MathSAT (Bozzano et al, 2005), or CVC (Barrett and Berezin, 2004). 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 areas such as model checking and abstraction (Lahiri

⋆ Supported by the DFG research grant FORTAS (VE 455/1-1)

et al, 2006), (symbolic) test-case generation (Roorda and Claessen, 2006), or in the verification of hybrid control systems (Bauer et al, 2007b; Rushby, 2006a), 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 (Wächter and Biegler, 2005). 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.

Organisation. In the next section, we give an account of the different approaches to SMT solving, and put some of the key ideas underlying our own work into context. In Section 3, we provide the foundations for the abstraction/refinement approach followed by ABsolver, before we describe the relation between *generalised* abstract solutions and concrete solutions to an SMT problem. In Section 4, we provide an algorithmic solution to the SMT problem, based on generalisation. In Section 5, we discuss implementational aspects. Experimental results in form of comparative benchmarks and several case studies are given in Section 6. Section 7 concludes our paper.

This paper is based on earlier versions that appeared as Bauer et al (2007a) and Bauer et al (2007b).

2 Existing approaches to SMT

The existing approaches to solve SMT problems can be subdivided into three main categories (see also (Sheini and Sakallah, 2006) for an overview).

In the *translation approach* (Sheini and Sakallah, 2006), 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. Een and Sörensson (2003); Moskewicz et al (2001); Prasad et al (2005)) this approach turned out quite successful—at least for certain background theories, see for example Jones and Dill (1994); Rodeh and Strichman (2006). 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* (Sheini and Sakallah, 2005), 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* (Ganzinger et al, 2004), 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 by Ganzinger et al (2004). 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 (Bauer et al, 2007b), 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 (in a sense made precise in Section 3) 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 cares*.

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 MIN-SAT problem and its variations (Belov and Stachniak, 2005; Delgrande and Gupta, 1996; Kirousis and Kolaitis, 2003), which, however, are known to be NP-complete (Delgrande and Gupta, 1996). 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 Section 6). 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.

3 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 and variables. Let Σ be a finite set of *types* and $\mathcal{D} = (\mathbb{D}_\sigma)_{(\sigma \in \Sigma)}$ be a family of respective *domains.* We require that every domain is non-trivial, i.e., every domain has at least two elements. 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*.

Assignments. 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

$$\tau \preceq \tau' \text{ iff for all } v \in \mathcal{V} \quad \tau(v) \preceq \tau'(v).$$

Thus, τ is smaller than τ' w. r. t. $\prec$, if reassigning ? to a number of variables in τ' results in τ. Likewise, τ' is larger than τ if τ' coincides with τ, except that it at most assigns values to variables that yield ? under 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 cares* 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' which are larger than τ, i.e.,

$$\langle\tau\rangle = \{\tau' \mid \tau \preceq \tau'\}$$

Similarly, the set of complete assignments generated by τ, denoted by $\overline{\langle\tau\rangle}$, is given by the set of complete assignments larger than τ, i.e.,

$$\overline{\langle\tau\rangle} = \{\tau' \mid \tau \preceq \tau' \text{ and } \tau' \text{ complete}\}$$

Remark 1. The number of complete assignments generated by an assignment τ is exponential in its freedom:

$$|\overline{\langle\tau\rangle}| \geq 2^{|\tau|_?}$$

We have $|\overline{\langle\tau\rangle}| = 2^{|\tau|_?}$ when all domains have exactly two elements.

Formulas. Let $\mathcal{F} = (F_\sigma)_{\sigma \in \Sigma}$ be a family of ranked function symbols and $\mathcal{P} = (P_\sigma)_{\sigma \in \Sigma}$ be a family of ranked predicate symbols. The set of (typed) *terms* is inductively defined by the two rules below:

– Every variable of $\mathcal{V}_\sigma$ is a term of type σ, and,
– 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 $\mathcal{V}_\mathbb{B}(\phi) = \{x, y\}$:

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

Solutions. A *structure* T pairs the domain $\mathcal{D}$ with an *interpretation* of the function and relational symbols over the respective domains.

A *complete solution* of ϕ (with respect to the structure T) 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 Example 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.

Note that, opposed to the literature, we identify in this paper solutions with respect to structures and *theories*. However, as long as faced with first-order logic, there is no significant difference.

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. In other words, the values assigned by τ to variables guarantee that ϕ evaluates to *tt*, regardless of what value a ? may take when making τ complete. 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 Example 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. In other words, trying to reduce τ' further by reassigning one of its values to ? would violate the property that all assignments generated by τ' are solutions. 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. This τ' is not only a minimal solution 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.

3.1 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

$$\mathcal{V}^R_\mathbb{B}(\mathsf{abst}(\phi)) \subseteq \mathcal{V}_\mathbb{B}(\mathsf{abst}(\phi)).$$

Since all representative variables do not occur otherwise in ϕ, we have $\mathcal{V}^R_\mathbb{B}(\mathsf{abst}(\phi)) \cap \mathcal{V}(\phi) = \emptyset$.

Example 2. The Boolean abstraction of ϕ shown in Example 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 $\mathcal{V}^R_\mathbb{B}(\mathsf{abst}(\phi)) = \{v_1, \ldots, v_5\}$. 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

$$\nu(p_{ij}) := (p_{ij}(a_1, \dots, a_{n_{ij}}))[\tau]$$

where $(p_{ij}(a_1, \dots, a_{n_{ij}}))[\tau]$ denotes the truth value of the atom $p_{ij}(a_1, \dots, a_{n_{ij}})$ 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 solution of ϕ. Then $\mathsf{abst}(\tau)$ is a solution of $\mathsf{abst}(\phi)$. If τ is moreover complete, $\mathsf{abst}(\tau)$ is also complete.

Concretisation. We now turn our attention towards concretisations of abstract solutions. 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

1. searching for a complete solution ν of $\mathsf{abst}(\phi)$ and then
2. 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

$$\begin{aligned}\mathsf{constr}(\phi, \nu) \equiv & \textstyle\bigwedge_{\nu(p_{ij})=tt} p_{ij}(a_1, \dots, a_{n_{ij}}) \wedge \\ & \textstyle\bigwedge_{\nu(p_{ij})=ff} \neg p_{ij}(a_1, \dots, a_{n_{ij}})\end{aligned}$$

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*

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

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

Example 3. Let us come back to our running example (see Example 1/2). One solution of

$$\mathsf{abst}(\phi) \equiv \{v_1 \vee y\} \wedge \{\neg v_2 \vee v_3\} \wedge \{x \vee \neg v_4\} \wedge \{v_5\}$$

is given by $\nu(v_1) = \nu(y) = \nu(v_3) = \nu(x) = \nu(v_4) = \nu(v_5) = tt$ and $\nu(v_2) = ff$ resulting in the constraint problem

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

However, this constraint problem is not satisfiable since $\neg(2i + j < 10) \wedge (i + j < 5)$ is not satisfiable when both i and j are required to be positive. A further solution of the abstract system is given by $\nu(v_1) = \nu(v_2) = \nu(v_3) = \nu(v_4) = \nu(x) = \nu(v_5) = tt$ and $\nu(y) = ff$ resulting in the constraint problem

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

which is satisfiable, for example, with the aforementioned complete solution $\tau(i) = 3$, $\tau(j) = 1$, $\tau(k) = 0$, $\tau(l) = 2$, $\tau(x) = tt$, and $\tau(y) = ff$.

The previous example shows that the typical abstraction/refinement approach does not yield incomplete and especially no minimal solutions. This, however, can be achieve by *generalising* abstract solutions, a concept which we introduce next.

3.2 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 derive 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 $p_{ij}(a_1, \dots, a_{n_{ij}})$ 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*

1. *a (possibly incomplete) solution ν' of $\mathsf{abst}(\phi)$ and*
2. *$\mathsf{constr}(\phi, \nu')$ 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 $(p_{ij}(a_1, \dots, a_{n_{ij}}))[\tau] = \nu'(p_{ij})$ 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.

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.

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.,*

$$\bigcup_{\nu' \in \boldsymbol{\nu}'} \overline{\langle \nu' \rangle} = \{\nu \mid \nu \text{ is a complete solution of } \mathsf{abst}(\phi)\}$$

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 $|\nu'|_?$ (Remark 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 Section 6.

Example 4. Reconsider

$$\mathsf{abst}(\phi) \equiv \{v_1 \vee y\} \wedge \{\neg v_2 \vee v_3\} \wedge \{x \vee \neg v_4\} \wedge \{v_5\}$$

from Example 2. In Example 3, we have identified $\nu(v_1) = \nu(v_2) = \nu(v_3) = \nu(v_4) = \nu(x) = \nu(v_5) = tt$ and $\nu(y) = ff$ as a solution of this abstract system. However, this solution is clearly not minimal: For example, the values of y, v_2, and v_4 do not influence the truth value of the abstract system, if the values for the other variables are fixed. Thus, $\nu(v_1) = \nu(v_3) = \nu(x) = \nu(v_5) = tt$ and $\nu(y) = \nu(v_2) = \nu(v_4) = ?$ is a smaller and even minimal solution, resulting in the constraint problem

$$\begin{aligned}\phi \equiv\ & (i \geq 0)\\ & \wedge(i + j < 5)\\ & \wedge x\\ & \wedge(k + (4 - k) + 2l \geq 7)\end{aligned}$$

which is satisfiable, for example, with the incomplete solution $\tau(i) = 3$, $\tau(j) = 1$, $\tau(k) = 0$, $\tau(l) = 2$, $\tau(x) = tt$.

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 **NP**-complete and enumerating all possible solutions is #**P**-complete, it is infeasible to construct such an optimal set $\boldsymbol{\nu}'$ (Delgrande and Gupta, 1996).

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 Section 6.

4 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 ABsolver and subsequently discuss the minimisation algorithm which is used to generalise the arising Boolean solutions. Finally we present and discuss a number of selection heuristics we used in the minimisation algorithm.

4.1 Main loop

ABsolver's main procedure solve for deciding an SMT problem is shown in Figure 1. The procedure takes a formula ϕ as input and returns a solution τ iff ϕ is satisfiable. To do so, in line 5, a Boolean abstraction ϕ' is computed with $\phi' := \mathsf{abst}(\phi)$ 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' with $\nu := \mathsf{boolean\_solver}(\phi')$ (line 7). If no such solution exists (line 8), then there exists no solution to the original SMT instance ϕ and the procedure returns ff (line 9).

Otherwise, following the ideas of Section 3.2, the Boolean solution ν is generalised by reducing the weight $|\nu|$ of ν with $\nu := \mathsf{minimisation}(\phi',\nu)$ (line 11). The minimisation algorithm ($\mathsf{minimisation}$) is discussed in Section 4.2. 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 12). If a solution τ exists (line 13), 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 $\mathsf{conflicts}(\tau)$ of the literals of $\mathsf{constr}(\phi,\nu)$ is identified by the procedure $\mathsf{conflicts}$ and added as a conflict clause $\neg\mathsf{conflicts}(\tau)$ to ϕ' (line 16). In our implementation, $\mathsf{conflicts}$ 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.

```
INPUT:    SMT instance $\phi$
OUTPUT:   if $\phi$ is satisfiable, a solution $\tau$ to $\phi$
          otherwise $\mathit{ff}$
proc solve($\phi$)
    $\phi' :=$ abst($\phi$)                          // initial Boolean Abstraction
    while $\mathit{tt}$ do
        $\nu :=$ boolean_solver($\phi'$)             // Boolean abstract solution
        if $\nu =$ fail then
            return $\mathit{ff}$                     // no concrete solution exists anymore
        end if
        $\nu :=$ minimisation($\phi', \nu$)          // generalisation via minimisation.
        $\tau :=$ constraint_solver(constr($\phi, \nu$))   // concretisation via constraint solver.
        if $\tau \neq$ fail then
            return $\tau$                            // concrete solution found
        end if
        $\phi' := \phi' \wedge \neg$(conflicts($\tau$))    // avoid this conflict in the future
    end while
```

Fig. 1: ABsolver's solving algorithm

4.2 Minimisation

Let us now turn our attention to the generalisation algorithm minimisation shown in Figure 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 5).

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 7–13, the clauses which are only satisfied by a single literal v or $\neg v$ (line 9) are removed (line 10) from ϕ' and the corresponding variable v from the set of variables V (line 11), because when a constraint is satisfied by a single literal, the corresponding variable cannot be assigned ?. Moreover, this step also potentially removes all occurrences of v in the remaining clauses, thus shortening them iteratively. If no candidate variable remains in V (line 14), 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 17) according to heuristics discussed below and reassigns ? to v (line 18). This v is then removed from V (line 19)—and a new iteration starts. Note that the number of iterations is bounded by the number of variables.

4.3 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 are not useful. 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 Figure 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

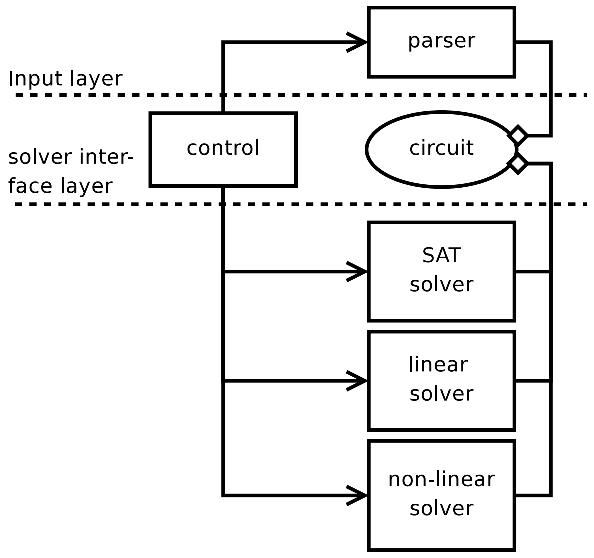


Fig. 3: Architecture of ABsolver.

ones, becomes a non-trivial and error-prone task. Consequently, having a separate generalisation algorithm gives us the flexibility we need, and eases implementation.

5 Implementation

ABsolver as originally introduced by Bauer et al (2007b), is a C++ framework that, once combined with the appropriate solvers, can be either used as a standalone 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. An overview of the architecture of the framework is presented in Figure 3.

ABsolver's core comprises a data structure for modelling an *integrated circuit* where arithmetic and Boolean operations are represented as gates taking either a single (e. g., negation), a pair (e. g., arithmetic comparison), or an arbitrary number of inputs. The variables are then seen as the input pins of a circuit, and the single output pin provides the formula's truth value, which is either *tt*, *ff*, or $\perp$ indicating that further treatment is necessary, internally.

An input problem to ABsolver (and, therefore, to ABsolverDC) then consists of a standard DIMACS (DIMACS, 1993) format SAT problem, where the background constraints are expressed in a custom language, encoded in the DIMACS comments, briefly shown in Figure 4. 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.

```
p cnf 4 3
1 0
-2 3 0
4 0

c def int 1 i >= 0
c def int 1 j >= 0
c def int 2 2*i + j < 10
c def int 3 i + j < 5
c def real 4 a * x + 3.5 / ( 4 - y ) + 2 * y >= 7.1
```

$$((i \geq 0) \wedge (j \geq 0))$$
$$\wedge \Big(\neg(2i + j < 10) \vee \boxed{\texttt{(i + j < 5)}}\Big)$$
$$\wedge \Big(a \cdot x + \tfrac{3.5}{4-y} + 2y \geq 7.1\Big)$$

Fig. 4: An SMT instance and ABsolver's representation.

This processing is performed by the *solver interface layer*. Using the circuit-object as described above, the solver interface computes variable assignments. Currently, ABsolver interfaces with LSAT (Bauer, 2005), grasp (Marques-Silva and Sakallah, 1996) and (z)Chaff (Moskewicz et al, 2001), 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 (Lougee-Heimer, 2003). 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.

Part of the SAT solver interface 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, where it is used as a black box. Which solver actually gets used to solve a problem, i.e., the linear or the non-linear solver, is determined solely by the background theory. 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.

6 Experimental results

This section gives three kinds of benchmarks showing the efficiency of our approach. First, we show the speedup 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.

```
INPUT:  Boolean abstraction $\phi'$ of an SMT instance,
        an assignment $\nu$ satisfying $\phi'$
OUTPUT: an assignment $\nu'$ satisfying $\phi'$ with $\nu' \preceq \nu$
proc minimisation($\phi'$, $\nu$)
    $V := \mathcal{V}_{\mathbb{B}}(\phi')$                  // extract Boolean variables as dontcare-candidates
    while $tt$ do
        for all clauses $C_i$ of $\phi'$ do
            $L :=$ satisfying_literals($C_i, \nu$)
            if $L = \{v\}$ or $L = \{\neg v\}$ then
                $\phi' :=$ remove_clause($C_i, \phi'$)   // remove all clauses satisfied by a single variable
                $V :=$ remove_variable($v, V$)   // remove the variable as dontcare-candidate
            end if
        end for
        if $V = \emptyset$ then
            return $\nu$                        // no more candidates, return assignment
        end if
        $v :=$ select_variable($V$)             // select dontcare variable
        assign $v$ in $\nu$ to ?                // generalise assignment
        $V :=$ remove_variable($v, V$)          // remove as candidate
    end while
```

Fig. 2: Iterative minimisation algorithm.

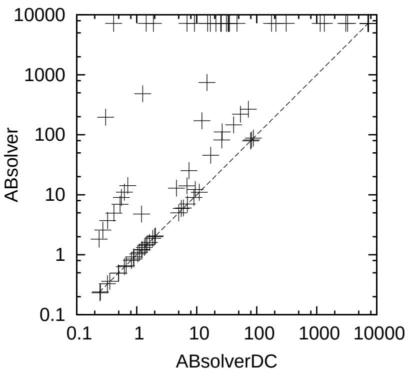


Fig. 5: With and without *don't cares*.

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.

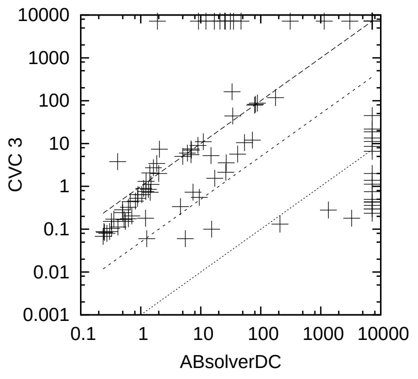
The benchmarks presented in the following sections have been executed using a timeout of two hours, and a memory limit of 1.2 GB on a 3.2 GHz Intel Xeon system, equipped with 2 GB of RAM. All test cases are taken from the QF_LIA suite that is part of the SMT-LIB benchmarks (Ranise and Tinelli, 2006).

6.1 ABsolver *vs.* ABsolverDC

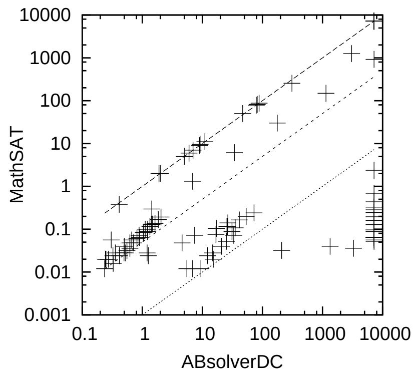
A direct comparison between ABsolver and ABsolverDC is shown in Figure 5. Each test case is represented by a cross in the diagram, where the x-coordinate reflects the runtime of ABsolverDC, and the y-coordinate the runtime of ABsolver. Consequently, when ABsolverDC outperforms ABsolver, the corresponding 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. Figure 5 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.

6.2 Comparison with other solvers

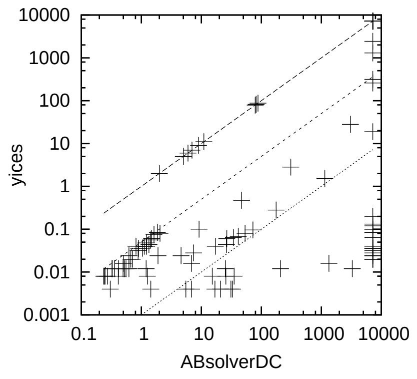
In Figure 6, ABsolverDC is compared to CVC 3, MathSAT, Yices, Ario, HTP and ExtSAT, individually. These solvers where chosen, because they competed in SMT-COMP 06. Let us use the same type of diagram as for the comparison between ABsolverDC and ABsolver above, i. e., for each test run, a cross is added 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



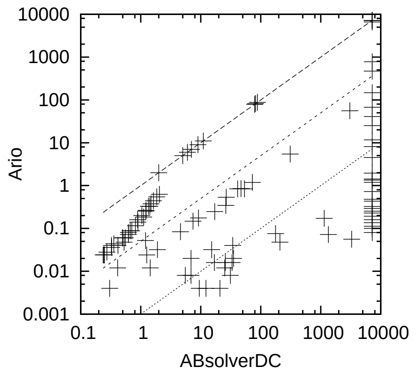
(a) ABsolver vs. CVC 3.



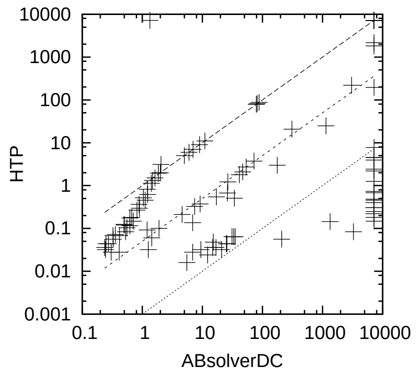
(b) ABsolver vs. MathSAT.



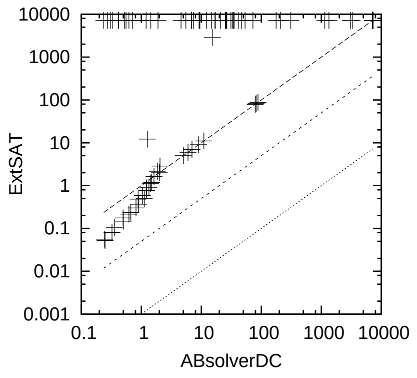
(c) ABsolver vs. Yices.



(d) ABsolver vs. Ario.



(e) ABsolver vs. HTP.



(f) ABsolver vs. ExtSAT.

Fig. 6: A detailed comparison.

to solve most test runs in additional time which is only greater by a constant factor. As seen from Figure 6f ABsolverDC outperforms ExtSAT on most of the benchmarks that took more than one second to solve. Furthermore, as shown in Figure 6a, 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. Figure 6b, and 6c, 6d, and 6e show that ABsolverDC is clearly slower than MathSAT, Yices, Ario and HTP. However, 60% of all benchmarks are solved by ABsolverDC within a runtime which is only larger by a constant factor. This 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. Note that part of this overhead is due to the text/file-based interface to the underlying solver.

6.3 Sudoku

To evaluate the strength of the mathematical solvers we encoded instances of the Sudoku puzzle as SMT instances. Sudoku is a logic problem, where the player has to arrange numbers from 1 to 9 horizontally as well as vertically in 9×9 squares, such that no numbers appear twice in a row.

The problem is known to be hard for NP and translations to SAT are well established (cf. Lynce and Ouaknine (2006); Weber (2005)). Having a solver at hand which solves Boolean as well as linear problems, however, the Sudoku puzzle can be tackled more efficiently as a mixed problem and the encoding is more natural as it can make use of integers. The resulting problems then constitute hard integer programming problems with a simple Boolean part.

The benchmark results shown in Figure 7 where obtained for Sudoku instances presented at `http://sudoku.zeit.de/`; dates indicate the magazine's respective issue and given hardness.

We used the LSAT and COIN-OR solvers provided with ABsolverDC, and compared to Yices and MathSAT. The results indicate that our specialised arithmetic solver is as powerful as that of Yices. Contrariwise, MathSAT, which had performed very well on generic SMT-LIB benchmark instances, is slower by several orders of magnitude.

6.4 Industrial case-study with non-linear arithmetic constraints

The ABsolver framework was originally developed to handle general mixed arithmetic and Boolean

Benchmark	ABsolverDC	Yices	MathSAT
2006_05_23_hard	0.283	0.187	5047.385
2006_05_24_hard	0.283	0.131	5988.447
2006_05_25_hard	0.282	0.378	6420.860
2006_05_26_hard	0.289	0.422	6750.929
2006_05_27_hard	0.289	0.532	5388.470
2006_05_28_hard	0.282	0.162	7049.500
2006_05_29_easy	0.279	0.128	4887.008
2006_05_29_hard	0.283	0.201	8251.245
2006_05_30_easy	0.287	0.151	4517.435
2006_05_30_hard	0.283	0.142	5675.672

Fig. 7: Results: Sudoku puzzles.

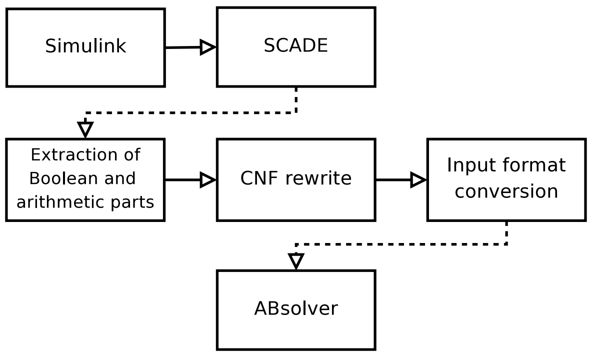


Fig. 8: Automated conversion work-flow.

constraints as arising in the verification of MATLAB/Simulink models (Bauer et al, 2007b). 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.

We therefore implemented a prototype tool-chain as outlined in Figure 8 to convert the control model from MATLAB/Simulink to ABsolver's input format. Note that conversion takes advantage of the SCADE modelling and verification suite which can import MATLAB/Simulink models. However, using SCADE in the conversion was merely a matter of convenience, because internally, SCADE uses a textual representation of the model in terms of the programming language Lustre (Halbwachs et al, 1991), from which we could then extract the multi-domain constraint satisfaction problems. 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 (Lougee-Heimer, 2003) (for the linear part), zChaff (Moskewicz et al, 2001) (for the Boolean part), and IPOPT (Wächter and Biegler, 2005) (for the non-linear part).

7 Conclusions

We have presented a simple yet surprisingly efficacious optimisation to the abstraction/ refinement approach in SMT solving. Starting with our ABsolver framework as originally presented by Bauer et al (2007b), 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 our framework in a practically applicable range.

In many 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, National ICT Australia (NICTA), for comments on an earlier version of this paper.

References

Ball T, Lahiri SK, Musuvathi M (2005) Zap: Automated theorem proving for software analysis. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), pp 2–22

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

Bauer A (2005) 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-AI-OR), pp 49–63

Bauer A, Leucker M, Schallhart C, Tautschnig M (2007a) Don't care in SMT—building flexible yet efficient abstraction/refinement solvers. In: Proceedings of the 2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'07), France, RNTI, Revue des Nouvelles Technologies de l'Information

Bauer A, Pister M, Tautschnig M (2007b) Tool-support for the analysis of hybrid systems and models. In: Design, Automation and Test in Europe (DATE), pp 924–929

Belov A, Stachniak Z (2005) Substitutional definition of satisfiability in classical propositional logic. In: Theory and Applications of Satisfiability Testing (SAT), pp 31–45

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

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

DIMACS (1993) Satisfiability: Suggested format. Tech. rep.

Dutertre B, de Moura LM (2006) A fast linear-arithmetic solver for DPLL(T). In: Computer Aided Verification (CAV), pp 81–94

Een N, Sörensson N (2003) An extensible sat-solver. In: Theory and Application of Satisfiability Testing (SAT), pp 502–518

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

Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous dataflow programming language lustre 79(9):1305–1320

Jones RB, Dill DL (1994) Automatic verification of pipelined microprocessors control. In: Computer Aided Verification (CAV), pp 68–80

Kirousis LM, Kolaitis PG (2003) The complexity of minimal satisfiability problems. Inf Comput 187(1):20–39

Lahiri SK, Nieuwenhuis R, Oliveras A (2006) SMT techniques for fast predicate abstraction. In: Computer Aided Verification (CAV), pp 424–437

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

Lynce I, Ouaknine J (2006) Sudoku as a SAT problem. In: Proc. of the Ninth International Symposium on Artificial Intelligence and Mathematics

Marques-Silva JP, Sakallah KA (1996) GRASP—A New Search Algorithm for Satisfiability. In: Int. Conf. Computer-Aided Design (ICCAD), pp 220–227

Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Design Automation Conference (DAC), pp 530–535

Prasad MR, Biere A, Gupta A (2005) A survey of recent advances in sat-based formal verification. Software Tools for Technology Transfer (STTT) 7(2):156–173

Ranise S, Tinelli C (2006) The SMT-LIB Standard: Version 1.2. Tech. rep., Dep. of Comp. Sci., University of Iowa, `www.SMT-LIB.org`

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

Roorda JW, Claessen K (2006) SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In: Computer Aided Verification (CAV), pp 175–189

Rushby J (2006a) Harnessing disruptive innovation in formal verification. In: Software Engineering and Formal Methods (SEFM), pp 21–30

Rushby J (2006b) Tutorial: Automated formal methods with PVS, SAL, and Yices. In: Software Engineering and Formal Methods (SEFM), p 262

Sheini H, Sakallah K (2006) From Propositional Satisfiability to Satisfiability Modulo Theories. In: Theory and Applications of Satisfiability Testing (SAT), pp 1–9

Sheini HM, Sakallah KA (2005) A scalable method for solving satisfiability of integer linear arithmetic logic. In: Theory and Application of Satisfiability Testing (SAT), pp 241–256

Shostak R (1981) Deciding linear inequalities by computing loop residues. J ACM 28(4):769–779

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

Weber T (2005) A SAT-based Sudoku solver. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Short Paper Proc., pp 11–15

Zantema H, Groote JF (2003) Transforming equality logic to propositional logic. Electr Notes Theor Comput Sci 86(1)

A Full table of benchmarks

The table of all performed benchmarks is given below. Entries of "–(TO)" denote a timeout or exhausted memory resources, entries of "–(E)" are used where the result reported by the solver was wrong. All times are given in seconds.

#	Benchmark	ABsolverDC	ABsolver	CVC 3	MathSAT	yices	Ario	HTP	ExtSAT
1	multiplier_prime_2	0.036	0.040	0.036	0.004	0.000	0.012	0.024	–(E)
2	fischer1-1-fair	0.052	0.120	0.016	0.008	0.004	0.004	0.024	–(E)
3	fischer2-1-fair	0.100	0.492	0.028	0.008	0.008	0.008	0.024	–(E)
4	multiplier_2	0.104	157.518	0.164	0.084	0.004	0.004	0.008	6619.914
5	fischer1-3-fair	0.120	0.124	0.040	0.008	0.008	0.020	0.020	0.028
6	fischer1-2-fair	0.156	0.856	0.028	0.012	0.000	0.016	0.020	0.184
7	fischer1-4-fair	0.156	0.160	0.060	0.012	0.008	0.008	0.024	0.036
8	fischer3-1-fair	0.176	0.988	0.044	0.008	0.008	0.008	0.024	–(E)
9	fischer1-5-fair	0.204	0.200	0.060	0.016	0.004	0.020	0.036	0.048
10	fischer4-1-fair	0.236	1.824	0.068	0.012	0.008	0.024	0.036	–(E)
11	fischer1-6-fair	0.244	0.232	0.088	0.020	0.008	0.024	0.032	0.056
12	fischer2-3-fair	0.248	0.240	0.084	0.020	0.008	0.024	0.044	0.052
13	fischer5-1-fair	0.272	2.580	0.080	0.020	0.008	0.028	0.044	–(E)
14	simplebitadder_compose_2	0.304	195.756	0.088	0.056	0.004	0.004	0.028	–(TO)
15	fischer2-4-fair	0.324	0.328	0.152	0.024	0.012	0.040	0.068	0.080
16	fischer6-1-fair	0.328	3.716	0.104	0.016	0.012	0.036	0.056	–(TO)
17	fischer3-3-fair	0.356	0.360	0.172	0.028	0.012	0.044	0.072	0.104
18	multiplier_3	0.412	–(TO)	3.788	0.380	0.008	0.012	0.028	–(TO)
19	fischer7-1-fair	0.416	4.940	0.112	0.024	0.012	0.040	0.068	–(TO)
20	fischer3-4-fair	0.496	0.496	0.284	0.028	0.012	0.060	0.124	0.176
21	fischer4-3-fair	0.500	0.492	0.244	0.040	0.016	0.060	0.120	0.148
22	fischer8-1-fair	0.528	6.936	0.168	0.028	0.012	0.048	0.084	–(TO)
23	fischer9-1-fair	0.556	9.001	0.152	0.032	0.012	0.060	0.104	–(TO)
24	fischer10-1-fair	0.620	11.061	0.176	0.036	0.012	0.072	0.116	–(TO)
25	fischer5-3-fair	0.620	0.632	0.324	0.048	0.024	0.080	0.176	0.216
26	fischer4-4-fair	0.668	0.656	0.440	0.056	0.020	0.088	0.180	0.236
27	fischer11-1-fair	0.708	14.205	0.204	0.048	0.020	0.088	0.148	–(TO)
28	fischer6-3-fair	0.816	0.804	0.452	0.060	0.040	0.116	0.268	0.300
29	fischer7-3-fair	0.884	0.920	0.524	0.072	0.032	0.160	0.364	0.484
30	fischer5-4-fair	0.900	0.848	0.644	0.064	0.036	0.140	0.296	0.368
31	fischer8-3-fair	1.040	1.036	0.644	0.084	0.036	0.200	0.520	0.584
32	fischer6-4-fair	1.120	1.096	0.896	0.100	0.040	0.184	0.456	0.504
33	fischer2-5-fair	1.200	4.764	0.180	0.028	0.012	0.052	0.092	–(TO)
34	fischer9-3-fair	1.216	1.204	0.756	0.088	0.048	0.252	0.820	0.884
35	fischer7-4-fair	1.224	1.220	1.312	0.100	0.040	0.264	0.620	0.768
36	fischer2-2-fair	1.256	482.574	0.060	0.024	0.008	0.024	0.032	12.173
37	fischer10-3-fair	1.336	1.324	0.852	0.116	0.044	0.328	–(TO)	0.912
38	fischer8-4-fair	1.416	1.436	2.052	0.128	0.056	0.328	1.136	1.112
39	simplebitadder_compose_3	1.436	–(TO)	0.724	0.296	0.004	0.012	0.060	–(TO)
40	fischer11-3-fair	1.484	1.496	1.112	0.132	0.060	0.376	1.308	1.172
41	fischer9-4-fair	1.624	1.604	2.716	0.136	0.076	0.444	1.508	1.624
42	fischer10-4-fair	1.848	1.872	3.428	0.168	0.084	0.544	1.948	2.164
43	multiplier_4	1.892	–(TO)	–(TO)	2.008	0.024	0.032	0.100	–(TO)
44	fischer11-4-fair	2.044	2.060	7.376	0.192	0.080	0.624	3.116	2.860
45	fischer3-5-fair	4.604	12.913	0.336	0.048	0.024	0.084	0.212	–(TO)
46	multiplier_prime_3	5.528	5.908	0.060	0.012	0.004	0.008	0.016	–(E)
47	multiplier_prime_4	6.900	14.053	7.496	0.012	0.004	0.008	0.028	–(E)
48	simplebitadder_compose_4	6.900	–(TO)	5.476	1.316	0.016	0.020	0.136	–(TO)
49	fischer4-5-fair	7.424	25.250	0.732	0.072	0.028	0.148	0.324	–(TO)
50	multiplier_5	9.153	–(TO)	–(TO)	9.577	0.100	0.176	0.372	–(TO)
51	multiplier_prime_5	9.469	12.241	0.552	0.012	0.000	0.004	0.032	–(TO)
52	multiplier_prime_6	12.237	171.859	–(TO)	0.024	0.000	0.004	0.024	–(TO)
53	multiplier_prime_7	14.853	742.854	5.236	0.028	0.000	0.000	0.036	–(TO)
54	fischer3-2-fair	15.221	–(TO)	0.100	0.020	0.008	0.032	0.048	2856.795
55	multiplier_prime_8	16.841	–(TO)	–(TO)	0.076	0.004	0.016	0.036	–(TO)

56	fischer5-5-fair	17.157	45.455	1.548	0.104	0.040	0.248	0.544	–(TO)
57	multiplier_prime_9	20.921	–(TO)	–(TO)	0.040	0.004	0.004	0.032	–(TO)
58	multiplier_prime_10	25.126	–(TO)	–(TO)	0.052	0.012	0.012	0.044	–(TO)
59	multiplier_prime_11	25.574	–(TO)	–(TO)	0.116	0.008	0.016	0.044	–(TO)
60	fischer6-5-fair	26.138	82.229	2.132	0.116	0.044	0.344	0.672	–(TO)
61	fischer7-5-fair	26.578	111.559	3.540	0.140	0.060	0.532	1.216	–(TO)
62	multiplier_prime_12	31.082	–(TO)	–(TO)	0.088	0.004	0.008	0.064	–(TO)
63	multiplier_prime_13	33.234	–(TO)	161.858	0.072	0.004	0.016	0.064	–(TO)
64	simplebitadder_compose_5	34.210	–(TO)	43.931	6.140	0.064	0.040	0.508	–(TO)
65	multiplier_prime_14	35.250	–(TO)	–(TO)	0.108	0.008	0.020	0.064	–(TO)
66	fischer8-5-fair	41.207	146.389	5.664	0.164	0.068	0.836	1.788	–(TO)
67	multiplier_6	46.811	–(TO)	–(TO)	50.187	0.472	0.852	2.084	–(TO)
68	fischer9-5-fair	53.703	220.246	10.465	0.200	0.080	0.836	2.716	–(TO)
69	fischer10-5-fair	1m12.625	267.165	12.005	0.240	0.096	1.184	3.724	–(TO)
70	simplebitadder_compose_6	176.495	–(TO)	117.283	30.162	0.280	0.076	2.984	–(TO)
71	fischer4-2-fair	209.013	–(TO)	0.132	0.032	0.012	0.048	0.056	–(TO)
72	multiplier_7	309.223	–(TO)	–(TO)	254.708	2.812	5.468	20.701	–(TO)
73	simplebitadder_compose_7	1140.939	–(TO)	–(TO)	149.673	1.528	0.172	24.750	–(TO)
74	fischer2-6-fair	1341.424	–(TO)	0.280	0.040	0.016	0.072	0.144	–(TO)
75	multiplier_8	3050.831	–(TO)	–(TO)	1256.907	28.086	55.803	219.554	–(TO)
76	fischer5-2-fair	3280.273	–(TO)	0.180	0.036	0.012	0.056	0.084	–(TO)
77	fischer10-2-fair	–(TO)	–(TO)	0.428	0.088	0.024	0.228	0.400	–(TO)
78	fischer11-2-fair	–(TO)	–(TO)	0.496	0.096	0.036	0.292	0.456	–(TO)
79	fischer2-7-fair	–(TO)	–(TO)	0.296	0.064	0.020	0.112	0.220	–(TO)
80	fischer3-6-fair	–(TO)	–(TO)	0.752	0.064	0.032	0.160	0.308	–(TO)
81	fischer3-7-fair	–(TO)	–(TO)	1.128	0.128	0.036	0.252	0.680	–(TO)
82	fischer3-8-fair	–(TO)	–(TO)	1.992	0.284	0.036	0.432	0.732	–(TO)
83	fischer4-6-fair	–(TO)	–(TO)	1.384	0.096	0.040	0.324	0.484	–(TO)
84	fischer6-2-fair	–(TO)	–(TO)	0.236	0.052	0.020	0.080	0.148	–(TO)
85	fischer6-6-fair	–(TO)	–(TO)	6.584	0.160	0.064	0.728	1.256	–(TO)
86	fischer6-7-fair	–(TO)	–(TO)	13.441	0.328	0.100	1.956	2.432	–(TO)
87	fischer6-8-fair	–(TO)	–(TO)	21.865	0.692	0.132	8.181	4.480	–(TO)
88	fischer6-9-fair	–(TO)	–(TO)	44.979	2.368	0.200	40.891	7.752	–(TO)
89	fischer7-2-fair	–(TO)	–(TO)	0.296	0.056	0.020	0.100	0.180	–(TO)
90	fischer7-6-fair	–(TO)	–(TO)	8.601	0.196	0.084	1.348	2.188	–(TO)
91	fischer7-7-fair	–(TO)	–(TO)	18.773	0.436	0.120	4.536	4.544	–(TO)
92	fischer8-2-fair	–(TO)	–(TO)	0.360	0.064	0.024	0.136	0.248	–(TO)
93	fischer8-6-fair	–(TO)	–(TO)	11.189	0.244	0.100	1.424	3.960	–(TO)
94	fischer9-2-fair	–(TO)	–(TO)	0.436	0.064	0.028	0.192	0.308	–(TO)
95	multiplier_10	–(TO)	–(TO)	–(TO)	–(TO)	2436.764	6610.073	–(TO)	–(TO)
96	multiplier_11	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)
97	multiplier_12	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)
98	multiplier_13	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)
99	multiplier_14	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)
100	multiplier_9	–(TO)	–(TO)	–(TO)	–(TO)	314.944	468.821	1821.458	–(TO)
101	simplebitadder_compose_10	–(TO)	–(TO)	–(TO)	–(TO)	1301.677	11.709	–(TO)	–(TO)
102	simplebitadder_compose_11	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	25.174	–(TO)	–(TO)
103	simplebitadder_compose_12	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	67.488	–(TO)	–(TO)
104	simplebitadder_compose_13	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	147.345	–(TO)	–(TO)
105	simplebitadder_compose_14	–(TO)	–(TO)	–(TO)	–(TO)	–(TO)	774.620	–(TO)	–(TO)
106	simplebitadder_compose_8	–(TO)	–(TO)	–(TO)	925.098	18.845	0.480	195.356	–(TO)
107	simplebitadder_compose_9	–(TO)	–(TO)	–(TO)	–(TO)	258.396	1.188	2161.831	–(TO)