Interval Constraint Solving Using Propositional SAT Solving Techniques

Martin Fränzle[1], Christian Herde[1], Stefan Ratschan[2], Tobias Schubert[3], and Tino Teige[1⋆]

[1] Dept. of CS, Carl von Ossietzky Universität Oldenburg, Germany
{fraenzle|herde|teige}@informatik.uni-oldenburg.de

[2] Inst. of CS, Academy of Sciences of the Czech Republic, Prague
stefan.ratschan@cs.cas.cz

[3] FAW, Albert-Ludwigs-Universität Freiburg, Germany
schubert@informatik.uni-freiburg.de

Abstract. In order to facilitate automated reasoning about large Boolean combinations of non-linear arithmetic constraints involving transcendental functions, we extend the paradigm of lazy theorem proving to interval-based arithmetic constraint solving. Algorithmically, our approach deviates substantially from "classical" lazy theorem proving approaches in that it directly controls arithmetic constraint propagation from the SAT solver rather than completely delegating arithmetic decisions to a subordinate solver. From the constraint solving perspective, it extends interval-based constraint solving with all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving, like conflict-driven learning combined with non-chronological backtracking.

1 Introduction

Within many application domains, e.g. the analysis of programs involving arithmetic operations or the analysis of hybrid discrete-continuous systems, one faces the problem of solving large Boolean combinations of non-linear arithmetic constraints over the reals, where solving means to find a satisfying valuation or to prove nonexistence thereof. This gives rise to a plethora of problems, in particular (a) how to effectively and efficiently solve conjunctive combinations of constraints in the in general undecidable domain of non-linear constraints involving transcendental functions, and (b) how to efficiently maneuver the large search spaces arising from the rich Boolean structure of the overall formula.

While promising solutions for these two individual sub-problems exist, it seems that their combination has hardly been attacked. Arithmetic constraint solving based on interval constraint propagation [5, 4, 3], on the one hand, has

⋆ This work has been partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

proven to be an efficient means for solving robust combinations of otherwise undecidable arithmetic constraints [22]. Here, robustness means that the constraints maintain their truth value under small perturbations of the constants in the constraints. Modern SAT solvers, on the other hand, can efficiently find satisfying valuations of very large propositional formulae (e.g., [20, 17]), as well as —using the lazy theorem proving paradigm— of complex propositional combinations of atoms from various decidable theories (e.g., [10, 8, 1, 9]).

Within this paper, we describe a tight integration of the paradigm of lazy theorem proving with interval-based arithmetic constraint solving, thus providing a lazy theorem proving approach that reasons over the undecidable arithmetic domain of Boolean combinations of non-linear constraints involving transcendental functions. Within our approach, a DPLL-based propositional satisfiability solver traverses the proof tree originating from the Boolean structure of the constraint formula, as is characteristic for the lazy theorem proving approach. Yet, in contrast to traditional lazy theorem proving approaches ranging over some *decidable* theory T, we do not pass a corresponding conjunctive constraint system over the respective theory T to a subordinate decision procedure serving as an oracle for consistency of the constraint set. Instead, we exploit the algorithmic similarities between DPLL-based propositional SAT solving and constraint solving based on constraint propagation for a much tighter integration, where the DPLL solver has full introspection in and direct control over constraint propagation within the theory T rather than completely delegating theory-related decisions to a subordinate solver. This tight integration has a number of advantages. First, by sharing the common core of the search algorithms between the propositional and the theory-related, interval-constraint-propagation-based part of the solver, we are able to transfer algorithmic enhancements from one domain to the other: in particular, we thus equip interval-based constraint solving with all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving, like non-chronological backtracking and conflict-driven learning. Second, the introspection into the constraint propagation process allows fine-granular control over the necessarily incomplete arithmetic deduction process, thus enabling a stringent extension of lazy theorem proving to an undecidable theory. Finally, due to the availability of learning, we are able to implement an almost lossless restart mechanisms within an interval-based arithmetic constraint propagation framework, thus being able to substantially accelerate incremental proof searches, where the individual constraint propagations are depth-constrained, yet incrementally less depth-constrained proof searches are iterated until a solution is found (or absence of such is proved). Such iteration is essential to quasi-completeness, i.e. termination on all constraint formulas that are robust in the sense that their truth value does not change under some small variation of constants [22].

Structure of the paper. We start our exposition in Section 2 with a description of the syntactic structure and the semantics of the arithmetic satisfiability problems we are going to address. Section 3 provides a brief introduction to the technologies that our development builds on. Thereafter, we provide a detailed

explanation of our new algorithm (Section 4) and benchmark results (Section 5). We conclude with an overview over ongoing work and planned extensions.

2 Logics

Aiming at automated analysis of programs operating over the reals, e.g. bounded model checking of hybrid systems, our constraint solver addresses satisfiability of non-linear arithmetic constraints over real-valued variables plus Boolean variables for encoding the control flow. The user thus may input constraint formulae built from quantifier-free constraints over the reals and from propositional variables using arbitrary Boolean connectives. The atomic real-valued constraints are relations between potentially non-linear terms involving transcendental functions, like $\sin(x + \omega t) + ye^{-t} \leq z + 5$. By the front-end of our constraint solver, these constraint formulae are rewritten to quantifier-free constraints in conjunctive normal form, with atomic propositions ranging over propositional variables and arithmetic constraints confined to a form resembling three-adress code (cf. the *primitive constraints* of interval constraint propagation, see Section 3). Thus, the *internal* syntax of constraint formulae is as follows:

$$
\begin{aligned}
\mathit{formula} &::= \{\mathit{clause} \wedge\}^* \mathit{clause} \\
\mathit{clause} &::= (\{\mathit{bound} \vee\}^* \mathit{bound}) \mid (\mathit{bound} \vee \mathit{equation}) \\
\mathit{bound} &::= \mathit{variable} \geq \mathit{rational\_const} \mid \mathit{variable} > \mathit{rational\_const} \\
&\quad \mid \mathit{variable} < \mathit{rational\_const} \mid \mathit{variable} \leq \mathit{rational\_const} \\
\mathit{variable} &::= \mathit{real\_variable} \mid \mathit{boolean\_variable} \\
\mathit{equation} &::= \mathit{triplet} \mid \mathit{pair} \\
\mathit{triplet} &::= \mathit{real\_variable} = \mathit{real\_variable}\ \mathit{bop}\ \mathit{real\_variable} \\
\mathit{pair} &::= \mathit{real\_variable} = \mathit{uop}\ \mathit{real\_variable} \\
\mathit{bop} &::= + \mid - \mid * \mid / \mid \ldots \\
\mathit{uop} &::= - \mid \sin \mid \exp \mid \ldots
\end{aligned}
$$

Such constraint formulae are interpreted over valuations $\sigma \in (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R})$, where BV is the set of Boolean and RV the set of real-valued variables. $\mathbb{B}$ is identified with the subset $\{0, 1\}$ of $\mathbb{R}$ such that any rational-valued bound on a Boolean variable v corresponds to a literal v or $\neg v$. The definition of satisfaction is standard: a constraint formula ϕ is satisfied by a valuation iff all its clauses are satisfied, i.e. iff at least one atom is satisfied in any clause, where the term *atom* refers to both bounds and equations. Satisfaction of atoms is wrt. the standard interpretation of the arithmetic operators and ordering relations over the reals. We assume all arithmetic operators to be total and therefore extend their codomain (as well as, for compositionality, their domain) with a special value $\mho \notin \mathbb{R}$. It is understood that $\mho$ does not satisfy any inequation, i.e. $\mho \not\sim c$ for any constant c and any relation $\sim$.

Instead of real-valued valuations of variables, our constraint solving algorithm manipulates interval-valued valuations $\rho \in (BV \xrightarrow{\text{total}} \mathbb{I}_{\mathbb{B}}) \times (RV \xrightarrow{\text{total}} \mathbb{I}_{\mathbb{R}})$, where

$\mathbb{I}_{\mathbb{B}} = 2^{\mathbb{B}} \setminus \emptyset$ and $\mathbb{I}_{\mathbb{R}}$ is the set of non-empty convex subsets of $\mathbb{R}$.[4] Slightly abusing notation, we write $\rho(l)$ for $\rho_{\mathbb{I}_{\mathbb{B}}}(l)$ when $\rho = (\rho_{\mathbb{I}_{\mathbb{B}}}, \rho_{\mathbb{I}_{\mathbb{R}}})$ and $l \in BV$, and similarly $\rho(x)$ for $\rho_{\mathbb{I}_{\mathbb{R}}}(x)$ when $x \in RV$. If both σ and η are interval assignments then σ is called a *refinement* of η iff $\sigma(v) \subseteq \eta(v)$ for each $v \in BV \cup RV$. We call an interval valuation ρ *weakly satisfying* for a constraint formula ϕ iff each clause of ϕ contains at least one weakly satisfied atom. Weak satisfaction of atoms is defined as follows:

$$\begin{array}{lll} \rho \models_{\mathrm{w}} x \sim c & \text{iff } \rho(x) \subseteq \{u \mid u \in \mathbb{R}, u \sim c\} & \text{for } x \in RV \cup BV, c \in \mathbb{Q} \\ \rho \models_{\mathrm{w}} x = y \circ z & \text{iff } \rho(x) \supseteq \rho(y) \hat{\circ} \rho(z) & \text{for } x, y, z \in RV, \circ \in bop \\ \rho \models_{\mathrm{w}} x = \circ y & \text{iff } \rho(x) \supseteq \hat{\circ} \rho(y) & \text{for } x, y \in RV, \circ \in uop \end{array}$$

where $\hat{\circ}$ is a conservative interval extension of operation $\circ$, i.e. satisfies $i_1 \hat{\circ} i_2 \supseteq \{x \circ y \mid x \in i_1, y \in i_2\}$ for all intervals i_1 and i_2 [19]. Note that equality is interpreted as set inclusion rather than set equality in the interval interpretation. This is motivated by the fact that under appropriate side-conditions, inclusion is sufficient for deciding real-valued (un-)satisfiability, as expressed in Lemma 1 below. In order to formalize these side conditions, we call ρ *strongly satisfying* for ϕ, denoted $\rho \models_{\mathrm{s}} \phi$, iff there is a finite family $a_1, \ldots, a_n$ of atoms such that the following three conditions hold:

1. Each clause in ϕ contains at least one atom a that *matches* some atom a_i in $\{a_1, \ldots, a_n\}$ in the following sense: a_i is a reshuffling of a obtained by using partial inverses of the operation entailed in a, e.g. a being $x = y + z$ and a_i being $z = x - y$. Note that such a reshuffling has no impact on real-valued satisfiability, yet yields additional freedom for interval satisfaction, as it allows to reorder the directions of the set inclusions.
2. The atoms a_1 to a_n are weakly satisfied by ρ.
3. If a_i is a triplet $x = y \circ z$ or a pair $x = \circ y$ then x is interpreted by a point interval (i.e., $|\rho(x)| = 1$), or x does neither occur in any a_j with $j > i$ nor on the right-hand side of a_i (i.e., $x \neq y$ and $x \neq z$).

Due to the ordering condition on equality constraints that are satisfied by non-point intervals, we obtain the following tight correspondence between strong interval satisfiability and real-valued satisfiability:

Lemma 1. *Assume that the interval extensions of the operations are tight on point-interval arguments, i.e. that* $\{a\} \hat{\circ} \{b\} = \{a \circ b\}$ *for each* $a, b \in \mathbb{R}$ *and each binary operation* $\circ$*, and analogously for unary operations. Then*

1. *If* $\sigma \models \phi$ *for some real-valued valuation* σ *then* $\rho \models_{\mathrm{s}} \phi$ *for some interval valuation* ρ *with* $\forall v \in BV \cup RV . \sigma(v) \in \rho(v)$.
2. *If* $\rho \models_{\mathrm{s}} \phi$ *then there exists a real-valued valuation* σ *such that* $\sigma \models \phi$ *and* $\forall v \in BV \cup RV . \sigma(v) \in \rho(v)$.

[4] Note that this definition covers the open, half-open, and closed non-empty intervals over $\mathbb{R}$, including unbounded intervals.

Proof. 1. Take $\rho(v) = \{\sigma(v)\}$ for each $v \in BV \cup RV$. Due to tightness of the interval extensions on point intervals, $\sigma \models \phi$ implies $\rho \models_{\mathrm{w}} \phi$, which in turn implies $\rho \models_{\mathrm{s}} \phi$ due to all intervals assigned by ρ being point intervals such that strong and weak satisfaction coincide.

2. If $\rho \models_{\mathrm{s}} \phi$ then we can recursively define a real-valued valuation σ exploiting the structure of the family of witnesses a_i as follows:
   (a) For each $v \in BV$ and for each $v \in RV$ not occurring on the left hand side of any equation in $\{a_1, \ldots, a_n\}$, select $\sigma(v) \in \rho(v)$ arbitrarily;
   (b) For $i = n$ downto 1, process the constraints a_n to a_1 in reverse sequence as follows:
      i. if a_i is a triplet $v = x \circ y$ then take $\sigma(v) = \sigma(x) \circ \sigma(y)$,
      ii. if a_i is a pair $v = \circ x$ then take $\sigma(v) = \circ\sigma(x)$.

   Note that solutions to the equation system in (b) do exist because the hierarchical order of variable dependencies in a_1 to a_n enforces that each $\sigma(v)$ either is subject to at most one defining equation or is picked from a point interval $\rho(v) \supseteq \rho(x)\hat{\circ}\rho(y)$ or $\rho(v) \supseteq \hat{\circ}\rho(y)$, respectively. Furthermore, $\sigma(v) \neq \mho$ as $\mho \notin \rho(v) \supseteq \rho(x)\hat{\circ}\rho(y) \ni \sigma(x) \circ \sigma(y)$ and $\mho \notin \rho(v) \supseteq \hat{\circ}\rho(y) \ni \circ\sigma(y)$, respectively. It is straightforward to check that $\sigma \models \phi$. □

When solving satisfiability problems of formulae with Davis-Putnam-like procedures, we will build interval valuations incrementally by successively contracting intervals.

We say that an interval valuation ρ is *weakly (or strongly) consistent with a formula (or clause or atom)* ϕ iff there exists a refinement η of ρ that weakly (strongly, resp.) satisfies ϕ. Otherwise, we call ρ *weakly (strongly, resp.) inconsistent with* ϕ. Note that deciding weak consistency of an interval valuation with a single bound is straightforward, as is deciding weak satisfaction of an arbitrary atom. If ρ is neither weakly satisfying for ϕ nor weakly inconsistent with ϕ then we call ϕ *inconclusive on* ρ.

3 Algorithmic Basis

Our constraint solving approach builds upon the well-known techniques of interval constraint propagation, propositional SAT solving by the DPLL procedure plus its more recent algorithmic enhancements, and lazy theorem proving.

Interval constraint propagation (ICP) is one of the sub-topics of the area of constraint programming where constraint propagation techniques are studied in various, and often discrete, domains. For the domain of the real numbers, given a constraint ϕ and a floating-point box B, so-called *contractors* compute another floating-point box $C(\phi, B)$ such that $C(\phi, B) \subseteq B$ and such that $C(\phi, B)$ contains all solutions of ϕ in B (cf. the notion of *narrowing operator* [2]).

There are several methods for implementing such contractors. The most basic method [5, 4] decomposes all atomic constraints (i.e., constraints of the form $t \geq 0$ or $t = 0$, where t is a term) into conjunctions of so-called primitive constraints (i.e., constraints such as $x + y = z$, $xy = z$, $z \in [\underline{a}, \overline{a}]$, or $z \geq 0$)

by introducing additional auxiliary variables (e.g., decomposing $x + \sin y \geq 0$ to $\sin y = v_1 \wedge x + v_1 = v_2 \wedge v_2 \geq 0$). Then it applies a contractor for these primitive constraints until a fixpoint is reached.

We illustrate contractors for primitive constraints using the example of a primitive constraint $x + y = z$ with the intervals $[1, 4]$, $[2, 3]$, and $[0, 5]$ for x, y, and z, respectively: We can solve the primitive constraint for each of the free variables, arriving at $x = z - y$, $y = z - x$, and $z = x + y$. Each of these forms allows us to contract the interval associated with the variable on the left-hand side of the equation: Using the first solved form we subtract the interval $[2, 3]$ for y from the interval $[0, 5]$ for z, concluding that x can only be in $[-3, 3]$. Intersecting this interval with the original interval $[1, 4]$, we know that x can only be in $[1, 3]$. Proceeding in a similar way for the solved form $y = z - x$ does not change any interval, and finally, using the solved form $z = x + y$, we can conclude that z can only be in $[3, 5]$.

Contractors for other primitive constraints can be based on interval arithmetic in a similar way. There is extensive literature [21, 13] providing precise formulae for interval arithmetic for addition, subtraction, multiplication, division, and the most common transcendental functions. The floating point results are always rounded outwards, such that the result remains correct also under rounding errors. There are several variants, alternatives and improvements of the basic approach described above, e.g. [14, 2, 18, 12, 16].

Propositional SAT solving. The *Propositional Satisfiability Problem* (SAT) is a well-known NP-complete problem, with extensive applications in various fields of computer science and engineering. In recent years a lot of developments in creating powerful SAT algorithms have been made, leading to state-of-the-art approaches like BerkMin [11], Mira [17], and zChaff [20]. All of them are enhanced variants of the classical backtrack search DPLL procedure [6, 7]. In contrast to local search strategies only such complete algorithms are able to prove the unsatisfiability of a problem instance, which is often the final objective in many applications, e.g. circuit verification and automated theorem proving.

Given a Boolean formula ϕ in *Conjunctive Normal Form* (CNF) and a partial valuation ρ, which is empty at the beginning of the search process, a backtrack search algorithm incrementally extends ρ until either $\rho \models \phi$ holds or ρ turns out to be inconsistent for ϕ. In the latter case another extension of ρ is tried through backtracking.

Extensions are constructed by performing *decision steps*, which entail selecting an unassigned variable and assigning a value to it. Since the days of the original DPLL procedure many variable selection strategies have been introduced, among them the *Variable State Independent Decaying Sum* (VSIDS) heuristic from zChaff [20]. The main idea of VSIDS is to prefer those variables that often occur in recently deduced *conflict clauses*.

Each decision step is followed by the *deduction phase*, involving the search for *unit clauses*, i.e. clauses that have only one unassigned literal left while all other literals are assigned incorrectly in the actual valuation ρ. Obviously, unit clauses require certain assignments in order to preserve their satisfiability, where

the execution of the implied assignments itself might force further assignments. In the context of SAT solving such necessary assignments are also referred to as *implications*. To perform the deduction phase in an efficient manner zChaff introduced a lazy clause evaluation technique based on *Watched Literals* (WL): for each clause two literals are selected in such a way, that they either are both unassigned or at least one of them is satisfying the clause. So, if at some point during the search one of the WLs is getting assigned incorrectly, a new WL for the corresponding clause has to be found. If such a literal does not exist and the second WL is still unassigned, the clause is forcing an implication. As a consequence of this method there is no need to check all clauses after making a decision step, but only those ones, where a WL is getting assigned incorrectly.

However, deduction may also yield a *conflicting clause* which has all its literals assigned false, indicating the need for backtracking. In order to avoid repeating the same unsatisfying valuation ρ multiple times, modern SAT algorithms incorporate *conflict-driven learning* to derive a sufficiently general reason (a combination of variable assignments) for the actual conflict. Based on that ideally minimal number of assignments that triggered the particular conflict, a *conflict clause* is generated and added to the clause set to guide the subsequent search. Additionally, the conflict clause is used to compute the backtrack level, which is defined as the maximum level the SAT algorithm has to backtrack to in order to solve the conflict. This approach often leads to a *non-chronological backtracking* operation, jumping back more than just one level and making conflict-driven learning combined with non-chronological backtracking a powerful mechanism to prune large parts of the search space [24].

4 Integrating interval constraint propagation and SAT

By combining interval constraint propagation with an interval splitting scheme to obtain a branch-and-prune algorithm, as shown on the left of Table 1, a constraint solving algorithm for constraints over the reals incorporating transcendental functions can be achieved being based on interval splitting over real-valued intervals as a branching step and on ICP as a forward inference step, it does closely resemble the core algorithm of DPLL SAT solving. In fact, DPLL-SAT can be viewed as its counterpart over the Boolean intervals $\mathbb{I}_{\mathbb{B}}$, where again interval splitting is the decision step and Boolean constraint propagation in the form of unit propagation provides the forward inference mechanism, cf. right-hand side of Table 1. This similarity motivates a tighter integration of propositional SAT and arithmetic reasoning than in classical lazy theorem proving, cf. Fig. 1. This tight integration shares the common algorithmic parts, thereby providing the SAT solver with full control over and full introspection into the ICP process. This way, recent algorithmic enhancements of propositional SAT solving, like lazy clause evaluation, conflict-driven learning, and non-chronological backjumping carry over to ICP-based arithmetic constraint solving. In particular, we are able to learn forms of conflicts that are considerably more general than

	Interval constraint solving	DPLL SAT
Given:	Constraint set $C = \{c_1, \ldots, c_n\}$, initial box $B \subseteq \mathbb{R}^{\lvert\text{free}(C)\rvert}$	Clause set $C = \{c_1, \ldots, c_n\}$, initial box $B \subseteq \mathbb{B}^{\lvert\text{free}(C)\rvert}$
Goal:	Find box $B' \subseteq B$ containing satisfying valuations throughout *or* show non-existence of such B'.	
Alg.: 1.	$L := \{B\}$	$L := \{B\}$
2.	If $L \neq \emptyset$ then take some box $b :\in L$, otherwise report "unsatisfiable" and stop.	If $L \neq \emptyset$ then take most recently added box $b :\in L$, otherwise report "unsatisfiable" and stop.
3.	Use contractor C to determine subbox $b' \subseteq b$ containing all solutions in b.	Use unit propagation to determine subbox $b' \subseteq b$ containing all solutions in b.
4.	If $b' = \emptyset$ then set $L := L \setminus \{b\}$, goto 2.	If $b' = \emptyset$ then set $L := L \setminus \{b\}$, goto 2.
5.	Check whether b' strongly satisfies all constraints in C; if so then report b' as satisfying and stop.	Check whether all clauses in C are satisfied throughout b'; if so then report b' as satisfying and stop.
6.	If $b' \subset b$ then set $L := L \setminus \{b\} \cup \{b'\}$, goto 2.	If $b' \subset b$ then set $L := L \setminus \{b\} \cup \{b'\}$, goto 2.
7.	Split b into subintervals b_1 and b_2, set $L := L \setminus \{b\} \cup \{b_1, b_2\}$, goto 2.	Split b into subintervals b_1 and b_2, set $L := L \setminus \{b\} \cup \{b_1, b_2\}$, goto 2.

Table 1. Interval constraint solving (left) vs. basic DPLL SAT (right).

classical as well as generalized nogoods [15] in search procedures for constraint solving.

Interval constraint solving as a multi-valued SAT problem. The underlying idea of our algorithm is that the two central operations of ICP-based arithmetic constraint solving —interval contraction by constraint propagation and by interval splitting— correspond to asserting bounds on real-valued variables $v \sim c$ with $v \in RV$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{Q}$. Likewise, the decision steps and unit propagations in DPLL proof search correspond to asserting literals. A unified DPLL- and ICP-based proof search on a formula ϕ from the formula language of Sect. 2 can thus be based on asserting or retracting atoms of the formula language, thereby in lockstep refining or widening an interval valuation ρ that represents the current set of candidate solutions:

1. Proof search on ϕ starts with an empty set of asserted atoms and the interval valuation ρ being the minimal element wrt. the refinement relation on interval valuations, i.e. all intervals being maximal ($\{\texttt{false}, \texttt{true}\}$ for Boolean variables and $\mathbb{R}$ or —if the variable has a bounded range— a maximal subrange thereof for real-valued variables) .
2. It continues with searching for *unit clauses* in ϕ, i.e. clauses that have only one inconclusive (on ρ) atom left and all other atoms being weakly inconsistent with the current interval valuation ρ. If such a clause is found then its unique unassigned atom is asserted. The asserted atom stems from the formula ϕ or some learned nogood and may thus be an arbitrary bound, triplet, or pair.

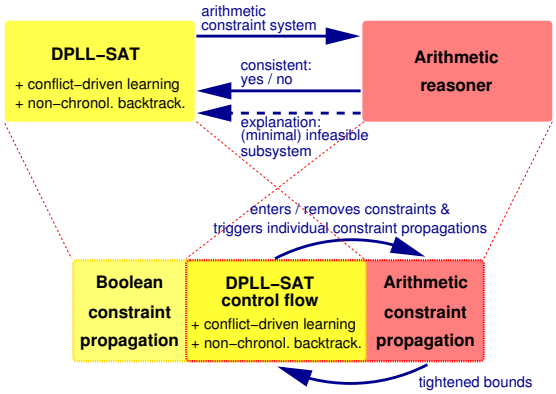

Fig. 1. Classical lazy theorem proving (top) vs. iSAT's tight integration of interval constraint solving and propositional SAT (bottom)

Step 2 is repeated until all unit clauses have been processed.

3. If there is an asserted atom a that is not weakly satisfied by the current interval valuation ρ then the contractors corresponding to a are applied to ρ. In the case of triplets and pairs, these contractors are the usual contractors for the primitive constraints of ICP, as explained in Sect. 3. For bounds $a = v \sim c$, contraction amounts to replacing $\rho(v)$ with $\rho(v) \cap \{x \in \mathbb{R} \mid x \sim c\}$, no matter whether they are literals or bounds on real-valued variables. In case of triplets and pairs, the contractions obtained are in turn asserted as bounds (this is redundant for contractions stemming from bounds, as the asserted atoms would be equal to the already asserted bound which effected the contraction).
   This step is repeated until no further contraction is obtained,[5] or until contraction detects a conflict in the sense of some interval $\rho(v)$ becoming empty. In case of a conflict, some previous splits (cf. step 4) have to be reverted, which is achieved by backtracking —thereby undoing all assertions being consequences of the split— and by asserting the complement of the previous split. Furthermore, a reason for the conflict can be recorded as a nogood, thus pruning the remaining search space (see below).
   If no conflict arose then, if new unit clauses resulted from the contraction, the algorithm continues at step 2, otherwise at 4.
4. The algorithm checks whether ρ strongly satisfies ϕ and stops with result "satisfiable" if so, as this implies real-valued satisfiability by Lemma 1. Otherwise, it applies a *splitting step*: it selects a variable $v \in BV \cup RV$ that is interpreted by a non-point interval (i.e., $|\rho(v)| > 1$) and splits its interval $\rho(v)$ by asserting a bound that contains v as a free variable and which is inconclusive on ρ.[6] Thereafter, the algorithm continues at 2. If no such vari-

[5] In practice, one stops as soon as the changes become negligible.

[6] Note that the complement of such an assertion also is a bound and is inconclusive on ρ too.

able v exists then the search space has been exhausted and the algorithm stops with result "unsatisfiable".

By its similarity to DPLL algorithms, this base algorithms lends itself to all the algorithmic enhancements and sophisticated data structures that were instrumental to the impressive recent gains in propositional SAT solver performance.

Lazy clause evaluation. In order to save costly visits to and evaluations of disjunctive clauses, we employ the lazy clause evaluation scheme of zChaff [20] to our more general class of atoms as follows: within each clause, we select two atoms which are inconclusive wrt. the current valuation ρ, called the "watched atoms" of the clause. Instead of scanning the whole clause set for unit clauses in step 2 of the base algorithm, we do only visit the clause if a free variable of one of its two watched atoms is contracted, i.e. a bound assigning a tighter bound is asserted. In this case, we evaluate the atoms truth value. If found to be inconsistent wrt. the new interval assignment, the algorithm tries to substitute the atom by a currently unwatched and not yet inconsistent atom to watch in the future. If this substitution fails due to all remaining atoms in the clause being inconsistent, the clause has become unit and the second watched atom has to be asserted.

Maintaining and compactifying an implication graph. In order to be able to tell reasons for conflicts (i.e., empty interval valuations) encountered, our solver maintains an implication graph akin to that known from propositional SAT solving [24]: all asserted atoms are recorded in a stack-like data structure (unwound upon backtracking, when the assertions are retracted). Within the stack, each assertion not originating from a split, i.e. each assertion a originating from a contraction (including unit propagations), comes equipped with pointers to its antecedents. In this case, a is a bound, i.e. a literal or a real-valued inequation $v \sim c$. The antecedent of a is an atom b containing the variable v plus a set of bounds for the other free variables of b which triggered the contraction a. As ICP often gives rise to long linear chains of contractions originating from mutually contracting via reshuffles of the same constraint, we compactify the implication stack by removing such chains, replacing them by their initial and final chain elements.

Conflict-driven learning and non-chronological backtracking. By following the antecedents of a conflicting assignment, a reason for the conflict can be obtained: reasons correspond to cuts in the antecedent graph, and such reasons can be "learned" for pruning the future search space by adding a *conflict clause* containing the disjunction of the negations of the atoms in the reason. We use the unique implication point technique [24] to derive a conflict clause which is general in that it contains few atoms and which is asserting upon backjumping to the last decision level contributing to the conflict, i.e. upon undoing all decisions and contractions younger than the chronologically youngest decision among the antecedents of the conflict, as shown in Fig. 2.

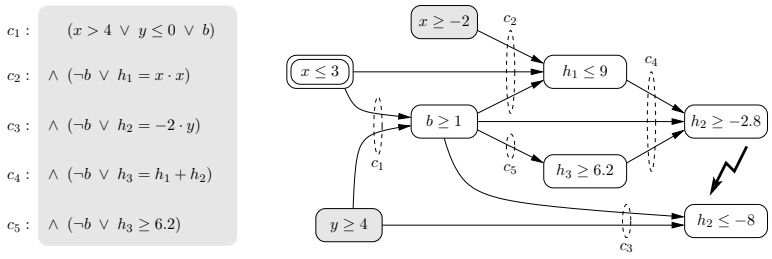

Fig. 2. Conflict analysis: Let the clause set $c_1, \ldots, c_5$ be a fragment of a formula to be solved. Assume $x \geq -2$ and $y \geq 4$ have been asserted on decision levels k_1 and k_2, resp., and another decision level is opened by asserting $x \leq 3$. The resulting implication graph, ending in a conflict on h_2, is shown on the right. Edges relate implications to their antecedents, dashed ellipses indicate the propagating clauses. Following the implication chains from the conflict yields the conflict clause $\neg(x \geq -2) \vee \neg(x \leq 3) \vee \neg(y \geq 4)$ which becomes unit after backjumping to decision level $max(k_1, k_2)$, propagating $x > 3$.

Note that, while adopting the conflict detection techniques from propositional SAT solving, our conflict clauses are still more general than those generated in propositional SAT solving: as the antecedents of a contraction may involve arbitrary atoms, so do the conflict clauses. In order to save us from being forced to generate (and handle in constraint propagation) negated triplets and pairs, we decided to allow those to occur only in the guarded form of a binary clause (*bound* $\vee$ *equation*) in our formulae (cf. syntax in sect. 2). Therefore, we can always replace a negated triplet or pair in a reason by a corresponding bound, to be found among its antecedents. Furthermore, in contrast to nogood learning, we are not confined to learning forbidden combinations of value assignments in the search space, which here would amount to learning disjunctions of interval disequations $x \notin I$ with x being a problem variable and I an interval. Instead, our algorithm may learn arbitrary combinations of atoms $x \sim c$, which provides stronger pruning of the search space: while a nogood $x \notin I$ would only prevent a future visit to any subinterval of I, a bound $x \geq c$, for example, blocks visits to any interval whose left endpoint is at least c, no matter how it is otherwise located relative to the current interval valuation.

Enforcing progress and termination. The naive base algorithm described above would apply unbounded splitting, thus risking nontermination due to the density of the order on $\mathbb{R}$. It traverses the search tree until either no further splits are possible due to the search space being fully covered by conflict clauses or until a strongly satisfying interval interpretation is found. In contrast to purely propositional SAT solving, where the split depth is bounded by the number variables in the SAT problem, this entails the risk of non-termination due to infinite sequences of splits being possible on each real-valued interval. Even worse, by pursuing depth-first search, the algorithm risks infinite descent into a branch of the search tree even if other branches may yield definite, strongly satisfying results.

We tackle this problems by either limiting the splitting width or the maximum number of splits a priori, later on refining it if necessary. I.e., we exclude a variable x from further splitting if the width of its interval $\rho(x)$ falls below a certain threshold δ, or if its predefined number of splits has been exhausted. If no further splitting is possible due to the bound having been reached for each variable, the solver derives a "pseudo-conflict clause" from that situation which —exactly as a true conflict clause would— causes the engine to backtrack and address another part of the search space, thereby abandoning the branch it has investigated previously. Besides guiding backtracking, the pseudo-conflict clause serves as a witness for the existence of an undecided branch, i.e. a branch which has not been fully explored. Note that we can re-open that branch simply by deleting the associated pseudo-conflict clause.

Achieving almost-completeness through restarts. With a given bound δ on split width or a fixed number k of maximum splits, the above procedure may terminate with inconclusive result: none of the visited boxes may be strongly satisfying, yet undecided branches in the search space —corresponding to inconclusive interval interpretations— remain. In this case, the solver simply is restarted with smaller splitting width or number of splits greater than k, respectively. Before restarting, all "pseudo conflicts" are removed from the clause database such that only the "real" conflict clauses are preserved from the previous run. These learned conflict clauses prevent the solver from re-visiting failed boxes such that the restart incurs a very low penalty: essentially, it does only visit those interval interpretations that were previously left in an inconclusive state, and it extends the proof tree precisely at these inconclusive leafs.

By iterating this scheme for incrementally smaller split-widths converging to zero, we obtain an "almost-complete" procedure determining the truth values of *robust* constraint formulae. I.e., it is able to determine the truth of a formula provided it is robust in the sense that the truth value does not change under some small variation of the constants in the formula.

5 Benchmark results

In order to demonstrate the potential of our approach, in particular the benefit of conflict-driven learning adapted to interval constraint solving, we compare the performance of our tool "iSAT" to a stripped version thereof, where learning and backjumping are disabled (but the optimized data structures, in particular watched atoms, remain functional). The benchmark results cover search for conflicts up to a given split width, yet omit the justification of conflict-free valuations thus found by checking for strong satisfaction, as the latter is not implemented yet. The benchmarks were performed on a 2.5 GHz AMD Opteron machine with 4 GByte physical memory, running Linux.

We considered bounded model checking problems, i.e. proving a property of a hybrid discrete-continuous transition system for a fixed unwinding depth k. Without learning, the interval constraint solving system failed on every moderately interesting hybrid system due to complexity problems exhausting memory

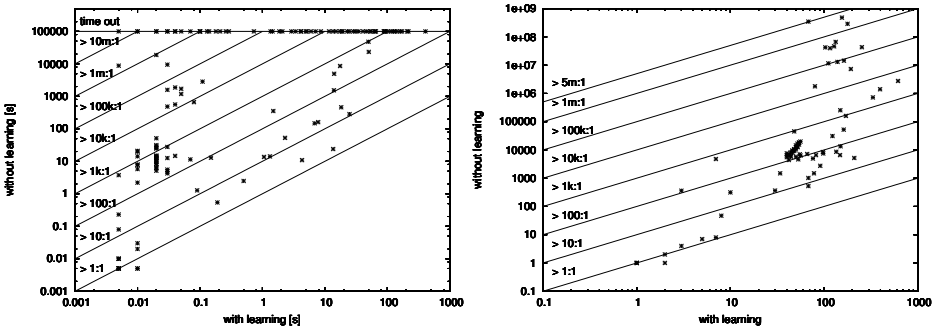

Fig. 3. Performance impact of conflict-driven learning and non-chronological backtracking: runtime in seconds (left) and number of conflicts encountered (right)

and runtime. This could be expected because the *expected* number of boxes to be visited grows exponentially in the number of variables in the constraint formula, which in turn grows linearly in both the number of problem variables in the hybrid system and in the unwinding depth k. When checking a model of an *elastic approach to train distance control* [9], the version without learning exhausts the runtime limit of 3 days already on unwinding depth 1, where formula size is 140 variables and 30 constraints. In contrast, the version with conflict-driven learning solves all instances up to depth 10 in less than 3 minutes, thereby handling instances with more than 1100 variables, a corresponding number of triplets and pairs, and 250 inequality constraints. For simpler hybrid systems, like the model of a *bouncing ball* falling in a gravity field and subject to non-ideal bouncing on the surface, the learning-free solver works due to the deterministic nature of the system. Nevertheless, it fails for unwinding depths > 11, essentially due to enormous numbers of conflicting assignments being constructed (e.g., $> 348 \cdot 10^6$ conflicts for $k = 10$), whereas learning prevents visits to most of these assignments (only 68 conflicts remain for $k = 10$ when conflict-driven learning is pursued). Consequently, the learning-enhanced solver traverses these problems in fractions of a second; it is only from depth 40 that our solver needs more than one minute to solve the bouncing ball problem (2400 variables, 500 constraints). Similar effects were observed on chaotic real-valued maps, like the *gingerbread map*. Without conflict-driven learning, the solver ran into approx. $43 \cdot 10^6$, $291 \cdot 10^6$, and $482 \cdot 10^6$ conflicts for $k = 9$ to 11, whereas only 253, 178, and 155 conflicts were encountered in the conflict-driven approach, respectively. This clearly demonstrates that conflict-driven learning is effective within interval constraint solving: it does dramatically prune the search space, as witnessed by the drastic reduction in conflict situations encountered and by the frequency of backjumps of non-trivial depth, where depths of 47 and 55 decision levels were observed on the gingerbread and bouncing ball model, respectively. Similar effects were observed on two further groups of benchmark examples: an oscillatory *logistic map* and some geometric decision problems dealing with the

intersection of n-dimensional geometric objects. On random formulae, we even obtained backjump distances of more than 70000 levels. The results of all the aforementioned benchmarks (excl. random formulae) are presented in Fig. 3.

6 Discussion

Within this paper, we have shown that a tight integration of DPLL-style SAT solving and interval constraint propagation can canonically lift to interval-based arithmetic constraint solving the crucial algorithmic enhancements of modern propositional SAT solvers, in particular lazy clause evaluation, conflict-driven learning, and non-chronological backtracking. First benchmarks on a prototype implementation demonstrate significant performance gains up to multiple orders of magnitude. Equally important, the performance gains were consistent throughout our set of benchmarks, with only one trivial instance incurring a negligible performance penalty due to the more complex algorithms.

The development of our constraint solver "iSAT" still is in an early phase, with some parts of the algorithm still under implementation. Of the algorithm described above, this does in particular apply to the justification of a solution by checking the interval valuation for strong satisfaction. Plans for future extensions do, furthermore, deal with three major topics: first, we will extend the base engine with specific optimizations for bounded model checking of hybrid systems, akin to the optimizations discussed in [9] for the case of linear hybrid automata. Second, we will use linear programming on the linear subset of the asserted atoms, i.e. on bounds and linear equations, to obtain stronger forward and backward inferences, including additional size reduction of conflicts to be learned. This would lower the overhead when reasoning over timed and (partially) linear hybrid automata, where polyhedral sets provide a more concise description of state sets than the rectangular regions provided by intervals. Finally, we are addressing native support for ordinary differential equations via ICP-based reasoning over safe numerical approximations of the solution in the interval domain, as pursued in [23].

References

1. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics, and R. Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In A. Voronkov, editor, *Proc. of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS, subseries LNAI*, pages 193–208. Springer-Verlag, 2002.
2. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *International Symposium on Logic Programming*, pages 124–138, Ithaca, NY, USA, 1994. MIT Press.
3. F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.
4. J. G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

5. E. Davis. Constraint propagation with interval labels. *Artif. Intell.*, 32(3):281–331, 1987.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
7. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.
8. L. de Moura, S. Owre, H. Ruess, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.
9. M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Formal Methods in System Design*, 2006.
10. H. Ganzinger. Shostak light. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *LNCS*, pages 332–346. Springer-Verlag, 2002.
11. E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Design, Automation, and Test in Europe*, 2002.
12. T. J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001.
13. T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.
14. L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer, Berlin, 2001.
15. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In F. Rossi, editor, *Principles and Practice of Constraint Programming — CP 2003*, volume 2833 of *LNCS*, pages 873–877. Springer-Verlag, 2003.
16. Y. Lebbah, M. Rueher, and C. Michel. A global filtering algorithm for handling systems of quadratic equations and inequations. In P. Van Hentenryck, editor, *Proc. of Principles and Practice of Constraint Programming (CP 2002)*, number 2470 in LNCS. Springer, 2002.
17. M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.
18. O. Lhomme, A. Gotlieb, and M. Rueher. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming*, 37(1–3):165–183, 1998.
19. R. E. Moore. *Interval Analysis*. Prentice Hall, NJ, 1966.
20. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC'01)*, June 2001.
21. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge, 1990.
22. S. Ratschan. Continuous first-order constraint satisfaction. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, number 2385 in LNCS, pages 181–195. Springer, 2002.
23. O. Stauning. *Automatic Validation of Numerical Solutions*. PhD thesis, Kgs. Lyngby, Denmark, 1997.
24. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.