A Survey of Satisfiability Modulo Theory

David Monniaux

To cite this version:

David Monniaux. A Survey of Satisfiability Modulo Theory. Computer Algebra in Scientific Computing, Sep 2016, Bucharest, Romania. hal-01332051

HAL Id: hal-01332051
https://hal.archives-ouvertes.fr/hal-01332051

Submitted on 26 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Survey of Satisfiability Modulo Theory*

David Monniaux

Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France

CNRS, VERIMAG, F-38000 Grenoble, France

June 15, 2016

Abstract

Satisfiability modulo theory (SMT) consists in testing the satisfiability of first-order formulas over linear integer or real arithmetic, or other theories. In this survey, we explain the combination of propositional satisfiability and decision procedures for conjunctions known as DPLL(T), and the alternative "natural domain" approaches. We also cover quantifiers, Craig interpolants, polynomial arithmetic, and how SMT solvers are used in automated software analysis.

1 Introduction

Satisfiability modulo theory (SMT) solving consists in deciding the satisfiability of a first-order formula with unknowns and relations lying in certain theories. For instance, the following formula has no solution $x, y \in \mathbb{R}$:[1]

$$(x \leq 0 \vee x + y \leq 0) \wedge y \geq 1 \wedge x \geq 1\,. \tag{1}$$

The formula may contain negations ($\neg$), conjunctions ($\wedge$), disjunctions ($\vee$) and, possibly, quantifiers ($\exists$, $\forall$).

A *SMT-solver* reports whether a formula is satisfiable, and if so, may provide a *model* of this satisfaction; for instance, if one omits $x \geq 1$ in the preceding formula, then its solutions include $(x = 0, y = 1)$. Other possible features include dynamic addition and retraction of constraints, production of proofs and Craig interpolants (Sec. 4.2), and optimization (Sec. 4.3). SMT-solving has major applications in the formal verification of hardware, software, and control systems.

Quantifier-free SMT subsumes Boolean satisfiability (SAT), the canonical NP-complete problem, and certain classes of formulas accepted by SMT-solvers belong to higher complexity classes or are even undecidable. This has not deterred researchers from looking for algorithms that, in practice, solve many relevant instances at reasonable costs. Care is taken that the worst-case cost does not extend to situations that can be dealt with more cheaply.

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

[1] This survey focuses on linear and polynomial numeric constraints over integers and reals. SMT however encompasses theories as diverse as character strings, inductive data structures, bit-vector arithmetic, and ordinary differential equations.

Most SMT solvers follow the DPLL(T) framework (Sec. 2.2): a CDCL solver for SAT (Sec. 2.1) is used to traverse the Boolean structure, and conjunctions of atoms from the formula are passed to a solver for the theory. This approach limits the interaction between theory values and Boolean reasoning, which led to the introduction of *natural domain* approaches (Sec. 3). Finally, we shall see in Sec. 4 how to go beyond mere quantifier-free satisfiability testing, by handling quantifiers, providing Craig interpolants, or providing optimal solutions. Let us now first see a few generalities, and how SMT-solving is used in practice.

1.1 Generalities

Consider quantifier-free propositional formulas, that is, formulas constructed from *unknowns* (or *variables*) taking the values "true" ($\mathbf{t}$) and "false" ($\mathbf{f}$) and propositional connectives $\vee$ (or), $\wedge$ (and), $\neg$ (not); $\bar{x}$ shall be short-hand for $\neg x$.[2] A formula is: in *negation normal form* (NNF) if the only $\neg$ connectives are at the leaves of its syntax tree (that is, wrap around unknowns but not larger formulas); a *clause* if it is a disjunction of literals (a literal is an unknown or its negation); in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals; in *conjunctive normal form* (CNF) if it is a conjunction of clauses. If A implies B, then A is *stronger* than B and B *weaker* than A. Uppercase letters (F) shall denote formulas, lowercase letters (x) unknowns, and lowercase bold letters ($\vec{x}$) vectors of unknowns.

Satisfiability testing consists in deciding whether there exists a *satisfying assignment* (or *solution*) for these unknowns, that is, an assignment making the formula true. For instance, $a = \mathbf{t}, b = \mathbf{t}, c = \mathbf{f}$ is a satisfying assignment for $(a \vee c) \wedge (b \vee c) \wedge (\bar{a} \vee \bar{c})$. In case the formula is satisfiable, a solver is generally expected to provide such a satisfying assignment; in the case it is unsatisfiable, it may be queried for an *unsatisfiable core*, a subset of the conjunction given as input to the solver that is still unsatisfiable.

Satisfiability modulo theory extends propositional satisfiability by having some atomic propositions be predicates from a theory. For instance, $(x > 0 \vee c) \wedge (y > 0 \vee c) \wedge (x \leq 0 \vee \bar{c})$ is a formula over linear rational arithmetic (LRA) or linear integer arithmetic (LIA), depending on whether x and y are to be interpreted over the rationals or integers.

Different unknowns may range in different sets; for instance $f(x) \neq f(y) \wedge x = z + 1 \wedge z = y - 1$ has unknowns $f : \mathbb{Z} \to \mathbb{Z}$ and $x, y, z \in \mathbb{Z}$. This formula is said to be over the combination of *uninterpreted functions and linear integer arithmetic* (UFLIA). In this formula, f is said to be uninterpreted because we give no definition for it; we shall see in Sec. 2.6 that this formula has no satisfying assignment and how to establish this fact automatically.

1.2 The SMT-LIB Format and Available Theories

SMT solvers can be used i) as a library, from an application programming interface, typically from C/C++, Java, Python, or OCaml ii) as an independent process, from a textual representation, possibly through a bidirectional pipe.

[2] Further propositional connectives, such as exclusive-or, or "let x be e_1 in e_2" constructs may be also considered.

Listing 1: Example of SMT-LIB 2 file. Assertions $x \geq 0$, $y \leq 0$, $f(x) \neq f(y)$ and $x + y \leq 0$ are added, then the problem is checked to be unsatisfiable. The last assertion is retracted and replaced by $x+y \leq 1$, the problem becomes satisfiable and a model is requested (see Listing 2)

```
(set-logic QF_UFLIA)
(set-option :produce-models true)
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun f (Int) Int)
(assert (>= x 0))
(assert (>= y 0))
(assert (distinct (f x) (f y)))
(push 1)
(assert (<= (+ x y) 0))
(check-sat)
(pop 1)
(assert (<= (+ x y) 1))
(check-sat)
(get-model)
```

Listing 2: Z3's answers to the SMT-LIB Listing 1

```
unsat
sat
(model
  (define-fun y () Int 1)
  (define-fun x () Int 0)
  (define-fun f ((x!1 Int)) Int
    (ite (= x!1 0) 2
    (ite (= x!1 1) 3
      2)))
)
```

APIs for SMT-solvers are not standardized, though there have been efforts such as JavaSMT[3] to provide a common layer for several solvers. In contrast, much effort has been put into designing and supporting the common SMT-LIB [4] format, a textual representation (Listing 1); some solvers support other languages than SMT-LIB, sometimes alongside it. Libraries of benchmark problems, sorted according to the theories involved and the presence or absence of quantifiers (Tab. 1), are available in that format. New theories are proposed; for instance, a theory for constraints over IEEE-754 floating-point arithmetic [37] is under evaluation.

Alas, some features, such as quantifier elimination or the extraction of Craig interpolants (Sec. 4.2) do not have standard commands. Furthermore, not all tools implement all operators and commands following the standard.

1.3 Use in Program Analysis Applications

A major use of SMT-solvers is the analysis of software. In most cases (but not always), the solutions of the formula to be tested for satisfiability correspond

[3] `https://github.com/sosy-lab/java-smt` [39]

linear real arithmetic	LRA
linear integer arithmetic	LIA
linear mixed integer and real arithmetic	LIRA
bit-vector arithmetic	BV
nonlinear (polynomial) real arithmetic	NRA
nonlinear (polynomial) integer arithmetic	NIA
nonlinear (polynomial) mixed integer and real arithmetic	NIRA
uninterpreted functions	UF
arrays	A / AX
quantifier-free	QF_

Table 1: Categories of formulas in SMT-LIB; e.g. QF_UFLIA means quantifier-free combination of uninterpreted functions.

to execution traces of the software verifying certain desirable or undesirable properties: for instance traces going into error states.

1.3.1 Symbolic Execution

In *symbolic program execution* [41], a program is executed as though operating on symbolic inputs. Along a straight path in the program, the semantics of the instructions and tests encountered accumulate as a *path condition*, expressing the relationship between the final values and the inputs. In case a branching instruction is encountered, the analyzer tests whether either branch may be taken by checking for a solution to the conjunction of the path condition and the guard associated with the branch: branches for which a solution is known not to exist are not retained for the rest of the analysis. The analysis thus explores a tree of possible executions, which in general does not cover all possible executions of the program: this is acceptable in bug-finding applications.

Pure symbolic execution may prove infeasible due to the large number of paths to explore. This is especially true if the program involves loads and writes to memory, due to the *aliasing* conditions to test ("does this read correspond to this write?"). Because of this, often what is done is a mixture of concrete and symbolic execution, dubbed *concolic*: sometimes a non-symbolic value is picked (e.g. memory allocation addresses) for simpler execution. In *whitebox fuzzing*, concolic execution is applied from symbolic values coming from external inputs (files, network communications) so as to reach security hazards [28].

1.3.2 Inductiveness Check and Bounded Model Checking

In some other cases [27, 32, 33], the formula encodes the full set of executions between two control locations in a program, such that there is no looping construct between these locations: one Boolean variable is added per control location, expressing whether or not the execution goes through that location.

In the Floyd-Hoare approach to proving the correctness of programs (see e.g. [69]), the user is prompted for an inductive invariant for each looping construct: a formula I that holds at loop initiation, and that, if it holds at one loop iteration, holds at the next (*inductiveness*). In other words, there is no execution of the loop guard and loop body that starts in I and ends in $\neg I'$ (I' is I where the variables are renamed in order to express their final, not initial,

values). In modern tools, the loop guard and body are turned into a first-order formula that is conjoined with I and $\neg I'$, then checked for unsatisfiability; or equivalently through a *weakest precondition* computation, as in Frama-C [17].

Example 1. *Consider the array fill program (assume* $n \geq 0$*):*

```
int t[n];
for(int i=0; i<n; i++) t[i] = 42;
```

In order to prove the postcondition $\forall k\ 0 \leq k < n \Rightarrow t[k] = 42$*, one needs the loop invariant*

$$I \stackrel{\triangle}{=} (0 \leq i \leq n) \wedge (\forall k\ 0 \leq 0 \leq k < i \Rightarrow t[k] = 42)\,. \tag{2}$$

The inductiveness condition is

$$(I \wedge i < n) \Rightarrow I[i \mapsto i+1, t \mapsto \mathit{update}(t, i, 42)]\,, \tag{3}$$

where $\mathit{update}(t, i, 42)$ *is the array* t *where* i *has been replaced by* 42*, and* $I[i \mapsto x]$ *is formula* I *where* i *has been replaced by* x*. This condition is checked by showing that the negation of this formula is unsatisfiable — after Skolemization:*

$$(0 \leq i \leq n) \wedge (\forall k\ 0 \leq k < i \Rightarrow t[k] = 42) \wedge i < n \\ \wedge (\neg(0 \leq i+1 \leq n) \vee (0 \leq k_0 \leq i \wedge \mathit{update}(t, i, 42)[k_0] \neq 42))\ . \tag{4}$$

$\mathit{update}(t, i, 42)[k_0]$ *expands into* $\mathit{ite}(k_0 = i, 42, t[k_0])$ *where* $\mathit{ite}(a, b, c)$ *means "if* a *then* b *else* c*". The universal quantifier is instantiated with* $k = k_0$*, a new unknown* $t_k = t[k]$ *is introduced to handle the* uninterpreted function f *(Sec. 2.6) and the resulting problem is solved over linear integer arithmetic (Sec.2.4).*

2 The DPLL(T) Architecture

Most SMT-solvers follow the DPLL(T) architecture: a solver for pure propositional formulas, following the DPLL or CDCL class of algorithms, drives decision procedures for each theory (e.g. linear arithmetic) by adding or retracting constraints and querying for satisfiability. *DPLL(T) and decision procedures for many interesting logics are explained in more detail in e.g. [9, 44].*

2.1 CDCL Satisfiability Testing

We shall only give a cursory view of satisfiability testing and refer the reader to e.g. [6] for more in-depth treatment.

Many algorithms for satisfiability testing for quantifier-free formulas only accept formulas in conjunctive normal form (conjunction of clauses). Naive conversion into conjunctive normal form, by application of distributivity of $\vee$ over $\wedge$, incurs an exponential blowup. It is however possible to construct, from any formula F, a formula F' in CNF but with additional free variables, such that any satisfying assignment to F can be extended to a satisfying assignment on F' and any satisfying assignment on F', restricted to the free variables of F, is a satisfying assignment of F. *Tseitin's encoding* is the simplest way to do so: to any subformula $e_1 \wedge e_2$ of F, associate a new propositional variable $x_{e_1 \wedge e_2}$ and constrain it such that it is equivalent to $e_1 \wedge e_2$ by clauses $\neg x_{e_1 \wedge e_2} \vee e_1$, $\neg x_{e_1 \wedge e_2} \vee e_2$, $\neg e_1 \vee \neg e_2 \vee x_{e_1 \wedge e_2}$ (and similarly for $e_1 \vee e_2$).

Example 2. *Consider*

$$\big((a \wedge \bar{b} \wedge \bar{c}) \vee (b \wedge c \wedge \bar{d})\big) \wedge (\bar{b} \vee \bar{c})\,. \tag{5}$$

Assign propositional variables to sub-formulas:

$$e \equiv a \wedge \bar{b} \wedge \bar{c} \qquad f \equiv b \wedge c \wedge \bar{d} \qquad g \equiv e \vee f \qquad h \equiv \bar{b} \vee \bar{c} \qquad \phi \equiv g \wedge h\,; \tag{6}$$

these equivalences are turned into clauses:

$$\begin{array}{llll} \bar{e} \vee a & \bar{e} \vee \bar{b} & \bar{e} \vee \bar{c} & \bar{a} \vee b \vee c \vee e \\ \bar{f} \vee b & \bar{f} \vee c & \bar{f} \vee d & \bar{b} \vee \bar{c} \vee d \vee f \\ \bar{e} \vee g & \bar{f} \vee g & \bar{g} \vee e \vee f & \\ b \vee h & c \vee h & \bar{h} \vee \bar{b} \vee \bar{c} & \\ \bar{\phi} \vee g & \bar{\phi} \vee h & \bar{g} \vee \bar{h} \vee \phi & \phi\,. \end{array} \tag{7}$$

The model $(a, b, c, d) = (\mathbf{t}, \mathbf{f}, \mathbf{f}, \mathbf{t})$ *of (5) is extended by* $(e, f, g) = (\mathbf{t}, \mathbf{f}, \mathbf{t})$*, producing a model of the system of clauses (7), i.e., the conjunction of these clauses. Conversely, any model of that system, projected over* (a, b, c, d)*, yields a model of (5).*

Let F' be the conjunction of clauses forming the problem. The Davis–Putnam–Logemann–Loveland algorithm (DPLL) decides a propositional formula in CNF (conjunction of clauses) by maintaining a partial assignment of the variables (that is, an assignment to only some of the variables) and *Boolean constraint propagation*: if we have assigned $a = \mathbf{f}, b = \mathbf{t}$ and we have a clause $a \vee \neg b \vee c$, then we can derive $c = \mathbf{t}$. If an assignment satisfies all clauses, then the algorithm terminates with one solution. If it falsifies at least one clause, then there is no solution for our starting partial assignment (thus no solution at all if our starting partial assignment was empty). If propagation is insufficient to conclude, then the algorithm chooses a variable x and a true value b and extends the assignment with $x = b$; if no solution is found for that assignment, then it *backtracks* and replaces it by $x = \bar{b}$. The solver thus constructs a *search tree*.

The practical performance of the solver depends highly on the heuristics for choosing x and b. Much effort has been put into researching these heuristics, such as *Variable State Independent Decaying Sum* (VSIDS) [56]; understanding why they work well is an active research topic. The Boolean constraint propagation phase must be implemented very efficiently, using data structures that minimize the traversal of irrelevant data (clauses that will not result in further propagation); e.g. the *two watched literals per clause* scheme [47, §4.5.1.2].

From a run of the DPLL algorithm concluding to unsatisfiability one can extract a *resolution proof* of unsatisfiability. The proof has the form of a tree whose leaves are some of the original clauses of the problem (constituting an unsatisfiable core) and whose inner nodes correspond to the choices made during the search. Each inner node is the application of the *resolution rule*: knowing $C_1 \vee a$ and $C_2 \vee \bar{a}$, where C_1 and C_2 are clauses and a is a choice variable, one can derive $C_1 \vee C_2$, written:

$$\frac{C_1 \vee a \quad C_2 \vee \bar{a}}{C_1 \vee C_2}\,. \tag{8}$$

Example 3. *Consider the system of clauses 7. Boolean clause propagation from unit clause* ϕ *simplifies* $\bar{\phi} \vee g$ *and* $\bar{\phi} \vee h$ *into* g *and* h *respectively, and removes*

clause $\bar{g} \vee \bar{h} \vee \phi$*. Since* g *and* h *are now* $\mathbf{t}$*, we can remove clauses* $\bar{e} \vee g$ *and* $\bar{f} \vee g$*,* $b \vee h$*, and* $c \vee h$*, and simplify* $\bar{g} \vee e \vee f$ *into* $e \vee f$ *and* $\bar{h} \vee \bar{b} \vee \bar{c}$ *into* $\bar{b} \vee \bar{c}$*:*

$$\begin{array}{ccccc} \bar{e} \vee a & \bar{e} \vee \bar{b} & \bar{e} \vee \bar{c} & \bar{a} \vee b \vee c \vee e & \bar{f} \vee b \\ \bar{f} \vee c & \bar{f} \vee d & \bar{b} \vee \bar{c} \vee d \vee f & e \vee f & \bar{b} \vee \bar{c}\,. \end{array} \tag{9}$$

The system no longer has unit clauses to propagate and thus must pick a literal, for instance b*. By propagation, the system now reaches a contradiction. Since contradiction was reached from assumption* b*, the converse* $\bar{b}$ *must be assumed. In fact, it is possible to derive the learned clause* $\bar{b}$ *by resolution from the set of clauses:*

$$\dfrac{\dfrac{\dfrac{e \vee f \quad \bar{f} \vee c}{e \vee c} \quad \bar{e} \vee \bar{b}}{\bar{b} \vee c} \quad \bar{b} \vee \bar{c}}{\bar{b}}\,. \tag{10}$$

From any "unsatisfiable" run of a DPLL (even in the CDCL variant, see below) solver, a resolution proof can be extracted. This is a fundamental limitation of that approach, since it is known that for certain families of formulas, such as the *pigeonhole principle* [30], any resolution proof has exponential size in the size of the formula — thus any DPLL/CDCL solver will take exponential time.

Performance was considerably increased by extending DPLL with *clause learning*, yielding constraint-driven clause learning (CDCL) algorithms [47]. In CDCL, when a partial assignments leads by propagation to the falsification of a clause, the deductions made during this propagation are analyzed to obtain a subset of the partial assignment sufficient to entail the falsification of this clause. This subset yields a conjunction $\hat{x}_1 \wedge \cdots \wedge \hat{x}_n$ (where $\hat{x}_i$ is either x_i or $\neg x_i$), such that its conjunction with F' is unsatisfiable. In other words, it yields a clause $\neg\hat{x}_1 \vee \cdots \vee \neg\hat{x}_n$ that is a consequence of F' (in fact, that clause can be obtained by resolution from F'). This clause can thus be conjoined to the problem F' without changing its set of solutions; but *learning* that clause may help cut branches in the search tree early.

Again, the learned clause appears as the root of a resolution proof whose leaves are clauses of the original problem. Since the same learned clause may be used several times, the final proof appears as a directed acyclic graph (DAG, i.e., a tree with shared sub-branches). There exist formulas admitting DAG resolution proofs exponentially shorter than the smallest tree resolution proof [67].

A resolution proof, or a more compact format, may thus be produced during an "unsatisfiable" run. A highly optimized SAT or SMT solver is likely to contain bugs, so it may be desirable to have an independent, simpler, possibly formally verified checker reprocess such as proof [3, 8, 40].

2.2 DPLL(T)

The most common way to deal with atomic propositions inside satisfiability testing is the so-called DPLL(T) scheme, combining a CDCL satisfiability solver and a decision procedure for conjunctions of propositions from theory T. A quantifier-free formula F over T, say

$$(x \geq 0 \vee 2x + y \geq 1) \wedge (y \geq 0) \wedge (x + y \leq -1)\,, \tag{11}$$

is converted into a propositional formula F' (here $(a \lor b) \land c \land d$) by replacing each atomic proposition by a propositional variable, using a dictionary (here, $x \geq 0 \mapsto a, 2x + y \geq 1 \mapsto b, y \geq 0 \mapsto c, x + y \leq -1 \mapsto d$) and after conversion to canonical form (so that e.g. $x + y \geq 1$ and $2x + 2y - 2 \geq 0$ are considered the same, and $x + y < 1$ is considered as $\neg(x + y \geq 1)$). F' realizes a *propositional abstraction* of F: any solution of F induces a solution of F', but not all solutions of F' necessarily induce a solution of F.

Consider the solution $a = \mathbf{t}, b = \mathbf{f}, c = \mathbf{t}, d = \mathbf{t}$ of F'; it corresponds to

$$x \geq 0 \land \neg(2x + y \geq 1) \land y \geq 0 \land x + y \leq -1 \,. \tag{12}$$

The inequalities $x \geq 0 \land y \geq 0 \land x + y \leq -1$ have no common solution; in other words, $\neg(a \land c \land d)$ is universally true. The *theory clause* $\neg a \lor \neg c \lor \neg d$ can be conjoined to F'. There remains a solution $a = \mathbf{f}, b = \mathbf{t}, c = \mathbf{t}, d = \mathbf{t}$ of F'; but it entails the contradiction $2x + y \geq 1 \land y \geq 0 \land x + y \leq -1$. The theory clause $\bar{b} \lor \bar{c} \lor \bar{d}$ is then conjoined to F'. Then the propositional problem becomes unsatisfiable, establishing that F has no solution. We have therefore refined the propositional abstraction according to spurious counterexamples.

In current implementations, the propositional solver does not wait until a total satisfying assignment is computed to call the decision procedure for conjunctions of theory formulas. Partial assignments, commonly at each decision point in the DPLL/CDCL algorithm, are tested for satisfiability. In addition, the theory solver may, opportunistically, perform *theory propagation*: if it notices that some asserted constraints imply the truth or falsehood of another known predicate, it can signal it to the SAT solver. The theory solver should be *incremental*, that is, suited for fast addition or retraction of theory constraints, keeping enough internal state to avoid needless recomputation. The SAT solver should be incremental as well, allowing the dynamic addition of clauses.

Multiple theories may be combined, most often by a variant of the Nelson–Oppen approach [44, Ch. 10].

2.3 Linear Real Arithmetic

In the case of linear rational, or equivalently real, arithmetic (LRA), the theory solver is typically implemented using a variant [21, 22] of the simplex algorithm [19, 63]. The atomic (in)equalities from the formula, put in canonical form, are collected; new variables are introduced for the linear combinations of variables that are not of the form $\pm x$ where x is a variable. For instance, (11) is rewritten as $(x \geq 0 \lor \alpha \geq 1) \land (y \geq 0) \land (\beta \leq -1)$, together with the system of linear equalities $\alpha = 2x + y$ and $\beta = x + y$.

The simplex algorithm both maintains a tableau and, for each variable, a current valuation and optional lower and upper bounds. At all times, the simplex tableau contains a system of linear equalities equivalent to this system, such that the variables are partitioned into those (*basic variables*) occurring (each alone) on the left side and those occurring on the right side. The non-basic variables are assigned one of their bounds, or at least a value between these bounds. The simplex algorithm tries to fit each basic variable within its bounds; if one does not fit, it makes it non-basic and assigns to it the bound that was exceeded, and selects a formerly non-basic variable to make it basic, through a *pivoting* operation maintaining the equivalence of the system of equalities.

The algorithm stops when either a candidate solution fitting all bounds is found, either one equation in the simplex tableau can be shown to have no

solution using interval arithmetic from the bounds of the variables (the interval obtained from the right hand side does not intersect that of the basic variable on the left hand side). A pivot selection ordering is used to ensure that the algorithm always terminates. Theory propagation may be performed by noticing that the current tableau implies that some literals are satisfied.

Example 4. *Consider the system*

$$\begin{cases} 2 & \leq 2x + y \\ -6 & \leq 2x - 3y \\ -1000 & \leq 2x + 3y & \leq 18 \\ -2 & \leq -2x + 5y \\ 20 & \leq x + y\,. \end{cases} \tag{13}$$

This system is turned into a system of equations ("tableau") and a system of inequalities on the variables:

$$\begin{cases} a = & 2x & +y & & 2 & \leq a \\ b = & 2x & -3y & & -6 & \leq b \\ c = & 2x & 3y & & -1000 & \leq c & \leq 18 \\ d = & -2x & +5y & & -2 & \leq d \\ e = & x & +y & & 20 & \leq e\,. \end{cases} \tag{14}$$

The variables on the left of the equal signs are deemed "nonbasic" and those on the right are "basic". The simplex algorithm performs pivoting steps on the tableau, akin to those of Gaussian eliminations, until a tableau such as this one is reached:

$$\begin{cases} e = & 7/16c & -1/16d \\ a = & 3/4c & -1/4d \\ b = & 1/4c & -3/4d \\ x = & 5/16c & -3/16d \\ y = & 1/8c & +1/8d\,. \end{cases} \tag{15}$$

Now consider the first equation ($e =$). By interval analysis, knowing $c \leq 18$ and $d \geq -2$, $-7/16c - 1/16d \leq 8$. Yet $e \geq 20$, thus the system has no solution. These coefficients $7/16$ and $1/16$ can be applied to the original inequalities constraining c and d, with coefficient 1 for that defining e, and the resulting inequalities are summed into a trivially false one:

$$\begin{array}{llrcr} 7/16 & (-2x & -3y) & \geq & -7/16 \times 18 \\ 1/16 & (-2x & +5y) & \geq & -1/16 \times 2 \\ 1 & x & +y & \geq & 20 \\ \hline & 0 & 0 & \geq & 28\,. \end{array} \tag{16}$$

By reading nonzero coefficients off the conflicting line of the simplex tableau, one gets a minimal set of contradictory constraints: $d + 1$ constraints, corresponding to the nonbasic variable and the basic variables with nonzero multipliers, where d is the dimension of the space. These multipliers may be presented as an *unsatisfiability witness* to an independent proof checker.

Most SMT solvers implement the simplex algorithm using rational arithmetic. In most cases arising from verification problems, rational arithmetic can be performed using machine integers, without need for going into extended precision arithmetic [57]. A common implementation trick is to use a datatype containing a machine-integer (*numerator*, *denominator*) pair or a pointer to an

extended precision rational.[4] This approach is however very inefficient in the rare cases where the solver goes a lot into extended precision: the size of numerators and denominators grows fast.

This is why it was proposed to perform linear programming in floating-point arithmetic [23, 42, 54, 59].[5] Because the results of floating-point computations cannot be immediately trusted, some checking is needed. One idea is not to recover floating-point numeric information, but the final partition between basic and nonbasic variables [42, 54, 59]; once this partition is known, the tableau is uniquely defined and can be computed by plain linear arithmetic — Gaussian elimination, or better algorithms, including multimodular [66, ch. 7] or p-adic approaches.[6] It is then easy to check the alleged conflicting line, in exact precision.

In some cases, linear arithmetic reasoning may be used to prove the unsatisfiability of polynomial problems. One approach is to expand polynomials and consider all monomials as independent variables (e.g. xy^2 is replaced by a fresh unknown v_{xy^2}). A refinement [46] is to consider lemmas stating that if two polynomials are nonnegative, then so is their product: e.g. $x - 1 \geq 0 \wedge y - 2 \geq 0 \implies v_{xy} - 2x - y + 2 \geq 0$.[7] Because the set of such products has size exponential in the maximal degree, heuristics are used to pick the most promising ones. Experiments have shown this approach to be competitive, even with a rudimentary and sub-optimal connection between linear SMT-solver and nonlinear reasoning.

Some earlier solvers (e.g. CVC3) solver linear real arithmetic by Fourier-Motzkin elimination [26]. This approach is generally not considered efficient, since Fourier-Motzkin elimination tends to generate many redundant constraints, which then may need to be eliminated by linear programming, which defeats the purpose of avoiding using the simplex algorithm.

2.4 Linear Integer Arithmetic

In the case of linear integer arithmetic, the scheme generally used is the same as the one generally used for integer linear programming: the solver first attempts solving the rational relaxation of the problem (nonstrict inequalities are kept, strict inequalities $x < e$ are rewritten as $x \leq e - 1$). If there is no solution over the rationals, there is no integer solution. If a rational solution is found, and has only integral coefficients (say, $(x, y, z) = (0, 1, 2)$), then the problem is decided.

If the proposed solution has non-integral coefficients (say, $(x, y, z) = (\frac{1}{3}, 0, 1)$), then it is excluded by a constraint removing not only that spurious solution but a whole chunk of them. Traditional approaches include i) *branch-and-bound* [63, Sec. 24.1]: add a lemma excluding one segment of non-integral values of the fractional unknowns (here, $x \leq 0 \vee x \geq 1$); branching is however not guaranteed

[4] e.g. ZArith `https://forge.ocamlcore.org/projects/zarith`

[5] The performance with linear programming solvers meant for large industrial instances was however disappointing [23], due to overhead. Closer integration is needed.

[6] As implemented in e.g. Linbox (`http://www.linalg.org/`), IML (`https://cs.uwaterloo.ca/~astorjoh/iml.html`) [11] and SageMath (`http://www.sagemath.org/`).

[7] One can in fact prove a form of completeness of that approach when the problem contains linear constraints defining a bounded polyhedron, and one nonlinear constraint: if such a problem is unsatisfiable, then this can be proved by going to a sufficiently high degree of products. This follows from Krivine–Handelman's theorem [31, 43].

to terminate in general [42]. ii) *Gomory cuts* [63, Ch. 23] iii) *branch-and-cut* [52], a combination of both of the above iv) *cuts from proofs* or *extended branches* [20], which can generate e.g. $x \leq z \vee x \geq z + 1$.

The full integer linear decision procedure can be encapsulated and only export theory lemmas and theory propagation, just as the rational linear procedure, or export the branching lemma to the SMT solver, as a learned clause, so as to allow propositional reasoning over it.

An alternative to linear programming plus branching and/or cuts is Pugh's Omega test [61], which may also be used to simplify constraints. This test is based on Fourier-Motzkin elimination [26], with the twist that, due to divisibility constraints, it may need to enumerate cases up to the least common multiple of the divisors.

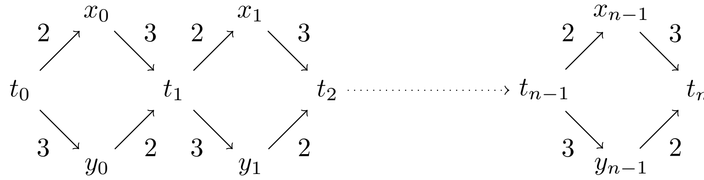
2.5 Exponential Behavior Due to Limited Predicate Vocabulary

Example 5. *Let $n > 0$ be a constant integer. Let $(t_i)_{0 \leq i \leq n}$, $(x_i)_{0 \leq i < n}$ and $(y_i)_{0 \leq i < n}$ be real unknowns (or rational or integer). Let*

$$D_i \triangleq (x_i - t_i \leq 2) \wedge (y_i - t_i \leq 3) \wedge ((t_{i+1} - x_i \leq 3) \vee (t_{i+1} - y_i \leq 2)) \;, \quad (17)$$

$$P_n \triangleq \bigwedge_{i=0}^{n-1} D_i \wedge t_n - t_0 > 5n \,. \quad (18)$$

These formulas are known as "diamond formulas" since they correspond to paths in a difference graph composed of "diamonds":



To a human, it is obvious that $D_i \Rightarrow t_{i+1} \leq t_i + 5$ and thus P_n is unsatisfiable. A DPLL(T) solver, however, proceeds by elimination of contradictory conjunctions of atoms from the original formula. Any contradictory conjunction of atoms from P_n must include a conjunction of the form $\bigwedge_{i=0}^{n-1} F_i \wedge t_n - t_0 > 5n$ where F_i is either $(x_i - t_i \leq 2) \wedge (t_{i+1} - x_i \leq 3)$ or $(y_i - t_i \leq 3) \wedge (t_{i+1} - y_i \leq 2)$. There are an exponential number of such conjunctions, and a DPLL(T) solver has to block them by theory lemmas one by one.

In other words, the proof system used by a DPLL(T) solver is sufficient to prove that a "diamond formula" is unsolvable, but needs exponential proofs for doing so. Any pure DPLL(T) solver, whatever its heuristics and implementation, must thereof run in exponential time on this family of formulas. This motivated the study of algorithms capable of inferring lemmas involving new atoms (Sec. 3.2).

Diamond formulas are simplifications of formulas occurring in e.g. worst-case execution time and scheduling applications. The solution proposed in [34] was to pre-compute upper bounds $t_j - t_i \leq B_{ij}$ on the difference of arrival times between i and j (or, equivalently, the total time spent in the program between

i and j) and conjoin these bounds to the problems. These bounds are logically implied by the original problem, and thus the set of solutions (valid execution traces with timings) does not change; but the resulting formula is considerably more tractable. The lemmas $t_j - t_i \leq B_{ij}$ and $t_k - t_j \leq B_{ik}$ allow the solver to avoid exploring many combinations of paths $i \to j$ and $j \to k$: for instance, if one searches for a path such that $t_k - t_i \geq 100$, it is known that $t_k - t_j \leq 40$, and the solver explores a path $i \to j$ such that $t_j - t_i \leq 42$ on this path, then the solver can immediately cut the search without exploring the paths $j \to k$ in detail.

2.6 Uninterpreted Functions and Arrays

There exists several variants of how to decide uninterpreted functions (UF) in combination with other theories [44, Ch. 4]; we shall expose only one approach here. A quantifier-free formula (e.g. $f(x) \neq f(y) \wedge x = z + 1 \wedge z = y - 1$) is rewritten so that each application of an uninterpreted function is replaced by a fresh variable (e.g. $f_x \neq f_y \wedge x = z+1 \wedge z = y-1$), several identical applications getting the same variable. A solution in x, y, z, f_x, f_y is sought. If $x = y$ but not $f_x \neq f_y$ in that solution, the implication $x = y \Rightarrow f_x = f_y$ is conjoined to the problem. Again, this is a counterexample-guided refinement of the theory.

Example 6. *$f(x) \neq f(y) \wedge x = z + 1 \wedge z = y - 1$, where $x, y, z \in \mathbb{Z}$ and $f : \mathbb{Z} \to \mathbb{Z}$, has no solution because $x = z + 1 \wedge z = y - 1$ implies that $x = y$, and it is then impossible that $f(x) \neq f(y)$. One may establish this by solving $f_x \neq f_y \wedge x = z + 1 \wedge z = y - 1$, getting $(x, y, z, f_x, f_y) = (1, 1, 0, 0, 1)$, noticing the conflict between $x = y$ and $f_x \neq f_y$ and conjoining $x = y \Rightarrow f_x = f_y$.*

Arrays are "functionally updatable" uninterpreted functions [44, Ch. 7]: $\mathit{update}(f, x_0, y_0)$ is the function mapping $x \neq x_0$ to $f[x]$ and x_0 to y_0.

3 Natural-Domain SMT

In DPLL(T) there is a fundamental difference between propositional and other kinds of unknowns: the second are never dealt with directly during the search process. In contrast, in *natural-domain* SMT, one directly constrains and assigns to numeric unknowns during the search. After initial attempts [16, 51], two main directions arose.

3.1 Abstract CDCL (ACDCL)

The DPLL approach is to assign to each unknown (propositional variable) one of **t**, **f**, and "undecided" — that is, a non-empty subset of the set of possible values $\{\mathbf{t}, \mathbf{f}\}$. Initially, all variables are assigned to "undecided". Then, the Boolean constraint propagation phase uses each individual clause as a constraint over its literals: if all literals except for one are assigned to **f**, then the last one gets assigned to **t**. In other words, information known about some variables leads to information on other variables linked by the same constraint. If the information derived is that some variable cannot be assigned some value ("contradiction"), then it means the problem is unsatisfiable. In most cases, however, a contradiction cannot be derived by only the initial pass of propagation. In that case, the system picks an undecided variable and splits the search between the **t** and

f cases. Several splits may be needed, thus the formation of a search tree. If a contradiction is derived in a branch, that branch is closed and the system backtracks to an earlier level.

That approach may be extended to variables lying within an arbitrary domain D, say, the real numbers or the floating-point numbers. The system maintains for each variable an assignment to a subset of D (several types of variables may be used simultaneously, there may therefore be several D), chosen among an *abstract domain*[8] $D^\sharp$ of subsets of D; say, for numeric variables, $D^\sharp$ may be the set of closed intervals of D. Constraints may now constrain variables of different types, and each constraint acts as a propagator of information. For instance, if there is a constraint $x = y + z$, and x is currently assigned the interval $[1, +\infty)$ and y the interval $[4, 10]$, then, applying $z = x - y$, one can derive $x \in [-9, +\infty)$: the current interval for x may thus be refined.

Note that, for soundness, it is not important that the information propagated should be optimally precise, as long as it contains the possible values: in the above example, it would be sound to propagate $x \in [-9.1, +\infty)$ — but unsound to derive $xx \in [-8.99, +\infty)$. In the case of interval propagation for $D = \mathbb{R}$, one sound way to implement it is using floating-point interval arithmetic with directed rounding: the upper bound of an interval is rounded towards $+\infty$, the lower bound towards $-\infty$.

ACDCL also applies clause learning, but in a more general manner than CDCL [10, Sec. 5]. Consider $F \stackrel{\triangle}{=} y = x \wedge z = x \cdot y \wedge z \leq -1$ and a search context with $x \leq -4$. Then, by interval propagation, $y \leq -4$, and $z \geq 16$, which contradicts $z \leq -1$. CDCL-style clause learning would learn that $x \leq -4$ contradicts F, and thus learn the clause $\neg(x \leq -4) \equiv x > -4$. But there is a weaker reason why such choice of x contradicts F: $x < 0$ is sufficient to ensure contradiction; the solver can exclude a larger part of the search space by learning the clause $\neg(x < 0) \equiv x \geq 0$. Generalizing the reasons for a contradiction is a form of *abduction*. One difficulty is that there may be no weakest generalization expressible in the abstract domain: for instance, the choices $x \geq 10$ and $y \geq 10$ contradict the constraint $x + y < 10$, but $x \geq 0 \wedge y \geq 10$, $x \geq 5 \wedge y \geq 5$ and $x \geq 10 \wedge y \geq 0$ are three incomparable generalizations of the contradiction (leading to three clauses $x < 0 \vee y < 10$ etc.), which are optimal in the sense that if one fixes the interval for x (resp. y), the interval for y (resp. x) is the largest that still ensures contradiction.

3.2 Model-Constructing Satisfiability Calculus (MCSAT)

In DPLL(T) i) only propositional atoms (including Boolean unknowns) are assigned during the search ii) the set of atoms considered does not change throughout the search (this may cause exponential behavior, see Sec. 2.5 iii) when the search process, after assigning $b_1, \ldots, b_n$ concludes that it is impossible to assign a Boolean value to an atom b_{n+1}, it derives a learned clause over a subset of $b_1, \ldots, b_n$ that excludes the current assignment but also, hopefully, many more. In contrast, in *model-constructing satisfiability calculus* (MCSAT) [58], both propositional atoms and numeric unknowns get assigned during the search, and new arithmetic predicates are generated through learning.

[8] Following the terminology of *abstract interpretation*; see [10] for more.

3.2.1 Linear Real Arithmetic

Assume variables $x_1, \dots, x_n$ have been assigned values $v(x_1), \dots, v(x_n)$ in the current branch of the search, and that two atoms $x_{n+1} \leq a$ and $x_{n+1} \geq b$, where a and b are linear combinations of variables other than x_{n+1}, have been assigned to $\mathbf{t}$, such that $b > a$ in the assignment v; then it is impossible to pick a value for x_{n+1} in that assignment. In fact, it is impossible to pick a value for it in *any* assignment such that $b > a$.

Assignments that conflict for the same reason are eliminated by a *Fourier-Motzkin elimination* [26] elementary step, valid for all $x_1, \dots, x_{n+1}$:

$$\neg x_{n+1} \leq a \vee \neg x_{n+1} \geq b \vee a \geq b\,. \tag{19}$$

Example 7. *Consider Ex. 5 with $n = 3$. The solver has clauses $x_i - t_i \leq 2$, $y_i - t_i \leq 3$, $t_{i+1} - x_i \leq 3 \vee t_{i+1} - y_i \leq 2$ for $0 \leq i < 3$, and $t_0 = 0$, $t_3 \geq 16$.*

The solver picks $t_0 \mapsto 0$, $t_1 - x_0 \leq 3 \mapsto \mathbf{t}$, $x_0 \mapsto 0$, $t_1 \mapsto 0$, $t_2 - x_1 \leq 3 \mapsto \mathbf{t}$, $x_1 \mapsto 0$, $t_2 \mapsto 0$, $t_3 - x_2 \leq 3 \mapsto \mathbf{t}$, $x_2 \mapsto 0$. But then, there is no way to assign t_3, because of the current assignment $x_2 \mapsto 0$ and the inequalities $t_3 - x_2 \leq 3$ and $t_3 \geq 16$. The solver then learns by Fourier-Motzkin:

$$\neg(t_3 \geq 16) \vee \neg(t_3 - x_2 \leq 3) \vee x_2 \geq 13\,. \tag{20}$$

which may in fact be immediately simplified by resolution with the original clause $t_3 \geq 16$ to yield $\neg(t_3 - x_2 \leq 3) \vee x_2 \geq 13$. The assignment to x_2 is retracted.

But then, there is no way to assign x_2, because of the current assignment $t_2 \mapsto 0$ and the inequality $x_2 - t_2 \leq 2$. The solver then learns by Fourier-Motzkin:

$$\neg(x_2 \geq 13) \vee \neg(x_2 - t_2 \leq 2) \vee t_2 \geq 11\,. \tag{21}$$

By resolution, $\neg(t_3 - x_2 \leq 3) \vee t_2 \geq 11$. The truth assignment to $t_3 - x_2 \leq 3$ is retracted.

At this point, the solver has $t_0 \mapsto 0$, $t_1 - x_0 \leq 3 \mapsto \mathbf{t}$, $x_0 \mapsto 0$, $t_1 \mapsto 0$, $t_2 - x_1 \leq 3 \mapsto \mathbf{t}$, $x_1 \mapsto 0$, $t_2 \mapsto 0$, $t_3 - x_2 \leq 3 \mapsto \mathbf{f}$. By similar reasoning in that branch, the solver derives $t_3 - x_2 \leq 3 \vee t_2 \geq 11$. By resolution between the outcomes of both branches, one gets $t_2 \geq 11$.

By similar reasoning, one gets $t_1 \geq 6$ and then $t_0 \geq 1$, but then there is no satisfying assignment to t_0. The problem has no solution.

In contrast to the exponential behavior of DPLL(T) on Ex. 5, MCSAT has linear behavior: each branch of each individual disjunction is explored only once, and the whole disjunction is then summarized by an extra atom.

The dynamic generation of new atoms by MCSAT, as opposed to DPLL(T), creates two issues. i) If infinitely many new atoms may be generated, termination is no longer ensured. One can ensure termination by restricting the generation of new atoms to a *finite basis* (this basis of course depends on the original formula); this is the case for instance if the numeric variables $x_1, \dots, x_n$ are always assigned in the same order, thus the generated new atoms are results of Fourier-Motzkin elimination of x_n, then of x_{n-1} etc. down to x_2.[9] In practice, the interest of being able to choose variable ordering trumps the desire to prove termination. ii) Since many new atoms and clauses are generated, some garbage collection must be applied, as with learned clauses in a CDCL solver.

Implementation-wise, note that, like a clause in CDCL, a linear inequality is processed only when all variables except for one are assigned. Similar to *two watched literals per clause*, one can apply *two watched variables per inequality*.

[9] Successive applications of Fourier-Motzkin may lead to very large sets of predicates, thus this argument seems of mostly theoretical interest.

3.2.2 Nonlinear Arithmetic (NRA)

The MCSAT approach can also be applied to polynomial real arithmetic. Again, the problem is: assuming a set of polynomial constraints over $x_1, \dots, x_n, x_{n+1}$ have no solution over x_{n+1} for a given valuation $v(x_1), \dots, v(x_n)$, how can we explain this impossibility by a system of constraints over $x_1, \dots, x_n$ that excludes $v(x_1), \dots, v(x_n)$ and hopefully many more?

Jovanović and de Moura [38] proposed applying a modified version of Collin's [14] projection operator in order to perform a partial *cylindrical algebraic decomposition*. In that approach, known as NLSAT, one additional difficulty is that assignments to variables may refer to algebraic reals, and thus the system needs to compute to compute over algebraic reals, including as coefficients to polynomials. It is yet unknown whether this approach could benefit from using other projection operators such as Hong's [36] or McCallum's [48].

4 Beyond Quantifier-Free Decidability

4.1 Quantifiers

4.1.1 Quantifier Elimination by Virtual Substitution

In the case of some theories, such as linear real arithmetic, a finite sequence of instantiations can be produced such that $F \triangleq \forall x\ P(x)$ is equivalent to $\bigwedge_{i=1}^{n} P(v_i)$; note that the v_i are not constants, but functions of the free variables of F, obtained by analyzing the atoms of P. Because this approach amounts to substituting expressions into the quantified variable, it is called *substitution*, or *virtual substitution* if appropriate data structures and algorithms avoid explicit substitution. Examples of substitution-based methods include Cooper's [15] for linear integer arithmetic, Ferrante & Rackoff's [24] and Loos & Weisfpenning's [45] methods for linear real arithmetic.

Example 8. *Consider $\forall y\ (y \geq x \Rightarrow y \geq 1)$. Loos & Weisfpenning's method collects the expression to which y is compared (here, x and 1) and then substitutes them into y. For each expression e, one must also substitute $e + \epsilon$ where ϵ is infinitesimal,[10] and also substitute $-\infty$ (equivalently, one can substitute $e - \epsilon$ for each expression, and also $+\infty$). The result is therefore*

$$\bigwedge_{e \in \{x, x+\epsilon, 1, 1+\epsilon, -\infty\}} e \geq x \Rightarrow e \geq 1 \tag{22}$$

or, after expansion and simplification, $x \geq 1$.

We thus have *eliminated* the quantifier; by recursion over the structure of a formula and starting at the leaves, we can transform any formula of linear real arithmetic into an equivalent quantifier-free formula.[11]

In these eager approaches, the size of the substitution set may grow quickly (especially for linear integer arithmetic, which may involve enumerating all cases up to the least common multiple of the divisibility constants). For this reason, lazy approaches were proposed where the substitutions are generated from

[10] $x \geq K + \epsilon$ with K real means $x > K$.

[11] In the case of linear integer arithmetic, we need to enrich the language of the output formula with constraints of divisibility by constants: e.g. $\exists x\ y = 2x$ is equivalent to quantifier-free $2 \mid x$.

counterexamples, in much the same way that learned lemmas are generated in DPLL(T) [7, 60]. For a formula $A(\vec{x}) \wedge \forall y\ B(\vec{x}, y)$, the system first solves $A(\vec{x})$ for a solution $\vec{x}_0$, then checks whether there exists y such that $\neg B(\vec{x}_0, y)$; if so, such an y_0 is generalized into one of the possible substitutions $S_1(\vec{x})$ and the system restarts by solving $A(\vec{x}) \wedge B(\vec{x}, S_1(\vec{x})$. The process iterates until a solution is found or the substitutions accumulated block all solutions for $\vec{x}$; termination is ensured because the set of possible symbolic substitutions is finite. Note that a full quantifier elimination is not necessary to produce a solution.

4.1.2 Quantifier Elimination by Projection

In case a quantifier elimination, or *projection*, algorithm, is available for conjunctions of constraints, as happens with linear real arithmetic,[12] one can, given a formula $\exists \vec{y}\ F(\vec{x}, \vec{y})$, find a conjunction $C_1 \Rightarrow F$, project C_1 over $\vec{x}$ as $\pi(C_1)$, conjoin $\neg\pi(C_1)$ to F and repeat the process (generating C_2 etc.) until the F becomes unsatisfiable [53]. $\bigvee_i C_i$ is then equivalent to $\exists \vec{y}\ F$. Again, this process may be made lazier, for nested quantification in particular [55, 60].

4.1.3 Instantiation Heuristics

The addition of quantifiers to theories (such as linear integer arithmetic plus uninterpreted functions) may make them undecidable. This does not however deter designers of SMT solvers from attempting to have them decide as many formulas as possible. A basic approach is quantifier instantiation by E-matching. If a formula in negation normal form contains a subformula $\forall x\ P(x)$, then this formula is replaced by a finite instantiation $\bigwedge_{i=1}^{n} P(v_i)$. The v_i are extracted from the rest of the formula, possibly guided by counterexamples. This approach is not guaranteed to converge: an infinite sequence of instantiations may be produced for a given quantifier. In the case of *local theories*, one can however prove termination.

4.2 Craig Interpolation

The following conjunction is satisfiable if and only if it is possible to go from a model $\vec{x}_0$ of A to a model $\vec{x}_n$ of B by a sequence of transitions $(\tau_i)_{1 \leq i \leq n}$:

$$A(\vec{x}_0) \wedge \tau_1(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \tau_n(\vec{x}_{n-1}, \vec{x}_n) \wedge B(\vec{x}_n)\,. \quad (23)$$

In program analysis, A typically expresses a precondition, $\neg B$ a postcondition, $\vec{x}_i$ the variables of the program after i instruction steps, and τ_i the semantics of the i-th instruction in a sequence, and the formula is unsatisfiable if and only if B is always true after executing that sequence of instruction starting from A.

A hand proof of unsatisfiability would often consist in exhibiting predicates $I_1(\vec{x}_1), \ldots, I_{n-1}(\vec{x}_{n-1})$, such that, posing $I_0 = A$ and $I_n = B$, for all $0 \leq i < n$,

$$\forall \vec{x}_i, \vec{x}_{i+1}\ I_i(\vec{x}_i) \wedge \tau_{i+1}(\vec{x}_i, \vec{x}_{i+1}) \Rightarrow I_{i+1}(\vec{x}_{i+1})\,, \quad (24)$$

along with proofs of these *local inductiveness* implications.[13]

[12] This amounts to projection of convex polyhedra, for which there exist algorithms based on conversion to generators (vertices), Fourier-Motzkin elimination and pruning, or parametric linear programming, among others [25].

[13] In program analysis, this corresponds to stating "after the first instruction, the program variables satisfy I_1, but then if one executes the second instruction from I_1, the program variables then satisfy I_2..."", and $\{I_i\}\ \tau_i\ \{I_{i+1}\}$ constitute Hoare triples.

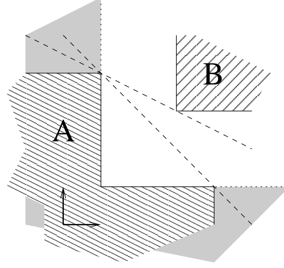


Figure 1: Binary interpolation in linear arithmetic. The hashed areas represent A and B (Eq. 28) respectively. A possible interpolant I between A and B ($A \Rightarrow I$, $I \Rightarrow \neg B$) is the grey area $x \leq 1 \vee y \leq 1$. The dashed lines define two other possible interpolants, $x + y \leq 5$ and $x + 2y \leq 9$.

A SMT-solver, in contrast, produces a monolithic proof of unsatisfiability of (23): it mixes variables from different $\vec{x}_i$, that is, in program analysis, from different times of the execution of the program. It is however possible to obtain instead a sequence I_i satisfying (24) by post-processing that proof [12, 13, 49].

In any theory admitting quantifier elimination, such a sequence must exist:

$$I_{i+1} \equiv \exists \vec{x}_i \; I_i(\vec{x}_i) \wedge \tau_{i+1}(\vec{x}_i, \vec{x}_{i+1}) \tag{25}$$

defines the strongest sequence of valid interpolants; the weakest is:

$$I_i \equiv \forall \vec{x}_{i+1} \; \tau_{i+1}(\vec{x}_i, \vec{x}_{i+1}) \Rightarrow I_{i+1}(\vec{x}_{i+1})\,. \tag{26}$$

The strongest sequence corresponds to computing exactly the sequence of sets of states reachable by τ_1, then $\tau_2 \circ \tau_1$ etc. from A.

Binary interpolation consists in: given A and B, produce I such that

$$\forall \vec{x}_0, \vec{x}_1, \vec{x}_2 \; A(\vec{x}_0, \vec{x}_1) \Rightarrow I(\vec{x}_1) \Rightarrow B(\vec{x}_1, \vec{x}_2)\,, \tag{27}$$

in which case, if the theory admits quantifier elimination, $\exists \vec{x}_0 \; A(\vec{x}_0, \vec{x}_1)$ and $\forall \vec{x}_2 \; B(\vec{x}_1, \vec{x}_2)$ are respectively the strongest and weakest interpolant, and any I in between ($\exists \vec{x}_0 \; A(\vec{x}_0, \vec{x}_1) \Rightarrow I \Rightarrow \forall \vec{x}_2 \; B(\vec{x}_1, \vec{x}_2)$) is also an interpolant.

One of the main uses of Craig interpolation in program analysis is to synthesize inductive invariants, for instance by counterexample-guided abstraction refinement in predicate abstraction (CEGAR) [50] or property-guided reachability (PDR). Interpolants obtained by quantifier elimination are too specific (*overfitting*): for instance, strongest interpolants exactly fit the set of states reachable in $1, 2, \ldots$ steps. It has been argued that interpolants likely to be useful as inductive invariants should be "simple" — short formula, with few "magical constants". A variety of approaches have been proposed for getting such interpolants [2, 65, 68] or to simplify existing interpolants [35].

Example 9. *Consider the interpolation problem $A \Rightarrow I$, $I \Rightarrow \neg B$ (Fig. 1):*

$$\begin{array}{llll} A_1 & \triangleq x \leq 1 \wedge y \leq 4 & A_2 & \triangleq x \leq 4 \wedge y \leq 1 \\ A & \triangleq A_1 \vee A_2 & B & \triangleq x \geq 3 \wedge y \geq 3\,. \end{array} \tag{28}$$

SMTInterpol[14] *and* MathSAT[15] *produce $I \triangleq x \leq 1 \vee y \leq 1$. This is due to the way these tools produce interpolants from DPLL(T) proofs of unsatisfiability. On this example, a DPLL(T) solver will essentially analyze both branches*

[14] SMTInterpol 2.1-31-gafd0372-comp
[15] MathSAT 5.3.10

of $A_1 \vee A_2$. The first branch yields $A_1 \Rightarrow \neg B$. Finding I_1 such that $A_1 \Rightarrow I_1$ and $I_1 \Rightarrow \neg B$ amounts to finding a separating hyperplane between these two convex polyhedra; $I_1 \stackrel{\triangle}{=} x \leq 1$ works. Similarly, I_2 such that $A_1 \Rightarrow I_2$ and $I_2 \Rightarrow \neg B$ can be $I_2 \stackrel{\triangle}{=} y \leq$. $I_1 \vee I_2$ is then produced as interpolant.

Yet, a search for a single separating hyperplane may produce $x + 2y \leq 9$, or $x + y \leq 5$. The second hyperplane may seem preferable according to a criterion limiting the magnitude of integer constants.

It is easy to see that if one can find interpolants for arbitrary conjunctions A, B such that $A \Rightarrow \neg B$, one can find them between arbitrary quantifier-free formulas, by putting them into DNF. Because such a procedure would be needlessly costly due to disjunctive normal forms, the usual approach is to post-process a DPLL(T) proof that $A(\vec{x}, \vec{y}) \wedge B(\vec{y}, \vec{z})$ is unsatisfiable [12, 13]. First, interpolants are derived for all theory lemmas: each lemma expresses that a conjunction of atoms from the original formula is unsatisfiable, these atoms can thus be divided into a conjunction α of atoms from A and a conjunction β of atoms from B, and an interpolant I is derived for $\alpha \Rightarrow \neg\beta$. Then, these interpolants are combined following the resolution proof of the solver. This is how the interpolants from Ex. 9 were produced by the solvers.

The problem is therefore: given $A(\vec{x}, \vec{y}) \wedge B(\vec{y}, \vec{z})$ unsatisfiable, where A and B are conjunctions, how do we find $I(\vec{y})$ such that $A \Rightarrow I$ and $I \Rightarrow \neg B$? If the theory is linear rational arithmetic, this amounts to finding a separating hyperplane between the polyhedra A and B. Let us note

$$A \stackrel{\triangle}{=} \bigwedge_i \vec{a''}_i \cdot \vec{x} + \vec{a}_i \cdot \vec{y} \geq a'_i\,, \qquad B \stackrel{\triangle}{=} \bigwedge_j \vec{b''}_j \cdot \vec{z} + \vec{b}_j \cdot \vec{y} \geq b'_j\,. \tag{29}$$

Each a'_i (resp. b'_j) is a pair $({a'}_i^{\mathbb{R}}, {a'}_i^{\epsilon})$ lexicographically ordered, where ${a'}_i^{\mathbb{R}}$ is the real part and ${c'}_i^{\epsilon}$ is infinitesimal; all other numbers are assumed to be real. $y \geq (x^{\mathbb{R}}, x^{\epsilon})$ with $x^{\epsilon} > 0$ and $y \in \mathbb{R}$ expresses that $y > x^{\mathbb{R}}$.

Since $A \wedge B$ is unsatisfiable, by Farkas' lemma, there exists an unsatisfiability witness (λ_i), (μ_j), such that

$$\begin{array}{rlrl} \sum_i \lambda_i \vec{a}''_i & = 0 & \sum_j \mu_j \vec{b}''_j & = 0 \\ \sum_i \lambda_i \vec{a}_i + \sum_j \mu_j \vec{b}_j & = 0 & \sum_i a'_i + \sum_j b'_j & > 0 \end{array} \tag{30}$$

Such coefficients can in fact be read off the simplex tableau from the most common way of implementing a DPLL(T) solver for linear real arithmetic, as described in Sec. 2.3. Then the following is a valid interpolant (recall that the right-hand side can contain infinitesimals, leading to $>$):

$$I \stackrel{\triangle}{=} \sum_i (\lambda_i \vec{a}_i) \cdot \vec{y} \geq \sum_i \lambda_i a'_i \tag{31}$$

For polynomial arithmetic, one approach replaces nonnegative reals by sums-of-squares of polynomials, and Farkas' lemma by Positivstellensatz [18].

Another difficulty is posed for certain theories, for which the solving process involves generating lemmas introducing atoms not present in the original. Consider the approaches for linear integer arithmetic described in Section 2.4: except for branch-and-bound, all can generate new constraints involving any of the unknowns, without respecting the original partition of variables. This poses a problem for interpolation: if interpolating for $A(x, y) \wedge B(y, z)$ over linear real arithmetic, we can rely on all atomic propositions being linear inequalities either

over x, y or y, z, but here, we have new atomic propositions that can involve both x, z. Special theory-dependent methods are needed to get rid of these new propositions when processing the DPLL(T) proof into an interpolant [12, 13].

4.3 Optimization

Instead of finding one solution, one may wish to find a solution that maximizes (or nearly so) some function f.

A simple approach is *binary search*: provided one can get a lower bound l and an upper bound h on the maximum $f(\vec{x}^*)$, one queries the solver for a solution $\vec{x}$ such that $f(\vec{x}) \geq m$, where $m = \frac{l+h}{2}$; if such a solution is found, refine the lower bound $l := f(\vec{x})$ and restart, otherwise $h := m$ and restart. Proceed until $l = h$. This converges in finite time if f has integer value. This approach has been successfully applied to e.g. worst-case execution time problems [34].

In the case of LRA (resp. LIA), optimization generalizes *linear programming* (resp. *linear integer programming*) to formulas with disjunctions. In fact, linear programming can be applied locally to a polyhedron of solutions: when a DPLL(T) solver finds a solution $\vec{x}$ of a formula F, it also finds a conjunction C of atoms such that $C \Rightarrow F$; C defines a polyhedron and one can optimize within it, until a local optimum $\vec{x}^l$. Then one adds the constraint $f(\vec{x}) > f(\vec{x}^l)$ and restart; the last $\vec{x}^l$ found is the optimum (one can also detected unboundedness). This approach can never enumerate the same C (or subsets thereof) twice and thus must terminate. It may, however, scan an exponential number of useless C's; it may be combined with binary search for best effect [64].

5 Conclusion

Considerable progress has been made within the last 15 years on increasingly practical decision procedures for increasingly large classes of formulas, even though worst-case complexity is prohibitive, and sometimes even though the class is undecidable.[16] Major ingredients to that success were i) lazy generation of lemmas, partial projections or instantiations, guided by counterexamples (as opposed to eager exhaustive generation, often explosive) ii) generalization of counterexamples so as to *learn* sufficiently general blocking lemmas iii) tight integration of propositional and theory-specific reasoning.

Nonlinear arithmetic reasoning (polynomials, or even transcendental functions) is still a very open question. Current approaches in SMT [38] are based on partial cylindrical algebraic decomposition [14]; possibly methods based on critical points [5, 29, 62] could be investigated as well.

There are several challenges to using computer algebra procedures inside a SMT solver. i) These procedures may not admit addition or retraction of constraints without recomputation. ii) They may compute eagerly large sets of formulas (as in conventional cylindrical algebraic decomposition). iii) They may be very complex and thus likely to contain bugs.[17] Being able to produce

[16]Worst-case complexity, or completeness in complexity classes, is therefore not always a good indicator of practical performance. Average complexity is difficult to define (one needs to suppose a probability distribution on formulas) and may ill-describe practical use cases: it is well-known that random SAT instances behave unlike industrial examples [1], and same with random linear constraints [54]. For want of better indication, performance is measured on libraries of benchmarks.

[17]The author had several computer algebra packages crash or produce wrong results. Perhaps running large libraries of benchmarks would help in finding such bugs.

independently-checkable proof witnesses would help in this respect.

Acknowledgements Thanks to the anonymous referees for their careful proof-reading.

References

[1] Dimitris Achlioptas. "Random satisfiability". In: *Handbook of Satisfiability.* Ed. by Armin Biere et al. Vol. 185. IOS Press, 2009. Chap. 8, pp. 245–270. ISBN: 978-1-58603-929-5.

[2] Aws Albarghouthi and Kenneth L. McMillan. "Beautiful interpolants". In: *Computer-Aided Verification (CAV).* Vol. 8044. LNCS. Springer, 2013, pp. 313–329. DOI: 10.1007/978-3-642-39799-8_22.

[3] Michaël Armand et al. "A modular integration of SAT/SMT solvers to Coq through proof witnesses". In: *Certified Programs and Proofs (CPP).* Vol. 7086. LNCS. Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB).* `www.SMT-LIB.org`. 2016.

[5] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in real algebraic geometry.* Springer, 2006. ISBN: 978-3-540-33098-1.

[6] Armin Biere et al., eds. *Handbook of satisfiability.* Vol. 185. IOS Press, 2009, p. 980. ISBN: 978-1-58603-929-5.

[7] Nikolaj Bjørner. "Linear quantifier elimination as an abstract decision procedure". In: *Automated reasoning (IJCAR).* Vol. 6173. LNCS. Springer, 2010, pp. 316–330. DOI: 10.1007/978-3-642-14203-1_27.

[8] Sascha Böhme and Tjark Weber. "Fast LCF-style proof reconstruction for Z3". In: *Interactive Theorem Proving (ITP).* Vol. 6172. LNCS. Springer, 2010, pp. 179–194. DOI: 10.1007/978-3-642-14052-5_14.

[9] Aaron R. Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification.* Springer, Oct. 2007. ISBN: 3-540-74112-7.

[10] Martin Brain et al. "Deciding floating-point logic with abstract conflict driven clause learning". In: *Formal Methods in System Design* 45.2 (2014), pp. 213–245. DOI: 10.1007/s10703-013-0203-7.

[11] Zhuliang Chen and Arne Storjohann. "A BLAS based C library for exact linear algebra on integer matrices". In: *ISSAC.* Beijing, China: ACM, 2005, pp. 92–99. ISBN: 1-59593-095-7. DOI: 10.1145/1073884.1073899.

[12] Jürgen Christ. "Interpolation modulo theories". PhD thesis. Albert-Ludwigs-Universität Freiburg, Oct. 2015. DOI: 10.6094/UNIFR/10342.

[13] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "Proof tree preserving interpolation". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Vol. 7795. LNCS. Springer, 2013, pp. 124–138. DOI: 10.1007/978-3-642-36742-7_9.

[14] George E. Collins. "Quantifier elimination for real closed fields by cylindrical algebraic decomposition". In: *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern.* Vol. 33. LNCS. Springer, 1975, pp. 134–183. ISBN: 978-3-540-07407-6. DOI: 10.1007/3-540-07407-4_17.

[15] D. C. Cooper. “Theorem proving in arithmetic without multiplication”. In: *Machine Intelligence 7*. Edinburgh University Press, 1972, pp. 91–100. ISBN: 0-85224-234-4.

[16] Scott Cotton. “Natural domain SMT: a preliminary assessment”. In: *Formal Modeling and Analysis of Timed Systems (FORMATS)*. Vol. 6246. LNCS. Springer, 2010, pp. 77–91. DOI: 10.1007/978-3-642-15297-9_8.

[17] Pascal Cuoq et al. “Frama-C - a software analysis perspective”. In: *SEFM'12*. Vol. 7504. LNCS. 2012. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7_16.

[18] Liyun Dai, Bican Xia, and Naijun Zhan. “Generating non-linear interpolants by semidefinite programming”. In: *Computer Aided Verification (CAV)*. Vol. 8044. Springer, 2013, pp. 364–380. DOI: 10.1007/978-3-642-39799-8_25.

[19] George B. Dantzig and Mukund N. Thapa. *Linear programming 1: introduction*. Springer, 1997.

[20] Işıl Dillig, Thomas Dillig, and Alex Aiken. “Cuts from proofs: a complete and practical technique for solving linear inequalities over integers”. In: *Formal Methods in System Design* 39.3 (2011), pp. 246–260. DOI: 10.1007/s10703-011-0127-z.

[21] Bruno Dutertre and Leonardo Mendonça de Moura. “A fast linear-arithmetic solver for DPLL(T)”. In: *Computer Aided Verification (CAV)*. Vol. 4144. LNCS. Springer, 2006, pp. 81–94. DOI: 10.1007/11817963_11.

[22] Bruno Dutertre and Leonardo Mendonça de Moura. *Integrating Simplex with DPLL(T)*. SRI-CSL-06-01. SRI International, computer science laboratory, May 2006. URL: http://www.csl.sri.com/users/bruno/publis/sri-csl-06-01.pdf.

[23] Germain Faure et al. “SAT modulo the theory of linear arithmetic: exact, inexact and commercial solvers”. In: *Theory and Applications of Satisfiability Testing (SAT)*. Vol. 4996. LNCS. Springer, 2008, pp. 77–90. DOI: 10.1007/978-3-540-79719-7_8.

[24] Jeanne Ferrante and Charles Rackoff. “A decision procedure for the first order theory of real addition with order”. In: *SIAM J. on Computing* 4.1 (Mar. 1975), pp. 69–76. ISSN: 0097-5397. DOI: 10.1137/0204006.

[25] Alexis Fouilhé. “Revisiting the abstract domain of polyhedra: constraints-only representation and formula proof”. PhD thesis. Université de Grenoble, 2015. HAL: tel-01286086.

[26] Joseph Fourier. “Histoire de l'académie, partie mathématique (1824)”. In: *Mémoires de l'Académie des sciences de l'Institut de France*. Vol. 7. Gauthier-Villars, 1827, xlvij–lv. Gallica: ark:/12148/bpt6k32227/f53.

[27] Thomas Gawlitza and David Monniaux. “Invariant generation through strategy iteration in succinctly represented control flow graphs”. In: *Logical Methods in Computer Science* (Sept. 2012). ISSN: 1860-5974. DOI: 10.2168/LMCS-8(3:29)2012. arXiv: 1209.0643.

[28] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Queue* 10.1 (Jan. 2012), 20:20–20:27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081.

[29] D. Yu. Grigor'ev and N. N. Vorobjov Jr. “Solving systems of polynomial inequalities in subexponential time”. In: *J. Symb. Comput.* 5.1–2 (Feb. 1988), pp. 37–64. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(88)80005-1.

[30] Armin Haken. "The intractability of resolution". In: *Theoretical Computer Science* 39 (1985), pp. 297–308. DOI: 10.1016/0304-3975(85)90144-6.

[31] David Handelman. "Representing polynomials by positive linear functions on compact convex polyhedra". In: *Pacific Journal of Mathematics* 132.1 (1988), pp. 35–62. DOI: 10.2140/pjm.1988.132.35.

[32] Julien Henry, David Monniaux, and Matthieu Moy. "PAGAI: a path sensitive static analyzer". In: *Tools for Automatic Program Analysis (TAPAS)*. 2012. DOI: 10.1016/j.entcs.2012.11.003. arXiv: 1207.3937.

[33] Julien Henry, David Monniaux, and Matthieu Moy. "Succinct representations for abstract interpretation. Combined analysis algorithms and experimental evaluation". In: *Static analysis (SAS)*. Vol. 7460. LNCS. Springer, 2012, pp. 283–299. DOI: 10.1007/978-3-642-33125-1_20. arXiv: 1206.4234.

[34] Julien Henry et al. "How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics". In: *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*. ACM, 2014, pp. 43–52. ISBN: 978-1-4503-2877-7. DOI: 10.1145/2597809.2597817. HAL: hal-00998138.

[35] Krystof Hoder, Laura Kovács, and Andrei Voronkov. "Playing in the grey area of proofs". In: *ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 2012, pp. 259–272. DOI: 10.1145/2103656.2103689.

[36] Hoon Hong. "An improvement of the projection operator in cylindrical algebraic decomposition". In: *ISSAC*. Tokyo, Japan: ACM, 1990, pp. 261–264. ISBN: 0-201-54892-5. DOI: 10.1145/96877.96943.

[37] *IEEE standard for Binary floating-point arithmetic for microprocessor systems*. ANSI/IEEE Std 754-1985. IEEE. 1985.

[38] Dejan Jovanović and Leonardo de Moura. "Solving non-linear arithmetic". In: *Automated reasoning (IJCAR)*. Vol. 7364. LNCS. Springer, 2012, pp. 339–354. ISBN: 978-3-642-31364-6. DOI: 10.1007/978-3-642-31365-3_27.

[39] Egor Karpenkov, Dirk Beyer, and Karlheinz Friedberger. "JavaSMT: a unified interface for SMT solvers in Java". In: *VSTTE*. To appear. 2016.

[40] Chantal Keller. "Extended resolution as certificates for propositional logic". In: *Proof Exchange for Theorem Proving (PxTP)*. Vol. 14. EPiC Series. EasyChair, 2013, pp. 96–109. URL: http://www.easychair.org/publications/?page=117514525.

[41] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[42] Tim King, Clark W. Barrett, and Cesare Tinelli. "Leveraging linear and mixed integer programming for SMT". In: *Formal Methods in Computer-Aided Design, (FMCAD)*. IEEE, 2014, pp. 139–146. DOI: 10.1109/FMCAD.2014.6987606.

[43] Jean-Louis Krivine. "Anneaux préordonnés". In: *Journal d'analyse mathématique* 12 (1964), pp. 307–326. HAL: hal-00165658.

[44] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008. ISBN: 978-3-540-74104-6.

[45] Rüdiger Loos and Volker Weispfenning. "Applying linear quantifier elimination". In: *The Computer Journal* 36.5 (1993). Special issue on computational quantifier elimination, pp. 450–462. DOI: 10.1093/comjnl/36.5.450.

[46] Alexandre Maréchal et al. "Polyhedral approximation of multivariate polynomials using Handelman's theorem". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI).* Vol. 9583. LNCS. Springer, 2016, pp. 166–184. DOI: 10.1007/978-3-662-49122-5_8. HAL: hal-01223362.

[47] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. "Conflict-driven clause learning SAT solvers". In: *Handbook of Satisfiability.* Ed. by Armin Biere et al. Vol. 185. IOS Press, 2009. Chap. 4, pp. 131–153. ISBN: 978-1-58603-929-5.

[48] Scott McCallum. "An improved projection operation for cylindrical algebraic decomposition". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer, 1998, pp. 242–268. ISBN: 978-3-211-82794-9. DOI: 10.1007/978-3-7091-9459-1_12.

[49] Kenneth L. McMillan. "An interpolating theorem prover". In: *Theoretical Computer Science* 345.1 (2005), pp. 101–121. DOI: 10.1016/j.tcs.2005.07.003.

[50] Kenneth L. McMillan. "Lazy abstraction with interpolants". In: *Computer-Aided Verification (CAV).* Vol. 4144. LNCS. Springer, 2006, pp. 123–136. DOI: 10.1007/11817963_14.

[51] Kenneth L. McMillan, A. Kuehlmann, and Mooly Sagiv. "Generalizing DPLL to richer logics". In: *Computer-aided verification (CAV).* Vol. 5643. LNCS. 2009, pp. 462–476. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_35.

[52] John E. Mitchell. "Branch-and-cut algorithms for combinatorial optimization problems". In: *Handbook of applied optimization.* Ed. by Panos M. Pardalos and Mauricio G. C. Resende. Oxford University Press, 2002. Chap. 3.3. ISBN: 0-19-512594-0. URL: http://homepages.rpi.edu/~mitchj/papers/bc_hao.pdf.

[53] David Monniaux. "A quantifier elimination algorithm for linear real arithmetic". In: *Logic for Programming Artificial Intelligence and Reasoning (LPAR).* LNCS 5330. Springer, 2008, pp. 243–257. ISBN: 978-3-540-89439-1. DOI: 10.1007/978-3-540-89439-1_18. arXiv: 0803.1575.

[54] David Monniaux. "On using floating-point computations to help an exact linear arithmetic decision procedure". In: *Computer-aided verification (CAV).* LNCS 5643. Springer, 2009, pp. 570–583. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_42.

[55] David Monniaux. "Quantifier elimination by lazy model enumeration". In: *Computer-aided verification (CAV).* LNCS 6174. Springer, 2010, pp. 585–599. ISBN: 3642142958. DOI: 10.1007/978-3-642-14295-6_51.

[56] Matthew W. Moskewicz et al. "Chaff: engineering an efficient SAT solver". In: *Design Automation Conference (DAC).* Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379017.

[57] Leonardo Mendonça de Moura. personal communication.

[58] Leonardo Mendonça de Moura and Dejan Jovanović. "A model-constructing satisfiability calculus". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI).* Vol. 7737. LNCS. Springer, 2013, pp. 1–12. DOI: 10.1007/978-3-642-35873-9_1.

[59] Diego Caminha Barbosa de Oliveira and David Monniaux. "Experiments on the feasibility of using a floating-point simplex in an SMT solver". In: *Workshop on Practical Aspects of Automated Reasoning (PAAR).* Vol. 21. EPiC Series. Easychair, 2012. URL: http://www.easychair.org/publications/?page=797429640.

[60] Anh-Dung Phan, Nikolaj Bjørner, and David Monniaux. "Anatomy of alternating quantifier satisfiability (work in progress)". In: *10th International Workshop on Satisfiability Modulo Theories (SMT)*. 2012. HAL: hal-00716323.

[61] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *Supercomputing*. New York, NY, USA: ACM, 1991, pp. 4–13. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125848.

[62] Mohab Safey El Din and Éric Schost. "Polar varieties and computation of one point in each connected component of a smooth real algebraic set". In: *ISSAC*. ACM, 2003, pp. 224–231. DOI: 10.1145/860854.860901.

[63] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 1998. ISBN: 0471982326.

[64] Roberto Sebastiani and Silvia Tomasi. "Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions". In: *Automated reasoning (IJCAR)*. 2012, pp. 484–498. DOI: 10.1007/978-3-642-31365-3_38.

[65] Rahul Sharma, Aditya V. Nori, and Alex Aiken. "Interpolants as classifiers". In: *Computer-Aided Verification (CAV)*. Vol. 7358. LNCS. Springer, 2012, pp. 71–87. DOI: 10.1007/978-3-642-31424-7_11.

[66] William A. Stein. *Modular forms, a computational approach*. Vol. 79. Graduate studies in mathematics. AMS, Feb. 2007. ISBN: 978-0821839607. URL: http://wstein.org/books/modform/modform/index.html.

[67] Grighori S. Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Springer, 1983, pp. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28.

[68] Hiroshi Unno and Tachio Terauchi. "Inferring simple solutions to recursion-free Horn clauses via sampling". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 9035. LNCS. Springer, 2015, pp. 149–163. DOI: 10.1007/978-3-662-46681-0_10.

[69] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993. ISBN: 0-262-23169-7.