Noname manuscript No.
(will be inserted by the editor)

OPTIMATHSAT: A Tool for Optimization Modulo Theories

Roberto Sebastiani · Patrick Trentin

the date of receipt and acceptance should be inserted later

Abstract Optimization Modulo Theories (OMT) is an extension of SMT which allows for finding models that optimize given objectives. OPTIMATHSAT is an OMT solver which allows for solving a list of optimization problems on SMT formulas with linear objective functions –on the Boolean, the rational and the integer domains, and on their combination thereof– including (partial weighted) MAXSMT. Multiple and heterogeneous objective functions can be combined together and handled either independently, or lexicographically, or in linear or min-max/max-min combinations. OPTIMATHSAT provides an incremental interface, it supports both an extended version of the SMT-LIBV2 language and a subset of the FLATZINC language, and can be interfaced via an API.

In this paper we describe OPTIMATHSAT[1] and its usage in full detail.

1 Introduction

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of first-order formulas with respect to background theories. In the last fifteen years very efficient SMT solvers have been developed, that combine the power of modern conflict-driven clause-learning (CDCL) SAT solvers [55] with the expressiveness of dedicated decision procedures ($\mathcal{T}$*-solvers*) for several first-order theories of practical interest and their combinations –e.g., linear arithmetic over the rationals ($\mathcal{LRA}$), the integers ($\mathcal{LIA}$) or their combination ($\mathcal{LIRA}$), non-linear arithmetic over the reals ($\mathcal{NRA}$) or the integers ($\mathcal{NLIA}$), arrays ($\mathcal{AR}$), bit-vectors ($\mathcal{BV}$), floating-point arithmetic ($\mathcal{FP}$). (See [62,69,17] for an overview.) This has brought previously-intractable problems at the reach of state-of-the-art SMT solvers, in particular in the domain of formal verification.

Many SMT problems of interest, however, require the capability of finding models that are *optimum* wrt. some objective functions. These problems are grouped under the umbrella term of *Optimization Modulo Theories – OMT* [61,32,35,70,71,48,51,22,23,73,72]. For instance, in SMT-based model checking with timed or hybrid systems, you may want to find executions which optimize the value of some parameter while fulfilling/violating some property –e.g., to find the minimum opening time interval for a rail-crossing causing a safety

DISI, University of Trento, via Sommarive 9, I-39123, Trento, Italy. Email: `name.surname@unitn.it`.

[1] This paper describes OPTIMATHSAT 1.4.2, which is the latest version available when it was submitted.

violation [71]. A non-exhaustive list of OMT applications can be found in [61,32,70,35, 51,71,22,48,73,74].

In this paper we present OPTIMATHSAT, an OMT tool extending the MATHSAT5 SMT solver [36], implementing the OMT procedures described in [70–74]. OPTIMATHSAT allows for solving a list of optimization problems on SMT formulas with linear objective functions –on the Boolean, the rational and the integer domains, and on their combination thereof– including (partial weighted) MAXSMT. Multiple and heterogeneous objective functions can be combined together and handled either independently, or lexicographically, or in linear or min-max/max-min combinations.

OPTIMATHSAT provides a push/pop interface for adding and removing objectives and pieces of formulas from the formula stack, which allows for reusing information from one optimization search to another to improve the global performance of the search. The tool can be used in two ways: via file exchange through its command line interface, or through its API. The former supports both an extended version of the SMT-LIBV2 language [16] and a subset of the FLATZINC language [4].

OPTIMATHSAT is freely available for research and evaluation purposes [7], and it is currently used in some innovative projects [76,59,60,19].

Content. The rest of this paper is structured as follows. §2 describes the main OMT procedures and functionalities which are implemented in OPTIMATHSAT; §3 describes OPTIMATHSAT in detail; §4 presents some empirical evaluations to showcase OPTIMATHSAT functionalities and performance; §5 briefly reviews some interesting applications of OPTIMATHSAT; §6 summarizes the related work; §7 concludes the paper and hints some future developments.

A much shorter (7 pages) and much less-detailed tool paper on OPTIMATHSAT has already been published at CAV 2015 conference [72]. OPTIMATHSAT is build on top of the MATHSAT5 SMT solver [36], so that it inherits all MATHSAT5 SMT functionalities (including, e.g., solving, unsat-core extraction [38], interpolation [37], predicate abstraction [47], and others), for which it behaves like a wrapper. Therefore, in this paper we describe only the functionalities of OPTIMATHSAT which are not in MATHSAT5.

2 Optimization Modulo Theories

In order to make the paper self-contained, in this section we describe the main OMT procedures and functionalities which are implemented in OPTIMATHSAT and which will be referred to in the next sections [70–74].

2.1 Basics on Lazy SMT Solving

We recall some background knowledge on lazy SMT solving. We assume a basic background knowledge on first-order logic and on CDCL SAT solving. We consider some first-order theory $\mathcal{T}$, and we restrict our interest to *ground* formulas/literals/atoms in the language of $\mathcal{T}$ ($\mathcal{T}$-formulas/literals/atoms hereafter). Theories of particular interest are, e.g., those of linear arithmetic over the rationals ($\mathcal{LRA}$), over the integers ($\mathcal{LIA}$) or their combination ($\mathcal{LIRA}$).

A *theory solver for* $\mathcal{T}$, $\mathcal{T}$*-solver*, is a procedure able to decide the $\mathcal{T}$-satisfiability of a conjunction/set μ of $\mathcal{T}$-literals. If μ is $\mathcal{T}$-unsatisfiable, then $\mathcal{T}$*-solver* returns UNSAT and a set/conjunction η of $\mathcal{T}$-literals in μ which was found $\mathcal{T}$-unsatisfiable; η is called a $\mathcal{T}$*-conflict set*, and $\neg\eta$ a $\mathcal{T}$*-conflict clause*. If μ is $\mathcal{T}$-satisfiable, then $\mathcal{T}$*-solver* returns SAT; it may also be able to return some unassigned $\mathcal{T}$-literal $l \not\in \mu$ from a set of all available $\mathcal{T}$-literals, s.t. $\{l_1, ..., l_n\} \models_{\mathcal{T}} l$, where $\{l_1, ..., l_n\} \subseteq \mu$. We call this process $\mathcal{T}$*-deduction* and $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ a $\mathcal{T}$*-deduction clause*. Notice that $\mathcal{T}$-conflict and $\mathcal{T}$-deduction clauses are valid in $\mathcal{T}$. We call them $\mathcal{T}$*-lemmas*.

Given a $\mathcal{T}$-formula φ, the formula φ^p obtained by rewriting each $\mathcal{T}$-atom in φ into a fresh atomic proposition is the *Boolean abstraction* of φ, and φ is the *refinement* of φ^p. Notationally, we indicate by φ^p and μ^p the Boolean abstractions of φ and μ, and by φ and μ the refinements of φ^p and μ^p respectively. We say that the truth assignment μ *propositionally satisfies* the formula φ, written $\mu \models_p \varphi$, if $\mu^p \models \varphi^p$.

In a lazy SMT($\mathcal{T}$) solver, the Boolean abstraction φ^p of the input formula φ is given as input to a CDCL SAT solver, and whenever a satisfying assignment μ^p is found s.t. $\mu^p \models \varphi^p$, the corresponding set of $\mathcal{T}$-literals μ is fed to the $\mathcal{T}$*-solver*; if μ is found $\mathcal{T}$-consistent, then φ is $\mathcal{T}$-consistent; otherwise, $\mathcal{T}$*-solver* returns a $\mathcal{T}$-conflict set η causing the inconsistency, so that the clause $\neg\eta^p$ is used to drive the backjumping and learning mechanism of the SAT solver. The process proceeds until either a $\mathcal{T}$-consistent assignment μ is found (φ is $\mathcal{T}$-satisfiable), or no more assignments are available (φ is $\mathcal{T}$-unsatisfiable).

Important optimizations are *early pruning* and $\mathcal{T}$*-propagation*. The $\mathcal{T}$*-solver* is invoked also when an assignment μ is still under construction: if it is $\mathcal{T}$-unsatisfiable, then the procedure backtracks, without exploring the (possibly many) extensions of μ; if it is $\mathcal{T}$-satisfiable, and if the $\mathcal{T}$*-solver* is able to perform a $\mathcal{T}$-deduction $\{l_1, ..., l_n\} \models_{\mathcal{T}} l$, then l can be unit-propagated, and the $\mathcal{T}$-deduction clause $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ can be used in backjumping and learning. To this extent, in order to maximize the efficiency, most $\mathcal{T}$-solvers are *incremental* and *backtrackable*, that is, they are called via a push/pop interface, maintaining and reusing the status of the search from one call to the other.

Many modern SMT solvers, including MATHSAT5, provide a *stack-based incremental interface* (see e.g. [41]), by which it is possible to push/pop sub-formulas ϕ_i into a stack of formulas $\Phi \overset{\text{def}}{=} \{\phi_1, ..., \phi_k\}$, and then to check incrementally the satisfiability of $\bigwedge_{i=1}^{k} \phi_i$. The interface maintains the *status* of the search from one call to the other, in particular it records the *learned clauses* (plus other information). Consequently, when invoked on Φ, the solver can reuse a clause C which was learned during a previous call on some Φ' if C was derived only from clauses which are still in Φ.

2.2 OMT ($\mathcal{LRA} \cup \mathcal{T}$)

In what follows, $\mathcal{T}$ is some stably-infinite theory with equality (or a combination thereof) s.t. $\mathcal{LRA}$ and $\mathcal{T}$ are signature-disjoint. Here we consider only the minimization of objectives, since maximization is dual. We call an *Optimization Modulo* $\mathcal{LRA} \cup \mathcal{T}$ *problem, OMT*($\mathcal{LRA} \cup \mathcal{T}$), a pair $\langle \varphi, obj \rangle$ such that φ is an SMT($\mathcal{LRA} \cup \mathcal{T}$) formula and *obj* is an $\mathcal{LRA}$ variable occurring in φ, representing the cost to be minimized. The problem consists in finding an $\mathcal{LRA}$-model $\mathcal{M}$ for φ (if any) whose value of *obj* is minimum. If φ is in the form $\varphi' \wedge (obj < c)$ [resp. $\varphi' \wedge \neg(obj < c)$] for some value $c \in \mathbb{Q}$, then we call c an *upper bound* [resp. *lower bound*] for obj. If ub [resp. lb] is the minimum upper bound [resp. the maximum lower bound] for φ, we also call the interval $[\mathsf{lb}, \mathsf{ub}[$ the *range* of obj.

Notice that with regard to upper and lower bounds we follow the same convention which was established in [70,71]. Therefore, in minimization the upper bound ub is always considered to be strict, since this value is typically derived from a previous iteration of the optimization search in which a model $\mathcal{I}$ with cost ub was found, whereas the lower bound lb is always considered non-strict, since this is usually derived from a previous iteration of the optimization search in which the conjunction $\varphi \wedge (obj < \mathsf{lb})$ was found to be unsatisfiable. The interpretation of upper and lower bounds is dual when maximizing. For consistency reasons, the same interpretation is also applied on any user-provided value.

2.2.1 The Inline OMT Schema

Unlike with the *offline OMT schema* taken by other OMT solvers [51,22,23], in which the SMT solver is used as a black-box and the optimization proceeds through a sequence of incremental SMT calls, OPTIMATHSAT implements the *inline OMT schema* of [70,71], in which the whole optimization procedure is pushed inside the SMT solver by embedding the range-minimization loop inside the CDCL Boolean-search loop of the standard lazy SMT schema [69,17]. Although harder to maintain, in our experience the online schema architecture has better performance than the offline one for OPTIMATHSAT [71].

The procedure takes as input a pair $\langle \varphi, obj \rangle$, plus optionally values for lb and ub (which are implicitly considered to be $-\infty$ and $+\infty$ if not present), and returns the model $\mathcal{M}$ of minimum cost and its cost $\mathsf{u} \overset{\text{def}}{=} \mathcal{M}(obj)$; it returns the value $+\infty$ and an empty model if φ is $\mathcal{LRA}$-inconsistent. The standard CDCL-based schema of the SMT solver, which is thus called only once, is modified as follows.

Initialization. The variables l, u and pivot (current values of lower bound, upper bound and pivot) are added inside the SMT solver, and are initialized as $\mathsf{l} \leftarrow \mathsf{lb}$, $\mathsf{u} \leftarrow \mathsf{ub}$, and $\mathsf{pivot} \leftarrow \mathsf{ub}$.

Range Updating & Pivoting. Every time the search of the CDCL SAT solver gets back to decision level 0, the range $[\mathsf{l}, \mathsf{u}[$ is updated s.t. u [resp. l] is assigned the lowest [resp. highest] value u_i [resp. l_i] such that the atom $(obj < \mathsf{u}_i)$ [resp. $\neg(obj < \mathsf{l}_i)$] is currently assigned at level 0. Then a heuristic function $\mathsf{BinSearchMode}()$ is invoked, which decides whether to work in *linear-search mode* or in *binary-search mode* in the current branch: in the latter case, a heuristic function $\mathsf{ComputePivot}()$ computes a value $\mathsf{pivot} \in]\mathsf{l}, \mathsf{u}[$, and the (possibly new) atom $\mathsf{PIV} \overset{\text{def}}{=} (obj < \mathsf{pivot})$ is decided to be true (level 1) by the internal SAT solver. This restricts temporarily the cost range to $[\mathsf{l}, \mathsf{pivot}[$.

Termination. If $\mathsf{u} \leq \mathsf{l}$, or two literals $l, \neg l$ are both assigned at level 0, then the procedure terminates, returning the current value of u and $\mathcal{M}$.

Decreasing the Upper Bound. When an assignment μ is generated s.t. $\mu^p \models \varphi^p$ and which is found $\mathcal{LRA}$-consistent by $\mathcal{LRA}$-Solver, μ is also fed to $\mathcal{LRA}$-$\mathsf{Minimize}$, returning the minimum cost min of μ; then the unit clause

$$C_\mu \overset{\text{def}}{=} (obj < \mathsf{min}) \qquad (1)$$

is learned and fed to the backjumping mechanism, which forces the SAT solver to backjump to level 0 and unit-propagate $(obj < \mathsf{min})$. This restricts the cost range to $[\mathsf{l}, \mathsf{min}[$. $\mathcal{LRA}$-$\mathsf{Minimize}$ is embedded within the $\mathcal{LRA}$-Solver –it is a simple extension of the LP algorithm in [39]– so that it is called incrementally after it, without restarting its search from scratch. Notice that the clauses C_μ ensure progress in the minimization every time that a new $\mathcal{LRA}$-consistent assignment is generated.

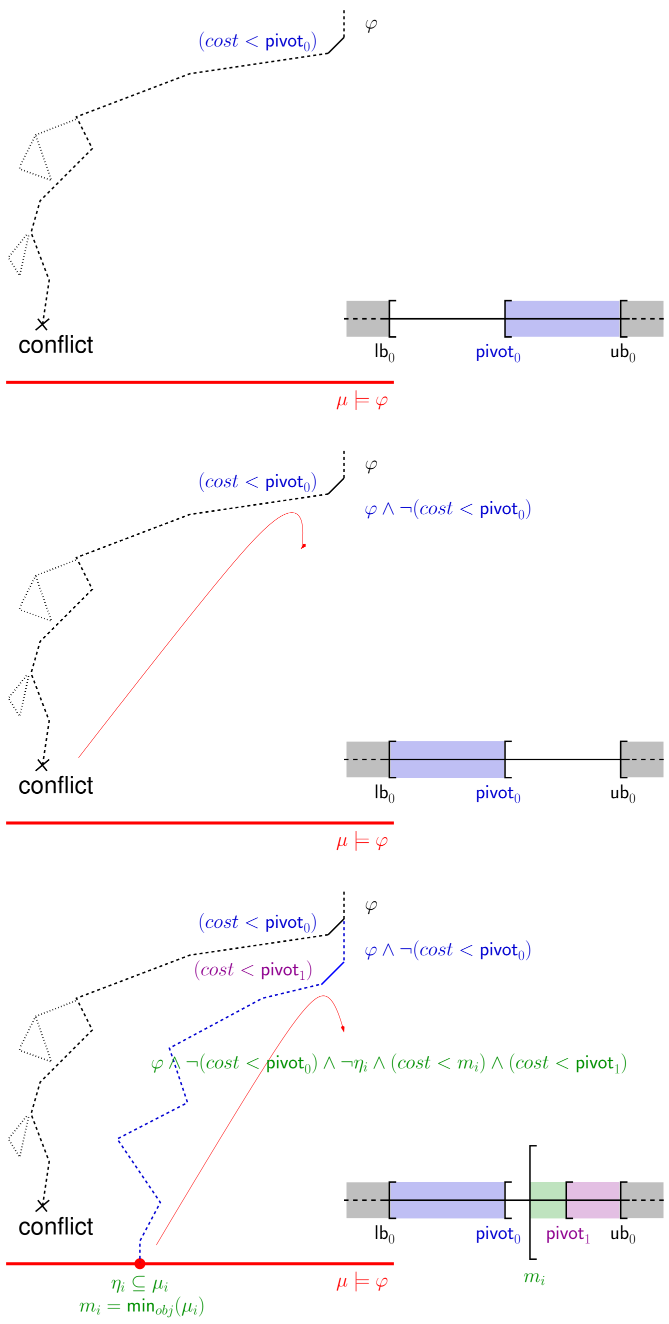

φ
$(cost < \text{pivot}_0)$
conflict
$\mu \models \varphi$
lb_0
pivot_0
ub_0
φ
$(cost < \text{pivot}_0)$
$\varphi \wedge \neg(cost < \text{pivot}_0)$
conflict
$\mu \models \varphi$
lb_0
pivot_0
ub_0
φ
$(cost < \text{pivot}_0)$
$\varphi \wedge \neg(cost < \text{pivot}_0)$
$(cost < \text{pivot}_1)$
$\varphi \wedge \neg(cost < \text{pivot}_0) \wedge \neg\eta_i \wedge (cost < m_i) \wedge (cost < \text{pivot}_1)$
conflict
$\eta_i \subseteq \mu_i$
$m_i = \text{min}_{obj}(\mu_i)$
$\mu \models \varphi$
lb_0
pivot_0
pivot_1
ub_0
m_i

Fig. 1: One piece of possible execution of the inline OMT procedure. (i) Pivoting on $(obj < \text{pivot}_0)$. (ii) Increasing the lower bound to pivot_0. (iii) Decreasing the upper bound to $\text{min}_{obj}(\mu_i)$.

We also have the following typical scenario, which is depicted in Figure 1.

Increasing the Lower Bound. In binary-search mode, when a conflict occurs and the conflict analysis of the SAT solver produces a conflict clause in the form $\neg(obj < \mathsf{pivot}) \vee \neg\eta'$ s.t. all literals in η' are assigned true at level 0 (i.e., $\varphi \wedge (obj < \mathsf{pivot})$ is $\mathcal{LRA}$-inconsistent), then the SAT solver backtracks to level 0, unit-propagating $\neg(obj < \mathsf{pivot})$. This case permanently restricts the cost range to $[\mathsf{pivot}, \mathsf{u}[$.

We recall a few facts about binary search [70,71]. First, binary search can be activated only when $\mathsf{u} \neq +\infty$ and $\mathsf{l} \neq -\infty$: when not so, $\mathsf{BinSearchMode}()$ returns false so that binary-search mode is not activated.

Second, to avoid getting stuck into Zeno's infinite loops,[2] each time halving the cost range right-bound —e.g., $[-1, 0[$, $[-\frac{1}{2}, 0[$, $[-\frac{1}{4}, 0[$, $[-\frac{1}{8}, 0[$, ...— binary-search steps are interleaved with linear-search ones. (We have empirically experienced the best performance with one linear-search step after every binary-search one, always starting the search in linear-search mode.)

Third, a binary-search step may improve relevantly and cheaply the current minimum if $\varphi \wedge (obj < \mathsf{pivot})$ is satisfiable, but can be very expensive otherwise, because detecting $\mathcal{T}$-unsatisfiability is typically much more expensive than finding one more model [3]. To this extent, the *aggressiveness* of binary search steps can be established by setting the value returned by $\mathsf{ComputePivot}()$: the nearer pivot is to the current lower bound l, the higher the improvement if satisfiable, the higher the chances of being unsatisfiable. Also, an *adaptive* version of $\mathsf{BinSearchMode}()$ can decide the next search mode according to the ratio between the progress obtained in the latest binary- and linear-search steps and their respective costs.

We refer the reader to [71] for details and for a description of further improvements to the OMT inline procedure.

2.2.2 Handling OMT($\mathcal{LRA} \cup \mathcal{T}$): Theory Combination

As described in [71], the implementation of an inline OMT($\mathcal{LRA} \cup \mathcal{T}$) procedure comes nearly for free if the SMT solver handles $\mathcal{LRA} \cup \mathcal{T}$-solving by *Delayed Theory Combination* [28], with the strategy of automatically case-splitting disequalities $\neg(x_i = x_j)$ into the two inequalities $(x_i < x_j)$ and $(x_i > x_j)$, which is implemented in MathSAT5. If so, the solver enumerates truth assignments in the form $\mu' \overset{\text{def}}{=} \mu_{\mathcal{LRA}} \cup \mu_{eid} \cup \mu_{\mathcal{T}}$, where (i) μ' propositionally satisfies φ, (ii) μ_{eid} is a set of interface equalities $(x_i = x_j)$ and disequalities $\neg(x_i = x_j)$, containing also one inequality in $\{(x_i < x_j), (x_i > x_j)\}$ for every $\neg(x_i = x_j) \in \mu_{eid}$; then $\mu'_{\mathcal{LRA}} \overset{\text{def}}{=} \mu_{\mathcal{LRA}} \cup \mu_{ei}$ and $\mu'_{\mathcal{T}} \overset{\text{def}}{=} \mu_{\mathcal{T}} \cup \mu_{ed}$ are passed to the $\mathcal{LRA}$-Solver and $\mathcal{T}$*-solver* respectively, μ_{ei} and μ_{ed} being obtained from μ_{eid} by dropping the disequalities and inequalities respectively.

If this is the case, it suffices to apply $\mathcal{LRA}$-$\mathsf{Minimize}$ to $\mu'_{\mathcal{LRA}}$, then learn $(obj < \mathsf{min})$ and use it for backjumping, as in §2.2.1.

A much more detailed description and justification is presented in [71].

2.3 OMT ($\mathcal{LIA} \cup \mathcal{T}$)

The inline OMT($\mathcal{LRA}$) schema in §2.2 is adapted to $\mathcal{LIA}$ and $\mathcal{LIRA}$ by exploiting the efficient $\mathcal{LIRA}$-Solver implemented in MathSAT5 [43], by embedding into it an $\mathcal{LIRA}$-

[2] In the famous Zeno's paradox, Achilles never reaches the tortoise for a similar reason.

[3] This observation is empirical and based on our experience with OMT ($\mathcal{LIRA}$) (see, e.g., [70,71,73]).

specific minimizing procedure, namely $\mathcal{LIRA}$-Minimize, which is called after the $\mathcal{LIRA}$-Solver each time the latter has checked the $\mathcal{LIRA}$-consistency of an assignment μ, s.t. $\mu^p \models \varphi^p$.

Check for lower-boundedness. The first step performed by $\mathcal{LIRA}$-Minimize is to check whether *obj* is lower bounded or not. Since a *feasible MILP* problem is unbounded if and only if its corresponding continuous relaxation is unbounded [31],[4] we run $\mathcal{LRA}$-Minimize on the relaxation of μ. If the continuous relaxation of μ is unbounded, then $\mathcal{LIA}$-Minimize returns $-\infty$; otherwise, $\mathcal{LRA}$-Minimize returns the minimum value c of *obj* in the continuous relaxation of μ. If every Integer variable in the problem is already assigned an integral value in the tableau found by $\mathcal{LRA}$-Minimize, then c is also the optimal solution for $\mathcal{LIA}$-Minimize and can be returned to the caller. Otherwise, c is a *lower bound* for *obj* in the propositional model μ currently being considered, and it is thus set as initial lb for *obj* in the subsequent branch&bound search. Similarly, the initial *upper bound* ub for *obj* in the subsequent branch&bound search is initialized to the integral value $\mathcal{M}(obj)$, where $\mathcal{M}$ is the model returned by the most recent call to the $\mathcal{LIRA}$-Solver on μ.

Branch&Bound. Then we explore the solution space by means of an LP-based Branch&Bound procedure that reduces the original MILP problem to a sequence of smaller sub-problems, which are solved separately. This is built on top of the $\mathcal{LIRA}$-Solver of MATHSAT5 and takes advantage of all the advanced features for performance optimization that are already implemented there [43]. In particular, we re-use its very-efficient internal Branch&Bound procedure for $\mathcal{LIRA}$-solving, which exploits historical information to drive the search and achieves higher pruning by *back-jumping* within the Branch&Bound search tree, driven by the analysis of unsatisfiable cores. (We refer the reader to [43] for details.)

Truncated Branch&Bound. We have also implemented a "sub-optimum" variant of $\mathcal{LIRA}$-Minimize in which the inner $\mathcal{LIRA}$-Solver minimization procedure stops as soon as either it finds its first solution or it reaches a certain limit on the number of branching steps. This is typically much faster than the complete B&B procedure. The drawback is that, in some cases, it analyzes a truth assignment μ (augmented with the extra constraint $(obj < \mathsf{ub})$) more than once.

We refer the reader to [73] for details.

2.4 OMT(PB) / MAXSMT

An important sub-case of OMT is that with pseudo-Boolean objective functions $obj \overset{\text{def}}{=} \sum_i w_i A_i$, OMT(PB), where the w_is are rational constants and the A_is are Boolean variables whose values are interpreted as $\{0, 1\}$. Alternatively, the same problem can be formulated as a (partial weighted) MAXSMT given by a pair $\langle \varphi_h, \varphi_s \rangle$, where φ_h is a set of "hard" $\mathcal{T}$-clauses and φ_s is a set of positive-weighted "soft" $\mathcal{T}$-clauses C_i each associated with some positive weight w_i, and the goal is to find the maximum-weight set of $\mathcal{T}$-clauses ψ_s, $\psi_s \subseteq \varphi_s$, s.t. $\varphi_h \cup \psi_s$ is $\mathcal{T}$-satisfiable [61,32,14,35].

[4] As in [31], by "continuous relaxation" –henceforth simply "relaxation"– we mean that the integrality constraints on the integer variables are relaxed, so that they can take fractional values.

The two problems can be reduced to each other. Given an OMT(PB) problem $\langle \varphi, \sum_i w_i A_i \rangle$, we can set $\langle \varphi_h, \varphi_s \rangle$ s.t.

$$\varphi_h \stackrel{\text{def}}{=} \varphi, \quad \varphi_s \stackrel{\text{def}}{=} \{C_i\}_i \; s.t. \; C_i \stackrel{\text{def}}{=} \neg A_i \; \text{ of weight } w_i. \tag{2}$$

Vice versa, a MAXSMT problem $\langle \varphi_h, \varphi_s \rangle$ can be encoded into an OMT(PB) problem $\langle \varphi, \sum_i w_i A_i \rangle$ as follows:

$$\varphi \stackrel{\text{def}}{=} \varphi_h \cup \bigcup_{C_i \in \varphi_s} \{(A_i \vee C_i)\}, \quad A_i \text{ fresh Boolean variables.} \tag{3}$$

2.4.1 MAXSMT*-specific techniques*

One approach for dealing with MAXSMT and OMT with PB objectives is to embed some MAXSAT engine within the SMT solver itself, and use it in combination with dedicated *$\mathcal{T}$-solvers* [35,22,23]. Some examples of this approach can be found in the literature:

- *Lemma-Lifting* [35]. Cimatti et al. [35] presented a "modular" approach for MaxSMT, combining a lazy SMT solver with a MaxSAT solver, which can be used as black-boxes, where the SMT solver is used as an oracle generating $\mathcal{T}$-lemmas that are then learned by the MAXSAT solver so as to progressively narrow the search space toward the optimal solution.
- *SMT with Core-Guided* MAXSAT *Resolution (*MAXRES*)* [58,22]. In [58], Narodytska et al. first presented a core-guided MAXSAT approach based on MAXSAT resolution, where they used this inference rule to construct a sequence of SAT formulas that only results in a linear growth of the initial formula. This very efficient MAXSAT engine was later on integrated in the Z3 OMT solver by Bjorner et al., [22], to deal with MAXSMT problems.

2.4.2 MAXSMT *as OMT*

As described in [71], an alternative approach for dealing with a MAXSMT problem $\langle \varphi_h, \varphi_s \rangle$ is to encode it into an OMT($\mathcal{LRA} \cup \mathcal{T}$) problem, by first encoding it into OMT(PB) as in (3), and then encoding the problem into the pair $\langle \varphi, obj \rangle$ defined as:

$$\varphi \stackrel{\text{def}}{=} \varphi^* \wedge \bigwedge_i ((\neg A_i \vee (x_i = w_i)) \wedge (A_i \vee (x_i = 0))) \wedge \tag{4}$$
$$\bigwedge_i ((0 \leq x_i) \wedge (x_i \leq w_i)) \wedge \tag{5}$$
$$(obj = \sum_i x_i), \qquad x_i, \; obj \text{ fresh rational variables.} \tag{6}$$

where $\varphi^* \stackrel{\text{def}}{=} \varphi_h \cup \bigcup_{C_i \in \varphi_s} \{(A_i \vee C_i)\}$ and each A_i is a fresh Boolean variable.

Notice that, although redundant from a logical perspective, the constraints in (5) serve the important purpose of allowing early-pruning calls to the $\mathcal{LRA}$-Solver (see [17]) to detect a possible $\mathcal{LRA}$ inconsistency among the current partial truth assignment over variables A_i and linear cuts in the form $\neg(\mathsf{ub} \leq obj)$ that are pushed on the formula stack by the OMT solver during the minimization of obj. The presence of such constraints improves performance significantly.

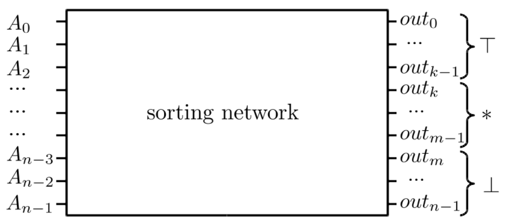


Fig. 2: The basic schema of a sorting network relation

2.4.3 MAXSMT *as OMT plus Sorting Networks*

On the one hand, although MAXSAT-based approaches to MAXSMT can be very efficient (see [35,22,23,74]), they can be an unfeasible choice in some OMT applications in which the objective function is given by either a linear combination of Pseudo-Boolean and arithmetic terms (like, e.g., for Linear Generalized Disjunctive Programming problems [71]) or a non-trivial combination of several PB sums as in [76].

On the other hand, the OMT-based approach for MAXSMT problems $\langle \varphi_h, \varphi_s \rangle$, described in §2.4.2, can suffer from poor performance when dealing with PB objectives where all soft constraints share the same weight value w:

$$obj = w \cdot \sum_{i=0}^{n-1} A_i. \tag{7}$$

As a matter of fact, the OMT solver can end up generating exponentially many Theory Lemmas along the optimization search for objective obj [74].

The solution to this efficiency issue we adopted in OPTIMATHSAT is to better exploit Boolean Constraint Propagation (BCP) by means of *bidirectional sorting-networks* [74]. A sorting-network relation, depicted in figure 2, takes $A_0, ..., A_{n-1}$ Boolean variables as input and returns $out_0, ..., out_{n-1}$ variables as output s.t., if in the current (partial) truth assignment μ, k variables are set to True, $n-m$ variables are set to False and $m-k$ are unassigned, then by Boolean Constraint Propagation $out_0, ..., out_{k-1}$ are set to True, $out_m, ..., out_{n-1}$ are set to False and $out_k, ..., out_{m-1}$ are not propagated. In a *bidirectional* sorting-network, if out_{k-1} is forced to be True (that is, at least k inputs must be True) and $n-k$ inputs A_i are False, then by Boolean Propagation all other unassigned A_is are automatically set to True; vice-versa, if out_{k+1} is forced to be False (that is, at most k inputs can be True) and $n-k$ inputs A_i are True, then all other unassigned A_is are automatically set to False.

Given a bidirectional sorting network C, an OMT problem $\langle \varphi, w \cdot \sum_{i=0}^{n-1} A_i \rangle$ can be encoded as $\langle \varphi', obj \rangle$ where obj is a fresh rational variable:

$$\varphi' = \varphi \wedge C \wedge \bigwedge_{k=0}^{k=n-1} \begin{cases} out_k \rightarrow ((k+1) \cdot w \leq obj) \\ \neg out_k \rightarrow (obj \leq k \cdot w) \\ \neg((k+1) \cdot w \leq obj) \vee \neg(obj \leq k \cdot w). \end{cases} \tag{8}$$

The benefit of this extension is that, whenever the optimization search finds a new satisfiable truth assignment μ which sets k variables A_i to True, so that a unit clause in the form $(obj < k \cdot w)$ is learned, then as soon as $k-1$ inputs A_i are assigned to True the remaining

$n - k + 1$ inputs are set to False by BCP. (A dual case occurs when some lower-bound unit clause $(obj > k \cdot w)$ is learned.)

This extension generalizes to the case in which groups of terms share the same weight, in which one cardinality network is generated for every group (see [74] for details):

$$obj = \sum_k w_k \cdot (\sum_{i_k=1}^{N_k} A_{i_k}). \tag{9}$$

In OPTIMATHSAT we have implemented two bidirectional sorting-networks: the sequential counter encoding [75] of size $O(n^2)$, and the cardinality network encoding [11] of size $O(n \log^2 n)$. We refer the reader to [74] for more details.

2.5 OMT with Multiple Objectives

OMT is not limited to dealing with a unique objective function, in fact, each formula φ might contain several objectives $obj_1, ..., obj_N$ which are then combined according to a set of configurable strategies. Importantly, notice that each obj_i can be indifferently an $\mathcal{LRA}$, an $\mathcal{LIA}$, an $\mathcal{LIRA}$ or a PB/MaxSMT objective. Currently, OPTIMATHSAT supports the following forms of multi-objective OMT.

2.5.1 Multiple-independent-objective OMT

A *multiple–independent-objective OMT problem* [51,22,73], a.k.a. *boxed OMT*, is given by a pair $\langle \varphi, \mathcal{O} \rangle$ s.t. $\mathcal{O} \overset{\text{def}}{=} \{obj_1, ..., obj_N\}$, and consists in finding in one single run a set of models $\{\mathcal{M}_1, ..., \mathcal{M}_N\}$ s.t. each $\mathcal{M}_i$ makes obj_i minimum on the common formula φ. Although the same result can be obtained by sequentially solving the sequence $\langle \varphi, obj_1 \rangle, ..., \langle \varphi, obj_N \rangle$ of single-objective OMT problems, in practice this optimization technique significantly improves the global performance because it allows for factorizing the search [51,22,73].

In OPTIMATHSAT the algorithm proceeds as follows. Initially, we set a list of watched objectives $\mathcal{O}' = \mathcal{O}$ and, for every $obj_i \in \mathcal{O}$, the initial upper bound u_i of obj_i is set to $+\infty$ and the model $\mathcal{M}_i$ is set to $\emptyset$. Every time a novel consistent truth assignment μ propositionally satisfying φ is found, the minimizer is invoked on μ *for each watched objective* $obj_i \in \mathcal{O}'$, returning the model $\mathcal{M}'_i$ and the corresponding minimum value u'_i. If $u'_i < u_i$, then we update $u_i = u'_i$ and $\mathcal{M}_i = \mathcal{M}'_i$. If $u'_i = -\infty$, then we update $u_i = u'_i$ and $\mathcal{M}_i = \mathcal{M}'_i$ and obj_i is removed from the list of watched objectives $\mathcal{O}'$. After all watched objectives have been checked, OPTIMATHSAT learns the clause:

$$C_\mu \overset{\text{def}}{=} \bigvee_{obj_i \in \mathcal{O}'} (obj_i < u_i) \tag{10}$$

and the SMT solver proceeds the search for another satisfiable truth assignment. The search terminates when $\mathcal{O}'$ is empty or when the SMT engine returns UNSAT. In either case, each u_i is the optimum solution value of the corresponding objective obj_i.

We refer the reader to [73] for more details and some improvements.

2.5.2 *Lexicographic OMT*

In lexicographic optimization a model $\mathcal{M}$ is produced which minimizes a prioritized list of objectives $\{obj_1, ..., obj_N\}$.

OPTIMATHSAT handles the lexicographic minimization in rounds, starting with obj_1. At each optimization round, OPTIMATHSAT finds the minimum value u_i for obj_i and the corresponding optimum model $\mathcal{M}_i$. If u_i is unbounded, or the there is no possible satisfiable solution, the search terminates. Otherwise, a clause in the form $(cost_i = u_i)$ is learned, and OPTIMATHSAT proceeds by incrementally optimizing the next objective obj_{i+1} in the list, if any. As described in more detail in [73], in both cases OPTIMATHSAT can apply a number of heuristic techniques to improve the search time.

It is worth noticing that each objective $\{obj_1, ..., obj_N\}$ can be of arbitrary type among the supported ones (e.g., some can be $\mathcal{LRA}$ costs, some be the sums of penalties of sub-groups of soft clauses in a MaxSMT sub-problem). For this reason, the lexicographic optimization framework described in this section does not (yet) contemplate specialized optimization techniques for the case in which all objective functions have the same type (e.g. when all obj_i are MAXSMT goals, some of the lexicographic extensions to MAXSAT presented in [54] could be adapted to deal with it).

2.5.3 *OMT with Linear Combinations of Objectives*

Given a list of objectives $\{obj_1, ..., obj_N\}$, it is possible to minimize arbitrary weighted sums $\sum_i w_i \cdot obj_i$.

In OPTIMATHSAT this is solved by simply encoding it into the OMT problem $\langle \varphi', obj \rangle$, where $\varphi' = \varphi \wedge (obj = \sum_i w_i \cdot obj_i)$ and obj is a fresh objective variable.

2.5.4 *OMT with Min-Max and Max-Min Combination of Objectives*

Given a list of objectives $\{obj_1, ..., obj_N\}$, *Min-Max OMT* consists in finding a model $\mathcal{M}$ which minimizes the maximum value of the obj_i's. (With *Max-Min OMT*, $\mathcal{M}$ maximizes their minimum value.)

In OPTIMATHSAT Min-Max OMT is solved by simply encoding it into the OMT problem $\langle \varphi \wedge \bigwedge_{i=1}^{N} (obj_i \le obj), obj \rangle$, where obj is a fresh objective variable. (The encoding for Max-Min OMT is dual.)

2.6 Incrementality in OMT

OPTIMATHSAT provides a push/pop interface for adding and removing objectives and pieces of formulas from the formula stack, which allows for reusing information from one optimization search to another to improve the global performance of the search [73]. This exploits MATHSAT5 incremental interface (see §2.1), and it is based on the very simple observation that in our inline OMT schema all learned clauses are either $\mathcal{T}$-lemmas or they are derived from $\mathcal{T}$-lemmas and some of the input subformulas ϕ_i's, with the exception of the clauses $C_\mu \overset{\text{def}}{=} (obj < \mathsf{min})$ (§2.2) [resp. $C_\mu \overset{\text{def}}{=} \bigvee_{obj_i \in \mathcal{C}^*} (obj_i < \mathsf{u}_i)$ (§2.5)] which are "artificially" introduced to ensure progress in the minimization steps, and for the unit clauses $(obj < \mathsf{pivot})$ and $\neg(obj < \mathsf{pivot})$ which may be learned in binary search (§2.2). Thus, in order to exploit MATHSAT5 incrementality, it suffices to drop only these clauses from one OMT call to the other, while preserving all the others, as with incremental SMT.

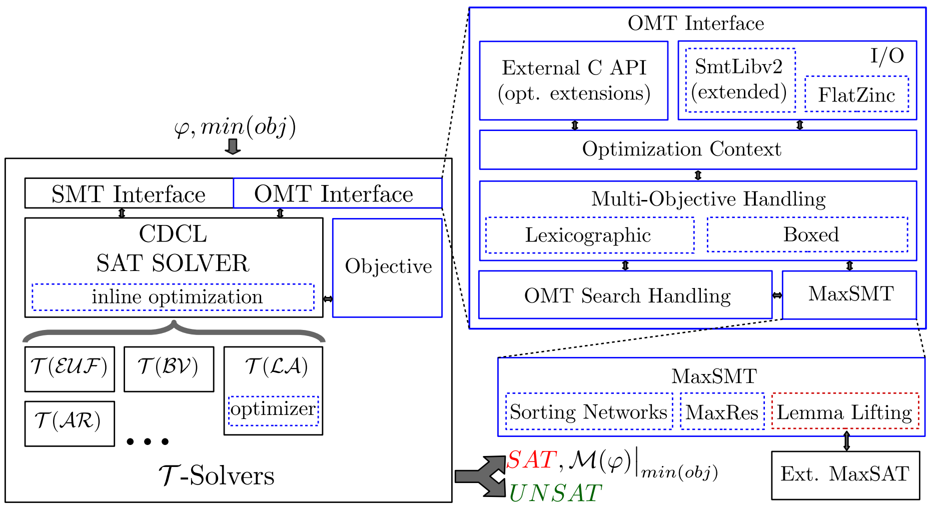


Fig. 3: High-level overview of OPTIMATHSAT architecture.
Legend: black components are part of the underlying SMT solver, blue ones are exclusively part of our OMT solver, and the only red component is an independent extension to MATHSAT5 that we ported to OPTIMATHSAT's code-base [35].

For a more detailed description of how incrementality is efficiently implemented in OPTIMATHSAT we refer the reader to [73].

3 OPTIMATHSAT

Like MATHSAT5, OPTIMATHSAT is developed in C++ and its binaries are made available for download on OPTIMATHSAT's web-page [7]. Currently, we support both *32-* and *64-bit x86* platforms for both Linux and Windows operating systems, and *64-bit x86* only for Mac OS X, which is also linked against Clang's `libc++` instead of `libstc++`.

OPTIMATHSAT inherits the license conditions of MATHSAT5, and it is thus made freely available for research and evaluation purposes only.

3.1 Architecture

A high-level overview of OPTIMATHSAT architecture is shown in Figure 3. OPTIMATHSAT extends the SMT Interface of MATHSAT5 with an OMT Interface implementing the optimization functionalities described in this paper, keeping the access to all functionalities of the underlying SMT solver. (Thus OPTIMATHSAT behaves as a wrapper of MATHSAT5 when none of its optimization, MAXSMT and Pseudo-Boolean extensions are used.)

The inline OMT schema of §2 is implemented by modifying the CDCL SAT solver in MATHSAT5 and its interface with the theory solvers (inline optimization block) and the $\mathcal{T}$*-solver* for linear arithmetic, which is enhanced with optimization procedures described in §2.2 and §2.3 (optimizer block).

The Objective block acts as a wrapper for the objective function and is meant to decouple the optimization search within the CDCL SAT engine from the specific type of the objective being optimized, making it more easily extensible to support new objectives (other than $\mathcal{LIRA}$ and PB/MAXSMT) in the future.

Most of the remaining functionalities introduced by OPTIMATHSAT are contained in its OMT Interface block, which is expanded in the right-hand side of Figure 3 and implemented as a set of self-contained components arranged in a stack of independent layers.

At the top level of the OMT Interface, an input/output layer (I/O block) contains the novel optimization extensions to both the External C API (§3.3.3) and the SMT-LIBv2 parser (§3.3.1), in addition to a brand-new parser for a subset of the FLATZINC standard (§3.3.2).

In the level below, the Optimization Context block manages the stack of objective functions, the configuration of the optimization search and other implementation details pertaining the OMT state.

The Multi-Objective Handling block is responsible for the high-level management of the optimization search. In particular, it selects the correct optimization algorithm for the given multi-objective combination (§2.5) selected in the configuration (e.g. Lexicographic/Boxed). [5] Moreover, in case the objective stack includes compatible MAXSMT/PB goals, it can forward these objectives to one of the available MAXSMT engines, according to the configuration.

The MAXSMT handling block has three main components: a set of generator functions for Sorting Networks (§2.4.3), a Maximum Resolution engine based on [58] and the porting into OPTIMATHSAT of the lemma-lifting approach presented in [35], which requires an external MAXSAT engine and was originally implemented as an independent extension to MATHSAT5.

The OMT Search Handling block mainly consists in appropriate hooks with the underlying SMT solver for enabling the optimization features within the CDCL SAT engine, and the retrieval of the optimum models.

3.2 Objectives in OPTIMATHSAT: a Compositional Approach

In OPTIMATHSAT, the availability of specialized techniques for dealing with MAXSMT makes it desirable to encode PB objectives as MAXSMT (§2.4). Nevertheless, in some OMT applications the objectives to optimize may be defined in terms of complex combinations of PB and $\mathcal{LIRA}$ objectives. This is the case, e.g., of *Structured Learning Modulo Theories* [76] (§5.1), where the objective function may take complex forms like that shown in (11) (see §4.3), or of Linear Generalized Disjunctive Programming (LGDP) [65], in which a goal is given by the linear combination of mixed Boolean/numeric objectives.

OPTIMATHSAT deals with this issue by adopting a *compositional definition of objectives*, according to which objective functions are no longer special, independent entities separated from the rest of the formula; rather, they are reusable objects that are treated in the same way as terms appearing in the original formula. As a consequence, OPTIMATHSAT allows for the arbitrary composition of objectives, by means of linear and Max-Min/Min-Max composition (§2.5), and for their Boxed or Lexicographic optimization (§2.5), regardless of the fact they are $\mathcal{LIRA}$ or PB objectives, or MAXSMT goals implicitly defined by the weights on soft constraints.

[5] Notice that the Linear, Max-Min and Min-Max combinations of objectives are handled instead by the Optimization Context block because this consists in a simple encoding, see §2.5.

In OPTIMATHSAT, an objective function can be viewed as a *term* occurring in the formula. We internally associate each objective with a fresh term of the correct type, which is optionally made available to the developer for later use. Semantically, we guarantee that the value of the associated term matches that of the objective function for the whole duration of the optimization search, thus acting as an *alias* for the objective itself.

Since setting constraints on objective functions has an impact on the outcome of the optimization search, OPTIMATHSAT normally protects the end user from improper inadvertent use by using masked internal variables. This safety guard can be circumvented by explicitly assigning a label to the objective function in the extended SMT-LIBV2 language, e.g.

```
(minimize (+ var_1 ... var_N) :id my_label)
...
(assert (= total (+ my_label ...)))
```

or by using one of the available *C API* functions, e.g.

```
msat_term internal_term = msat_objective_get_term(env, obj);
```

The reification of objectives into terms, which include MAXSMT goals, allows for combining an objective function with other terms in a formula so as to impose novel constraints or compose novel objectives. This can be done by linear and Max-Min/Min-Max composition (§2.5) because for any given set $\{obj_1, ..., obj_N\}$ of $\mathcal{LIRA}$ and MAXSMT/PB goals the optimum value is always of numerical type.

3.3 Input Interfaces

It is possible to use OPTIMATHSAT in two ways: directly, using its *command line* interface, or through its *API*. The *command line* interface accepts problem files encoded in

– an extended version of the SMT-LIBV2 format, and
– a subset of the FLATZINC language.

3.3.1 Extended SMT-LIBv2

One of the cornerstones of SMT is represented by the SMT-LIBV2 initiative, which developed and promoted common input and output languages for all SMT solvers. Conversely, the spectrum of OMT solvers is fragmented: there currently does not yet exist a common language that is both accepted by all OMT solvers and handled in the same way. In an effort to guarantee the maximum interoperability among solvers, and possibly move towards a de-facto standard, we redesigned the input language accepted by OPTIMATHSAT to be as compatible as possible with that of Z3 . Even so, as a consequence of the compositional approach adopted by OPTIMATHSAT (§3.2), there are still important and irreconcilable differences in the input formats accepted by OPTIMATHSAT and Z3 that may require additional design care and translation effort.

We provide a comprehensive list of the syntactic extensions to SMT-LIBV2 adopted by OPTIMATHSAT, together with a brief explanation. We use square brackets to highlight optional syntactic elements that can be omitted when not needed. Moreover, we allow for a `<numeral>` to be used in place of a `<decimal>` when the decimal part of the latter number is equal to zero.

```
(minimize <term> [:id <string>]
            [:local-lb <decimal>] [:local-ub <decimal>])
(maximize <term> [:id <string>]
            [:local-lb <decimal>] [:local-ub <decimal>])
```

`minimize` pushes the linear arithmetic `<term>` on the internal objective function stack to be minimized at the next `(check-sat)` call. Dual for `maximize`.

```
(minmax <term> ... <term> [:id <string>]
            [:local-lb <decimal>] [:local-ub <decimal>])
(maxmin <term> ... <term> [:id <string>]
            [:local-lb <decimal>] [:local-ub <decimal>])
```

`minmax`[6] pushes a fresh `<term>`, which optimum value matches the minimum maximum value of the argument list of terms `<term> ... <term>`, on the internal objective function stack. Dual for `maxmin`.

```
(assert-soft <term> [:id <string>]
            [:weight <decimal>] [:dweight <decimal>])
```

`assert-soft` adds `term` on the stack of soft clauses with weight `<decimal>` (1 if omitted).

All soft clauses with the same id are grouped into the same MAXSMT/PB objective; if no id is provided they are added to the default group I. `dweight` is completely interchangeable with `weight`, and it is kept for compatibility with legacy versions of Z3 . Moreover, in order to lift the burden of re-scaling weight values from the developer, we allow for zero and negative weights to be set on soft constraints and then we internally map the objective function to a purely-positive weighted PB term when optimizing.

An important difference wrt. Z3 is that, in OPTIMATHSAT the MAXSMT or PB objective term that is defined with `assert-soft` is not implicitly minimized. Instead, we require the user to explicitly invoke the minimization of the MAXSMT group identified by `<string>`, using the latter as a term. The advantage of this approach is that it enables arbitrary compositions of MAXSMT objectives and other $\mathcal{LIRA}$ objectives, which can be useful in particular applications, e.g. to build mixed Boolean/numeric objective functions [76,59] (see §3.2).

```
(set-option :opt.priority lex|box)
```

`opt.priority` configures the multi-objective combination to be used: LEX stands for *lexicographic* combination, whereas BOX stands for the (default) multi-independent combination. This option has no effect until `(check-sat)` is called, at which point all the objective functions previously pushed on the objective function stack (using one or more calls to `minimize`, `maximize`, `minmax`, or `maxmin`) –that were also not previously destroyed as a consequence to one (or more) `pop`– are combined *on-the-fly* and optimized as described in §2.5.

```
(set-model <numeral>)
```

`set-model` loads in the environment the satisfiable model associated with the objective having index `<numeral>` in the internal objective function stack (starting from 0). For easier handling of incremental formulas, a negative `numeral` is interpreted counting the objectives on the stack in the reverse order. Thus, indexes 0 and -1 always point to the oldest and most recent objective on the stack, respectively.

[6] This is a purely syntactic sugar extension for easiness of use.

Objective Naming. The attribute `:id` can be used to name an objective function, and associate a term with the same name and type to it. This has two useful applications. First, it can be used to retrieve the model value of the objective function using the standard SMT 2 command `(get-value (<string>))`. Second, it allows for linearly combining objective functions with one another, as described in §3.2[7].

Binary/Adaptive Search. Attributes `:local-lb` and `:local-ub` are optional supplementary information that the user can add to restrict the domain of an objective function. For `minimize` and `maxmin`, the lower bound is considered not strict, whereas the upper bound is considered strict. Dual for `maximize` and `minmax`. The non-strict bound is required by OPTIMATHSAT in order to perform the optimization search in either binary- or adaptive-search modes, as described in §2.2.1. For MAXSMT and PB-objectives, OPTIMATHSAT automatically determines local lower and upper bounds at runtime when the option `-opt.asoft.detect_bounds` is enabled. We remark that, in boxed multi-objective optimization, local bounds defined over an objective function do not affect the domain of other objectives in the same formula, even when two or more objective functions have one or more variables in common (e.g. x and $2x$) . This is because in boxed multi-objective optimization each objective is considered an independent goal within the same formula. Conversely, in single-objective and lexicographic optimization mode, local bounds can be thought to be equivalent to bounds with a global scope over the formula.

An Example. A small company wants to process an order of 1100 goods for a customer that needs the delivery to be made by the next day. For this task, it can allocate 3 identical machines M_0, M_1, M_2 with a maximum production capacity of 800 items per day and a small machine M_3 with a capacity of 200 items per day. Due to maintenance reasons, the expected production capacity of machines M_1 and M_2 is estimated to be temporarily limited to 500 and 600 items each. The company manager estimates the operating cost for machines M_0, M_1, M_2, and M_3 to be respectively $8, 9, 9$, and 5 euro per each produced unit. One goal is to find a production allocation that meets the demand such that (A) the overall production cost is minimized and, at tie, (B) the least number of machines is used, to limit wear.

An alternative goal, for the same problem, is to minimize the total cost (C), which includes both the production cost and the compensation for the employees processing this order. The company has already planned to employ 8 workers to package the goods, plus additional 2 workers for each machine that needs to be operated. The average daily cost for each worker, including taxes and other fees, is estimated to be equal to 78.5 euro.

This simple problem can be encoded in a lexicographic OMT formula, as shown in Figure 4. Figure 5, instead, shows a sample of OPTIMATHSAT output when given this problem, which is solved in negligible time.

Solving for (A) reveals that 8300 euro is the minimum production cost that meets the demand. However, due to the symmetry among machines M_1 and M_2, there are several ways to allocate the production so that it meets the demand at same minimum cost. The tie is broken by the secondary goal (B), which imposes a preference on a solution that allows either M_1 or M_2 to be completely shut down. We show that this is effectively the case by separately printing the optimum model found right after solving for goal (A), selected with

[7] Notice that SMT-LIBV2 makes available the `define-fun` command, which can be used to achieve similar goals. The "id" attribute was introduced over arbitrary objectives for consistency with the case of MAXSMT/PB goals defined with `assert-soft`. It also makes it simpler to identify objectives recombination at the implementation level, which may allow for introducing new techniques exploiting this knowledge in the future.

```
(set-option :produce-models true)  ; enable print model
(declare-fun production_cost () Real)
(declare-fun q0 () Real)           ; machine 'i' production load
(declare-fun q1 () Real)
(declare-fun q2 () Real)
(declare-fun q3 () Real)
(declare-fun m0 () Bool)           ; machine 'i' is used
(declare-fun m1 () Bool)
(declare-fun m2 () Bool)
(declare-fun m3 () Bool)
(assert (<= 1100 (+ q0 q1 q2 q3))) ; set goods quantity
(assert (and                       ; set goods produced per machine
    (and (<= 0 q0) (<= q0 800)) (and (<= 0 q1) (<= q1 500))
    (and (<= 0 q2) (<= q2 600)) (and (<= 0 q3) (<= q3 200))
))
(assert (and                       ; set machine 'used' flag
    (=> (< 0 q0) m0) (=> (< 0 q1) m1)
    (=> (< 0 q2) m2) (=> (< 0 q3) m3)
))
(assert (= production_cost (+ (* q0 8) (* q1 9) (* q2 9) (* q3 5)) ))
(assert-soft (not m0) :id used_machines)
(assert-soft (not m1) :id used_machines)
(assert-soft (not m2) :id used_machines)
(assert-soft (not m3) :id used_machines)
(push 1)
  ; optimize (A) and (B) lexicographically
(minimize production_cost)
(minimize used_machines)
(set-option :opt.priority lex)
(check-sat)
  ; print model for (A)
(set-model 0)
(get-value (production_cost)) (get-value (used_machines))
(get-value (q0)) (get-value (q1)) (get-value (q2)) (get-value (q3))
  ; print model for (B) after (A)
(set-model 1)
(get-value (production_cost)) (get-value (used_machines))
(get-value (q0)) (get-value (q1)) (get-value (q2)) (get-value (q3))
(pop 1)
  ; optimize (C), use :id to print model value
(minimize (+ production_cost (* (/ 785 10) (+ (* 2 used_machines) 8)))
          :id total_cost)
(set-option :opt.priority box)
(check-sat)
  ; print value of (C)
(set-model 0)
(get-value (total_cost))
```

Fig. 4: Sample SMT2 example.

`(set-model 0)`, and the optimum model after lexicographically optimizing for goal (B), selected with `(set-model 1)`.

The total cost (C) for processing the order is found to be equal to 9399 euro. Here, it should be noted that the particular encoding of goal (C) that we used in our example is made possible by the *compositional approach* of OPTIMATHSAT (see §3.2), which allows us to express *total_cost* as a linear combination of a $\mathcal{LIRA}$ term and the MAXSMT objective.

```
### MINIMIZATION STATS ###
# objective:      (`+_rat` (`*_rat` 5 q3) ... ) (index: 0)
# interval:     [ -INF , +INF ]
#
# Search terminated!
# Exact non-strict optimum found!
# Optimum: 8300
# Search steps: 1 (sat: 1)
#  - binary: 0 (sat: 0)
#  - linear: 1 (sat: 1)
# Restarts: 1 [session: 1]
# Decisions: 63 (0 random) [session: 63 (0 random)]
# Propagations: 225 (0 theory) [session: 243 (0 theory)]
# Watched clauses visited: 162 (119 binary) [session: 165 (122 binary)]
# Conflicts: 7 (7 theory) [session: 7 (7 theory)]
# Error:
#  - absolute: 0
#  - relative: 0
# Total time: 0.000 s
#  - first solution: 0.000 s
#  - optimization: 0.000 s
#  - certification: 0.000 s
# Memory used: 12.129 MB
...
sat
  ; model for (A)
( (production_cost 8300) )
( (used_machines 4) )
( (q0 800) )
( (q1 (/ 199999999 2000000)) )
( (q2 (/ 1 2000000)) )
( (q3 200) )
  ; model for (B) after (A)
( (production_cost 8300) )
( (used_machines 3) )
( (q0 800) )
( (q1 100) )
( (q2 0) )
( (q3 200) )
...
sat
  ; value of (C)
( (total_cost 9399) )
```

Fig. 5: Sample SMT2 example output.

This encoding is currently not legal in Z3, because it does not allow for linearly combining the `id` of a MaxSMT goal with other terms in the formula. Moreover, Z3 assumes that a MaxSMT goal is to be implicitly minimized at the location corresponding to the `assert-soft` definition. Therefore, as opposed to OptiMathSAT, Z3 assigns a higher priority value to $used\_machines$ than to $production\_cost$ in the lexicographic optimization of this example. In order to properly express the third goal (C) of this example, in Z3 one would have to define $used\_machines$ as a PB objective, without using the `assert-soft` statement.

3.3.2 FlatZinc / MiniZinc

Since version 1.4.0, OPTIMATHSAT supports a subset of the FLATZINC 1.6 standard [4], the easily-parsable and flattened counterpart of MINIZINC [6], a widely adopted high-level declarative language for modeling CSP problems. MINIZINC supports an extensive library of global constraints, three scalar types (Booleans, integers and floats), two compound types (sets and fixed-size arrays), if-then-else, let expressions, user-defined predicates and much more. For an in-depth overview of the FLATZINC and MINIZINC languages, we refer the reader to [4] and [6] respectively.

FLATZINC *Supported Subset.* OPTIMATHSAT currently supports only a subset of the FLATZINC language, which does not include any constraint making use of some trigonometric, power or logarithmic function. All other constraints are fully supported, except when they introduce some form of non-linear arithmetic in the encoding. In this case, the problem cannot be solved, as OPTIMATHSAT does not handle $\mathcal{NRA}$ yet. For what concerns basic FLATZINC types, `bool` is mapped into `Boolean`, `int` into `Integer`, `float` into high-precision Real, and `sets` are encoded with an occurrence representation using additional Boolean variables as witnesses of existence within the set (similarly to [12]). A more comprehensive list of the supported functionalities is available at [9].

Multi-Objective Extension. OPTIMATHSAT accepts an extended version of the FLATZINC format which allows for multiple objectives to be present in the same problem. These goals should be specified in a comma-separated list within the same `solve` constraint, e.g.

```
solve minimize goal_1, maximize goal_2;
```

The value of the `opt.priority` priority can be conveniently configured as in §3.4, so as to select the desired multi-objective optimization combination among the available independent and lexicographic approaches (see §2.5).

FlatZinc to SMT-LIBV2. By exploiting the internal *API* tracing capabilities, OPTIMATHSAT can be used as a tool for converting supported FLATZINC problems into SMT-LIBV2 formulas enriched with optimization extensions. This can be accomplished with the following command:

```
$ optimathsat -input=fzn \
              -debug.api_call_trace=1 \
              -debug.api_call_trace_dump_config=False \
              -debug.solver_enabled=False \
              -debug.api_call_trace_filename=output.smt2 \
              < input.fzn
```

Cutstock Example. We hereby show with an example how OPTIMATHSAT can be used to solve an instance of the well-known NP-hard Cutstock problem, taken from the MINIZINC distributed package [56]. The goal in this problem is to minimize the number of fixed-length stock material units used to obtain a number of fixed-size pieces.

Figure 6 contains the original MINIZINC formulation of the problem instance we consider, which we found within MINIZINC installation files. Constraint $C.1$ encodes the fact that the number of pieces carved out of each stock material unit cannot be negative, $C.2$ requires the number of produced pieces to be large enough to meet their corresponding demand, $C.3$ constraints the solution to not exceed the amount of stock material available for each unit, and last $C.4$ binds *obj* to be equal to the number of stock material units used.

```
%-----------------------------------------------------------%
% Jakob Puchinger <jakobp@cs.mu.oz.au>
% December 2007
%
% The cutting stock problem: Kantorovitch
% formulation for colgen.
%-----------------------------------------------------------%

int: L;                       % stock unit length
int: K;                       % max no. of stock units used
int: N;                       % number of pieces
array[1..N] of int: lengths;  % pieces length
array[1..N] of int: demands;  % pieces demand
array[1..K] of var 0..1: pieces;
                              % 1: stock used,
                              % 0: otherwise
array[1..K, 1..N] of var int: items;
                              % # pieces n cut
                              % from stock unit k
var int: obj;                 % objective

constraint % C.1
  forall(k in 1..K, i in 1..N) (
    items[k,i] >=0
  );

constraint % C.2
  forall(i in 1..N) (
    sum([ items[k, i] | k in 1..K]) >= demands[i]
  );

constraint % C.3
  forall( k in 1..K ) (
    sum(i in 1..N) (items[k,i] * lengths[i])
      <= pieces[k] * L
  );

constraint % C.4
  obj = sum([ pieces[k] | k in 1..K]);

solve minimize obj;

output
    [ "Cost = ",  show( obj ), "\n" ] ++
    [ "Pieces = \n\t" ] ++ [show(pieces)] ++ [ "\n" ] ++
    [ "Items = \n\t" ] ++
    [ show(items[k, i]) ++ if k = K then "\n\t" else " " endif |
    i in 1..N, k in 1..K ] ++ [ "\n" ];

% data:
N = 3; L = 10;
K = sum(demands);
lengths = [7, 5, 3];
demands = [2, 2, 4];
```

Fig. 6: Sample MINIZINC code.

```
array [1..3] of int: demands = [2, 2, 4];
array [1..3] of int: lengths = [7, 5, 3];
array [1..24] of var int: items  :: output_array([1..8, 1..3]);
array [1..8] of var 0..1: pieces :: output_array([1..8]);
var 0..8: obj :: output_var;
constraint int_le(0, items[1]);
  ...
constraint int_le(0, items[24]);
constraint int_lin_eq([-1, 1, 1, 1, 1, 1, 1, 1, 1],
    [obj, pieces[1], pieces[2], pieces[3], pieces[4],
    pieces[5], pieces[6], pieces[7], pieces[8]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[1], items[2], items[3], pieces[1]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[4], items[5], items[6], pieces[2]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[7], items[8], items[9], pieces[3]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[10], items[11], items[12], pieces[4]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[13], items[14], items[15], pieces[5]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[16], items[17], items[18], pieces[6]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[19], items[20], items[21], pieces[7]], 0);
constraint int_lin_le([7, 5, 3, -10],
    [items[22], items[23], items[24], pieces[8]], 0);
constraint int_lin_le([-1, -1, -1, -1, -1, -1, -1, -1],
    [items[1], items[4], items[7], items[10], items[13],
    items[16], items[19], items[22]], -2);
constraint int_lin_le([-1, -1, -1, -1, -1, -1, -1, -1],
    [items[2], items[5], items[8], items[11], items[14],
    items[17], items[20], items[23]], -2);
constraint int_lin_le([-1, -1, -1, -1, -1, -1, -1, -1],
    [items[3], items[6], items[9], items[12], items[15],
    items[18], items[21], items[24]], -4);
solve minimize obj;
```

Fig. 7: Sample FLATZINC code.

We compile this code-sample into FLATZINC using the `mzn2fzn` tool distributed with MINIZINC:

```
$ mzn2fzn cutstock.mzn
```

and obtain its equivalent problem instance in the FLATZINC format, shown in Figure 7. We run OPTIMATHSAT over this formula using the following command:

```
$ optimathsat -input=fzn < cutstock.fzn
```

and get the optimal solution value of 4, as witnessed by the sample output in Figure 8.

3.3.3 API

OPTIMATHSAT *API* consists of MATHSAT5's *C API* extended with additional commands for accessing the novel optimization functionalities. In addition to its native *C API*, OP-

```
% minimize obj => sat
% optimum: 4
% interval:     [ 0 , 8 ]
% Search steps: 2 (sat: 2)
%  - binary: 0 (sat: 0)
%  - linear: 2 (sat: 2)
% Restarts: 1 [session: 1]
% Decisions: 2 (0 random) [session: 2 (0 random)]
% Propagations: 3 (0 theory) [session: 72 (0 theory)]
% Watched clauses visited: 0 (0 binary) [session: 21 (19 binary)]
% Conflicts: 1 (1 theory) [session: 1 (1 theory)]
% Error:
%  - absolute: 0
%  - relative: 0
% Total time: 0.016 s
%  - first solution: 0.000 s
%  - optimization: 0.000 s
%  - certification: 0.016 s
% Memory used: 10.375 MB
obj = 4;
items = array2d(1..8, 1..3, [1, 0, 0, 0, 2, 0, 1, 0, 1, 0,
                0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0]);
pieces = array1d(1..8, [1, 1, 1, 0, 1, 0, 0, 0]);
----------
=========
```

Fig. 8: Sample FLATZINC output.

TIMATHSAT is distributed with the same scripts of MATHSAT5 for building *python API* bindings and library using *swig*. A detailed documentation of the *C API*, beyond the scope of this paper, is available on OPTIMATHSAT website [7].

Example. Figure 9 shows a *C API* code sample which encodes the first two goals of the problem described in §3.3.1. (In this example, we chose to omit some parts making use of functionalities that are shared with the underlying SMT solver MATHSAT5, and focused instead on the new optimization features introduced with OPTIMATHSAT.)

The source code is divided up in 6 sections. In the first section, an instance of the OPTIMATHSAT solver is created and conveniently configured to optimize multiple objectives in a lexicographic fashion, as well as to generate a model for the optimal solution. The formulation of the problem through the *API* is done in section 2, however we omitted the bulk of this code because it is completely equivalent to what one would write for a regular SMT problem in MATHSAT5. In section 3, two objective sums are created: `production_cost` and `used_machines`. While the first goal is encoded as a regular $\mathcal{LRA}$ term, the second Pseudo-Boolean objective is constructed using the `assert-soft` statement. This encoding allows OPTIMATHSAT to use its advanced techniques for MAXSMT/OMT with PB goals. The objectives are then pushed on the formula stack in section 4. Here, it should be noted that in order to refer to the created Pseudo-Boolean goal we retrieve the associated term by calling `msat_from_string` with the same `id` used at declaration time. At last, a single call to `msat_solve()` fires the optimization search in section 5 of the code. After the search status code is checked for correctness, the optimum model of the lexicographic optimization –which corresponds to that of the top-most objective on the formula stack– is

```
#include <assert.h>
...

int main(int argc, char *argv[])
{
    // 1. Create Environment

    msat_config cfg = msat_create_config();
    msat_set_option(cfg, "opt.priority", "lex");
    msat_set_option(cfg, "model_generation", "true");
    ...

    msat_env env = msat_create_env(cfg);

    // 2. add formula

    msat_term formula;

    ...

    msat_assert_formula(env, formula);

    // 3. encode cost functions

    msat_term production_cost = ... ;
    ...

    msat_assert_soft_formula(env, not_m0, one, "used_machines");
    ...

    // 4. add objectives

    msat_term used_machines = msat_from_string(env, "used_machines");

    msat_objective obj[2];
    obj[0] = msat_push_minimize(env, production_cost, NULL, NULL);
    obj[1] = msat_push_minimize(env, used_machines, NULL, NULL);

    // 5. optimize

    msat_result res = msat_solve(env);
    assert(res == MSAT_SAT);

    // 6. dump optimum model

    msat_set_model(env, obj[1]);

    ...

    msat_destroy_env(env);
    msat_destroy_config(cfg);
}
```

Fig. 9: Sample *C API* code.

loaded in the environment so as to allow the user to inspect its content. As it is usual in MATHSAT5, the allocated resources are then cleared before termination.

3.4 OPTIMATHSAT Functionalities and Options

In this section, we describe how the OMT techniques described in §2 can be accessed in OPTIMATHSAT, focusing in particular on its configurable options. Similarly to MATHSAT5, on which OPTIMATHSAT is based, there are several ways in which an option can be set:

- with a command line argument, e.g. `-option=VALUE`
- stored in a newline-separated configuration file, e.g. `option=VALUE`
- through the SMT-LIBv2 format, e.g.
  `(set-option :config option=VALUE)`
- through the *C API*, e.g.
  `msat_set_option(cfg, "option", "VALUE")`

Multi-Objective Combination. In the presence of multiple objectives within the same input formula, it is possible to instruct OPTIMATHSAT wrt. the preferential multi-objective combination to be used, among those illustrated in §2.5.

```
-opt.priority=STR                                    [default: box]
```

Sets the multi-objective combination to be used. Possible values are `box`, for the boxed multi-independent objective combination, and `lex` for lexicographic optimization.

Search Strategies. As described in §2.2.1, the search can advance towards the optimum goal according to several strategies. In OPTIMATHSAT, the *search strategy* can be selected using the option

```
-opt.strategy=STR                                    [default: lin]
```

Possible values for this option are: `lin` for *linear-search*, `bin` for *binary-search* and `ada` for *adaptive-search*. When minimizing, in order to use *binary-* and *adaptive-search* strategies, a *local lower bound* must be imposed on the objective function. Dual requirements hold for maximization. The behavior of the *binary-* and *adaptive-search* can be additionally configured with the following options:

```
-opt.bin.pivot_position=FLOAT                        [default: 0.5]
```

This option allows for adjusting the desired aggressiveness of *pivoting* cuts used in *binary-* and *adaptive-search* (see §2.2.1). Valid parametric values are contained in the $]0,1]$ interval, and express the relative position of the cut in the interval of values that goes from the *local lower bound* to the *local upper bound* of a given objective function. A value of 0.5 means that the *pivoting cut* bisects the search space in two exactly equal halves.

```
-opt.bin.first_step_linear=BOOL                      [default: true]
```

This option forces the first search step to be *linear*. In minimization, it allows for quickly finding an initial *local upper bound* estimate for a given objective, which can be smaller than the user-provided value (if any), yielding a better placement of the initial *pivoting* value.

```
-opt.bin.max_consecutive=INT                         [default: 1]
```

This option configures the maximum number of consecutive *pivoting* steps before a *linear-search* step is performed. This option is used to avoid "Zenoness" of the search when dealing with OMT ($\mathcal{LRA}$) objectives (see §2.2).

Search Learning. An important property of some OMT solvers is *incrementality* (see §2.6), that is the ability of exploiting learned information across multiple optimization searches to improve the search performance.To this extent, OPTIMATHSAT provides one useful option whose aim is to increase the chance of fruitfully re-using learned information when the tool is used incrementally:

```
-opt.learn_trivial_implications=BOOL                    [default: true]
```

When this option is true, it enables the learning of $\mathcal{LIRA}$-valid clauses in the form $(obj_i < u_i) \rightarrow (obj_i < u_i')$, where u_i is the most recent satisfiable value of obj_i found during minimization and u_i' is the previous value of u_i. As soon as the literal corresponding to $(obj_i < u_i)$ is assigned to true, this clause allows for "activating" all previously-learned clauses in the form $\neg(obj_i < u_i') \vee C$. See [73].

Search Termination. As described in §2.2, normally OPTIMATHSAT performs a complete optimization search, which can roughly divided in two phases: in the first phase, the solver enumerates a number of satisfiable solutions which get closer and closer to the optimal solution, whereas in the second phase the optimality of the latest solution found is certified. In this regard, the common experience is that the certification step takes in general as much search time as the initial enumeration phase.

In some applications, due to the particular time-demanding characteristic of the optimization search, the need for a certified optimal solution might be relaxed in favor of "good enough" approximations of the optimal solution [76]. For this reason, OPTIMATHSAT provides a number of useful options that allow for early terminating the optimization search.

```
-opt.no_optimization=BOOL                              [default: false]
```

When this option is enabled, the optimization search stops at the first (possibly not optimal) satisfiable solution.

```
-opt.soft_timeout=BOOL                                 [default: false]
```

If this flag is activated, the search timeout is ignored when it occurs before each objective has at least one (possibly sub-optimal) solution. If this is the case, the search is then automatically terminated as soon as the condition becomes true. In essence, this option is meant to postpone an otherwise hard timeout which could terminate the optimization search even before the solver is able to determine the satisfiability of the input formula, a situation which may occur since the timer starts ticking from the moment the tool is launched. This feature can be used, in combination with the `timeout=FLOAT` option inherited from MATHSAT5, to handle those OMT applications in which an approximated optimal solution is acceptable but the absence of any solution is not.

```
-opt.abort_interval=FLOAT                                [default: 0.0]
```

This option makes the optimization search stop as soon as the search interval size for a given objective is below the configured threshold absolute value, thus yielding an approximated value of the actual optimal solution.

```
-opt.abort_tolerance=FLOAT                               [default: 0.0]
```

Similarly to the previous one, this option makes the search stop as soon as the ratio among the current search interval size and the initial search interval size is smaller than the given threshold value.

General $\mathcal{T}$-Optimization. Normally, for each satisfiable Boolean assignment that is enumerated during the optimization search, OPTIMATHSAT invokes a $\mathcal{T}$-Solver optimizer that yields the corresponding optimal value of a given objective function, as described in §2.2. This behavior can be globally adjusted using the following option:

```
-opt.theory.no_optimization=BOOL                              [default: false]
```

If this option is enabled, the $\mathcal{T}$-Solver optimizer is not called and the objective function is assigned an arbitrary value that is consistent with the current Boolean assignment. Importantly, disabling the $\mathcal{T}$-Solver optimization might cause non-termination on Theories with infinitely-many enumerable satisfiable solutions over a closed interval (e.g. $\mathcal{LRA}$).

$\mathcal{LIRA}$-Optimization. The optimization procedures for OMT($\mathcal{LRA}$) and OMT($\mathcal{LIA}$) described in §2.2 and §2.3 respectively, can be tuned with the following options:

```
-opt.theory.la.ignore_non_improving=BOOL                       [default: true]
```

When this flag is enabled, the $\mathcal{T}$-Solver optimizer is not invoked for a given objective function if the current Boolean assignment does not allow for an improving solution. In order to test for this condition, OPTIMATHSAT performs a quick satisfiable check using the most recent optimal solution approximation for the objective function. This option can be useful in *boxed multi-objective* optimization to avoid wasting time within the (usually more expensive) $\mathcal{T}$-Solver optimizer when it is not profitable.

```
-opt.theory.la.lar_always_optimize=BOOL                       [default: false]
```

This option forces the $\mathcal{LRA}$ tableau to be $\mathcal{T}$-optimized after each satisfiability check in single-objective mode, including early-pruning calls, so as to always leave it in an optimal configuration wrt. the objective function. It has no effects in multi-objective mode.

```
-opt.theory.la.laz_mode=STR                                    [default: part]
```

This option sets the aggressiveness of Branch&Bound optimization in the $\mathcal{LIA}$ theory solver, as described in §2.3. The value: `part` corresponds to a quick, incomplete search – a.k.a. *truncated B&B* in §2.3– whereas `full` stands for the complete, but possibly expensive, Branch&Bound search.

OPTIMATHSAT offers additional fine-tuning capabilities for controlling the value of $\mathcal{LIRA}$ variables in the optimum model associated with $\mathcal{LIRA}$-objectives. These two options are used exclusively at model-construction time.

```
-opt.theory.la.infinite_pow=INT                                   [default: 9]
```

This option sets the finite representation of infinite $\mathcal{LIRA}$ values to be equal to 10^N, where N is the input value.

```
-opt.theory.la.delta_pow=INT                                      [default: 6]
```

This other option, sets the value of delta to be equal to 10^{-N}, where N is the input value. Given an infinite-precision $\mathcal{LRA}$ variable $\langle real, eps \rangle$, its model value is set to be $real + eps \cdot delta$.

OMT(PB) / MAXSMT. At the present time, OPTIMATHSAT supports three different approaches for dealing with MAXSMT/OMT + Pseudo Boolean objectives (§2.4). It is possible to select the desired engine using the following option:

```
-opt.maxsat_engine=STR                                      [default: omt]
```

Valid values for this option are `omt` for standard OMT techniques, `maxres` for the Maximum Resolution engine and `ext` for an external MaxSAT solver engine using the *Lemma Lifting* approach described in [35]. Importantly, OPTIMATHSAT takes advantage of these specialized routines only when the MAXSMT or Pseudo Boolean goals are defined in the input formula with the aid of *soft clauses*. When the `omt` approach is selected, as described in §2.4.2 and §2.4.3, the following options can be used to fine-tune the encoding of MAXSMT and OMT + PB objectives into OMT:

```
-opt.asoft.encoding=STR                                     [default: car]
```

This option selects the preferred encoding of Pseudo-Boolean / MAXSMT cardinality constraints induced by the *soft-clauses* appearing in the input formula. Valid options are `seq` and `car` for the sorting network encoding respectively based on the *sequential counter* and *cardinality network* circuits described in §2.4.3, and `la` for the plain linear arithmetic encoding.

```
-opt.asoft.circuit_limit=INT                                 [default: 20]
```

If greater than zero, this option imposes an upper-bound on the maximum number of inputs that each generated sorting network circuit can have. If there exist MAXSMT or Pseudo-Boolean terms using more than the given threshold of variables, these are encoded with a number of smaller sorting network circuits combined by means with linear arithmetic techniques. This option serves the purpose of limiting the amount of memory used by Sorting Networks, and has been shown to significantly improve the performance when the *sequential counter* encoding for sorting networks is being used [74].

```
-opt.asoft.no_bidirection=BOOL                            [default: false]
```

If this flag is enabled, the encoding of Pseudo-Boolean / MAXSMT objectives is not bidirectional, and it is guaranteed to yield a correct value only if the objective is minimized. This feature reduces the number of clauses used, and it can positively affect the search performance in some situations.

```
-opt.asoft.prefer_pbterms=BOOL                             [default: true]
```

When this option is activated, the Boolean label associated with each soft-clause is added to the list of variables preferred for branching within the CDCL/SAT engine, which typically improves the search performance. This option has no effect if `opt.asoft.reduce_vars` is also set.

```
-opt.asoft.reduce_vars=BOOL                               [default: false]
```

If this flag is true, then no Boolean label is associated with each soft-clauses, thus decreasing the number of variables used by the encoding.

```
-opt.asoft.detect_bounds=BOOL                             [default: false]
```

If this option is true, then OPTIMATHSAT automatically computes the *local lower* and *upper bounds* for MAXSMT and Pseudo-Boolean objectives defined in terms of *soft-clauses*. This feature relieves the end-user from manually computing and inserting the correct values in order to use *binary-* and *adaptive-search* strategies.

Miscellaneous Functionalities. We conclude with a short list of additional options available in OPTIMATHSAT that have no clear grouping reference.

```
-opt.verbose=BOOL                                    [default: false]
```

If this flag is enabled, OPTIMATHSAT shows progress information along the optimization search, printing in particular each update to the *local lower* and *upper bounds* of any objective function.

```
-opt.fzn.use_asoft_encoding=BOOL                      [default: true]
```

If this option is true, Pseudo-Boolean terms induced by FLATZINC global constraints are encoded using *soft clauses* rather than with linear programming.

```
-opt.fzn.use_ite_encoding=BOOL                        [default: true]
```

When this option is enabled, 0-1 variables appearing in the original formula are replaced with the corresponding `ITE` terms whenever possible. This is a work-around which aims to avoid non-linear constraints, which might appear in the formula as a result of the 0-1 encoding, since they are not supported by OPTIMATHSAT. For best results, all `bool2int` constraints in the input problem should be sorted to appear at the beginning of the constraints section.

```
-opt.debug.expand_soft=BOOL                          [default: false]
```

When this flag is enabled, the *API Tracing* functionality expands *soft-clauses* in the corresponding SMT-LIBV2 generated formula, which can then be fed as input to other OMT solvers which lack support for the `assert-soft` statement or handle it differently.

4 Experimental Evaluations

In order to showcase OPTIMATHSAT functionalities and performance in different scenarios, we present some previously published empirical evaluations. The first evaluation is taken from [73], and depicts a performance comparison among the single-objective, incremental and boxed multi-objective optimization modes. The second and third experiments are taken from [74], and provide two samples of OPTIMATHSAT applications unsuitable for MAXSAT-based optimization approaches, for which OMT-based techniques must be used. All the benchmarks, the experimental data and the scripts necessary to reproduce the three experiments are made available at [3].

4.1 Single-Objective, Incremental and Multi-Objective OMT

In this experiment, we report the performance of single-objective, incremental and multi-objective optimization with OMT-based techniques over the set of benchmarks used in [51]. Each of these benchmarks, generated from a set of C programs used at the SW Verification Competition of 2013, is a multi-objective problem which computes an over-approximation of the feasible domain of a number of variables.

Given a multi-objective problem $\langle \varphi, obj_1, ..., obj_N \rangle$, we consider three different OMT-based approaches for solving it:

Tool:	#inst.	#solved	#timeout	time
SYMBA(100)	1103	1091	12	10917
SYMBA(40)+OPT-Z3	1103	1103	0	1128
Z3 -multiobjective	1103	1090	13	1761
Z3 -incremental	1103	1100	3	8683
Z3 -singleobjective	1103	1101	2	10002
optimathsat-multiobjective	1103	1103	0	**901**
optimathsat-incremental	1103	1103	0	3477
optimathsat-singleobjective	1103	1103	0	16161

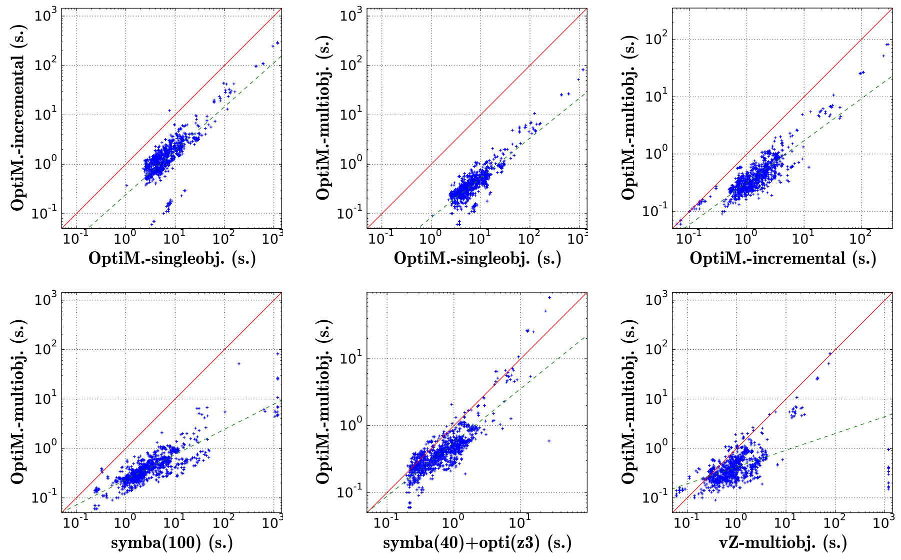


Fig. 10: [Table:] comparison among OPTIMATHSAT, SYMBA and Z3 on SW verification problems in [51]. [Top Plots:] pairwise comparisons between different versions of OPTIMATHSAT. [Bottom Plots:] pairwise comparisons between OPTIMATHSAT-MULTI-OBJECTIVE, the two versions of SYMBA and Z3 -MULTIOBJECTIVE.

- SINGLE-OBJECTIVE: the problem is split in N independent problems $\langle \varphi, obj_i \rangle$ which are sequentially solved, and the cumulative time is taken;
- INCREMENTAL: as above, this time leveraging on the incremental interface to pop the definition of the previous *obj* before pushing a new one;
- MULTI-OBJECTIVE: the original problem is fed to the OMT solver configured to run in boxed multi-objective mode, as explained in 2.5.

We tested each configuration with both Z3 and OPTIMATHSAT, and compared their performance with that of the two best-performing versions of SYMBA presented in [51], that is SYMBA(100) and SYMBA(40)+OPT-Z3.

Figure 10 shows the global performance data of all procedures under test (top) and the corresponding pairwise comparisons (bottom).

In the top-left plot in Figure 10, we observe that moving from non-incremental to incremental OMT results in an uniformly relevant speedup. This improvement can be explained by the chance of reusing learned clauses across multiple incremental searches, which can save considerable efforts as explained in §2.6.

tool, configuration & encoding	**inst.**	**term.**	**timeout**	**time (s.)**
OPTIMATHSAT (OMT-based)	2399	2340	59	20841
OPTIMATHSAT (OMT-based + seq. counter)	2399	2394	5	9511
OPTIMATHSAT (OMT-based + card. network)	2399	**2395**	4	8275
Z3 (OMT-based)	2399	2390	9	8076

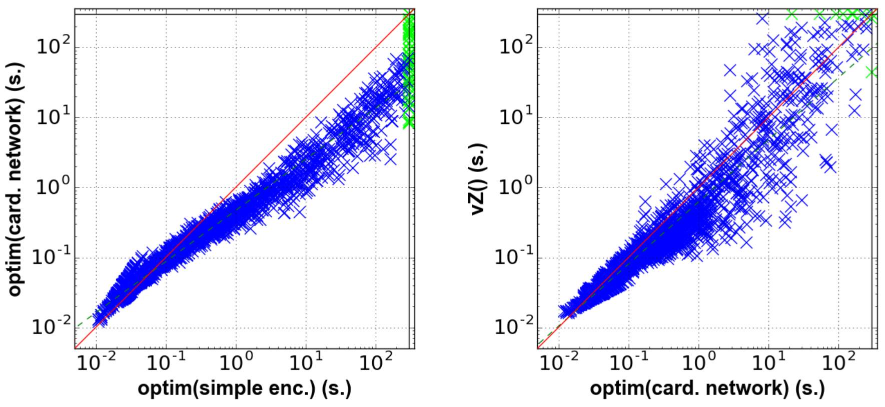


Fig. 11: [Table:] results of various solvers with OMT-based configurations on CGM-encoding problems of [60,59] with max-min objective functions. [Left scatterplot:] OPTIMATHSAT + card. network vs. plain OPTIMATHSAT. [Right scatterplot:] Z3 vs. OPTIMATHSAT + card. network.

The performance speedup is even more drastic –about one order of magnitude– when boxed multi-objective configuration is considered, as shown in the top-center plot. We also notice, in the top-right plot, that this performance improvement is significantly better than that obtained with incremental OMT.

In the bottom row of figure 10, we observe that compared with the other scrutinized OMT solvers, OPTIMATHSAT-MULTI-OBJECTIVE performs much better than the default configuration of SYMBA (right), and significantly better than both SYMBA(40)+OPT-Z3 (center) and Z3 -MULTI-OBJECTIVE (left).

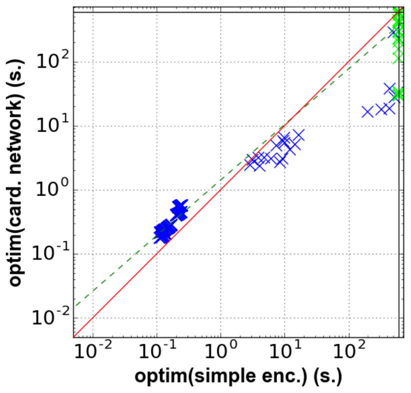
4.2 CGMs with max-min goal + PB Objectives

Here we consider a variant of the problem of computing a lexicographically-optimum realization of a constrained goal model [60,59]. In this comparison, the three PB/MAXSMT objectives appearing in the original set of formulas are normalized so that their respective range is equal to $[0, 1]$ and combined in a single max-min goal rather than lexicographically optimized.

To the best of our knowledge, none of the most-efficient MAXSAT-based techniques described in §2.4.1 is applicable on this problem. Therefore, we settled on using OMT-based techniques presented in §2.4.2 and §2.4.3 only.

Looking at the table and the scatter-plot on the left in figure 11, we observe a significant performance improvement when the baseline OMT-based technique of OPTIMATHSAT is enhanced with cardinality networks. We notice also that this approach, when implemented

tool, configuration & encoding	inst.	term.	timeout	time (s.)
OPTIMATHSAT (OMT-based)	500	421	79	2607
OPTIMATHSAT (OMT-based + seq. counter)	500	441	59	6381
OPTIMATHSAT (OMT-based + card. network)	500	**442**	58	6189
Z3 (OMT-based)	500	406	94	2120



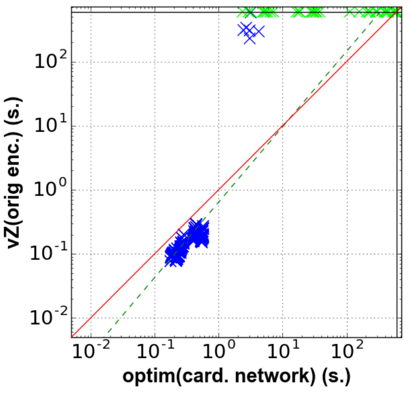


Fig. 12: [Table:] results of various solvers with OMT-based configurations on LMT-encoding problems of [76] with complex objective functions. [Left scatterplot:] OPTIMATHSAT + card. network vs. plain OPTIMATHSAT. [Right scatterplot:] Z3 vs. OPTIMATHSAT + card. network.

in OPTIMATHSAT, yields equivalent or slightly better performance than Z3 by looking at both the table and the scatter-plot on the right.

4.3 LMTs with mixed complex objective functions.

In this experiment we consider a benchmark set of 500 formulas generated with PYLMT [8], a tool for Structured Learning Modulo Theories [76] –doing inference in the context of machine learning in hybrid domains– which uses OPTIMATHSAT as a black-box engine.

The optimization goal *obj* used in these formulas is a complex arithmetic combination of several PB terms and has the following general form:

$$\begin{aligned} obj &\stackrel{\text{def}}{=} \textstyle\sum_j w_j \cdot B_j + cover - \sum_k v_k \cdot C_k - |K - cover|, \\ cover &\stackrel{\text{def}}{=} \textstyle\sum_i z_i A_i, \end{aligned} \tag{11}$$

where A_i, B_j, C_k are Boolean atoms and w_i, v_j, z_k, K are rational constants.

The results are presented in Figure 12. Similarly to the other experiments, by looking at the table and the scatter-plot on the left we observe that the performance of the OMT-based technique in OPTIMATHSAT is improved by cardinality networks. In this case, however, the performance gain is not dramatic. We explain this by observing that on this set of benchmarks the values of weights w_i, v_i, z_i can be very heterogeneous, and this limits the viability of the sorting-networks which are used only when several weights share the same value (§2.4.3). We notice also that, even without the help of sorting networks, the OMT-based technique of OPTIMATHSAT performs significantly better than that of Z3 .

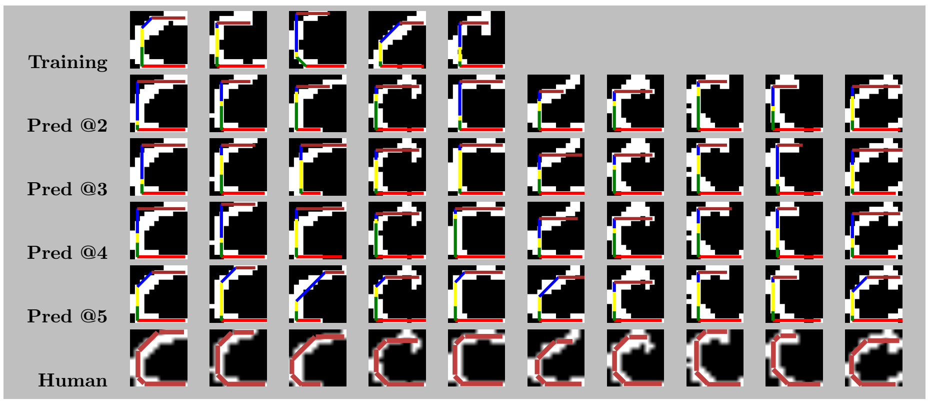


Fig. 13: An example from [76] of SLMT application to the hybrid domain of automatic character drawing. Results for the "C" 12×12-pixel character drawing task.

5 Applications

We briefly mention a few examples of recent applications –some of which are very innovative in their respective domains– that have been technologically enabled by OMT and by the usage of OPTIMATHSAT as backend automated-reasoning engine. Some such applications were of primary importance to solicit and drive the development of some of the ideas and techniques in §2 and §3.

5.1 Learning Modulo Theories

In Machine Learning applications, performing inference and learning in hybrid domains –characterized by both continuous and Boolean/discrete variables– is a particularly daunting task. *Structured Learning Modulo Theories (SLMT)* [76] addresses the problem by combining (Structured-Output) Support Vector Machines (SVNs) with OMT, so that the latter plays the role of inference and separation oracle for the former. The tool LMT implementing the SLMT method [5] uses OPTIMATHSAT as backend OMT engine. An example application to automatic character drawing via LMT with OPTIMATHSAT taken from [76] is reported in Figure 13. Interestingly, as reported in §4.3, SLMT required defining objectives which were complex arithmetic combinations of several PB terms (like, e.g., (11)). We refer the reader to [76] for details.

5.2 Constrained Goal Models

Goal Models (GM) are used in Requirements Engineering to represent software requirements, objectives, and design qualities [77]. *Constrained Goal Models (CGM)* [59,60] are a novel, formal version of GM, representing AND/OR goal decomposition graphs which are enriched with constraints so that to handle preferences, numerical attributes and resources (e.g., scores, financial cost, workforce, etc.).

OPTIMATHSAT is used as a backend reasoning engine of CGM-TOOL [59,60,1], a tool for building and reasoning on *CGM*s, allowing for automatically verifying the realizability

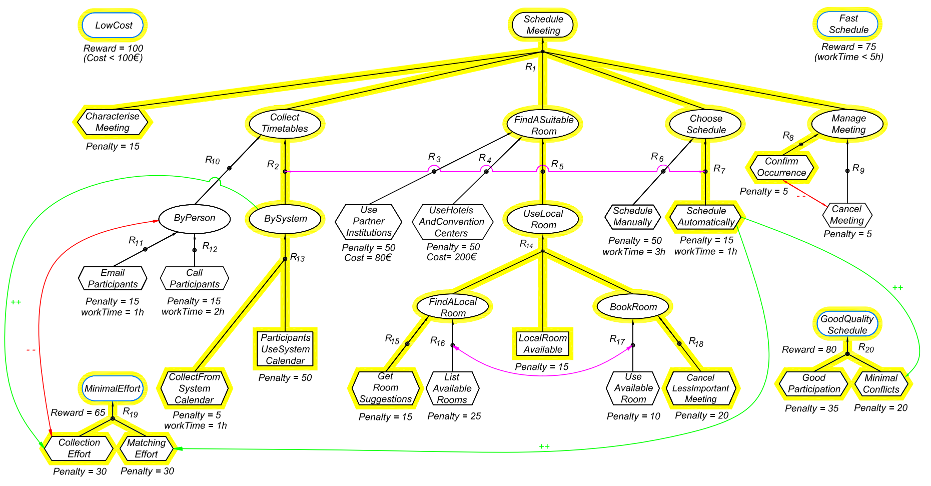


Fig. 14: An example from [59] of a simple CGM and its realization which minimizes lexicographically the numerical objective $\langle \text{PENALTY} - \text{REWARD}, \text{WORKTIME}, \text{COST} \rangle$. The realization is highlighted in yellow.

of a CGM and for finding optimal realizations according to some specified criterion. A simple example of CGM for a meeting scheduling example taken from [59,60] is presented in Figure 14. We refer the reader to [59,60] for details.

5.3 WCET

In the context of real-time systems, it is often necessary to compute an upper bound on the worst-case execution time (WCET) of programs with hard time constraints. In [45], Henry et al. presented a novel approach for computing upper bounds for loop-free C programs based on Optimization Modulo Theories and Z3. By taking into account the semantics of a program, which allows for pruning infeasible paths from the search space, and conjoining appropriate cuts to the formula, to bring it to a tractable size, they were able to obtain tighter estimates of the WCET, in some cases with an impressive improvement.

Recently, there has been an attempt to expand on the research of [45] with a new project, which is still work in progress, which uses OPTIMATHSAT at its core [10]. In this work, OPTIMATHSAT has been extended with the capability of learning a new set of $\mathcal{T}$-lemmas, starting from subsequent $\mathcal{T}$-conflicts, which relax the need for conjoining cuts to the initial formula to bring it to a tractable size. Unpublished preliminary experimental results show the usefulness of this approach.

5.4 Quantum Annealing

Quantum annealers (QA) are specialized quantum computers that minimize objective functions over binary variables by physically exploiting quantum effects [46,18]. Current QA platforms like D-Wave 2000Q [2] allow for the optimization of quadratic objectives defined over binary variables (qubits), solving quadratic unconstrained binary optimization (QUBO) problems. In the last decade QA systems have scaled with Moore-like growth, s.t. current architectures provide 2048 sparsely-connected qubits, and continued exponential growth is anticipated.

Bian et al. [19] have investigated the problem of effectively encoding SAT and MaxSAT problems into QUBOs to be fed to and solved by state-of-the-art D-Wave 2000Q QAs. To this extent, OPTIMATHSAT is used off-line to create libraries of QUBO encodings of useful Boolean functions by automatically computing an optimum choice of (a) the QUBO input parameter values and of (b) variable-to-qubit placement, so that to maximize a parameter (gap) stating the robustness of the system wrt. noise. The two problems (a) and (a)+(b) are addressed by solving an OMT($\mathcal{LRA}$) and an OMT($\mathcal{LIRA} \cup \mathcal{UF}$) problem respectively. We refer the reader to [19] for details.

6 Related Work

MAXSMT *and OMT(PB)* The idea of optimization in SMT was first introduced by Nieuwenhuis & Oliveras [61], who presented an abstract logical framework of "SMT with progressively stronger theories" and presented implementations for MaxSMT based on this framework. Cimatti et al. [32] introduced the notion of "Theory of Costs" $\mathcal{C}$ to handle PB cost functions and constraints by an ad-hoc and independent "$\mathcal{C}$-solver" in the standard lazy SMT schema, and implemented a variant of MathSAT tool able to handle SMT with PB constraints and to minimize PB cost functions. Cimatti et al. [35] presented a "modular" approach for MaxSMT, combining a lazy SMT solver with a MaxSAT solver, which can be used as blackboxes. We recall that SMT with PB functions and MaxSMT can be encoded into each other, and that both are strictly less general than the OMT($\mathcal{LRA} \cup \mathcal{T}$) problems (see [70,71]).

OMT($\mathcal{LIRA} \cup \mathcal{T}$). Sebastiani and Tomasi [70,71] introduced a wider notion of optimization in SMT, namely *Optimization Modulo Theories (OMT) with $\mathcal{LRA}$ cost functions*, OMT($\mathcal{LRA} \cup \mathcal{T}$), which allows for finding models minimizing some $\mathcal{LRA}$ cost term –$\mathcal{T}$ being some (possibly empty) stably-infinite theory s.t. $\mathcal{T}$ and $\mathcal{LRA}$ are signature-disjoint– and presented novel OMT($\mathcal{LRA} \cup \mathcal{T}$) tools which combine standard SMT with LP minimization techniques. ($\mathcal{T}$ can also be a combination of Theories $\bigcup_i \mathcal{T}_i$.) Importantly, both MAXSMT and SMT with PB objectives can be encoded into OMT($\mathcal{LRA} \cup \mathcal{T}$), whereas the contrary is not possible [70,71]. Eventually, OMT($\mathcal{LRA} \cup \mathcal{T}$) has been extended so that to handle costs on the integers, incremental OMT, multi-objective, and lexicographic OMT and Pareto-optimality [51,48,22,73,23,72].

OMT Tools. To the best of our knowledge only four OMT solvers are currently implemented: BCLT [48], Z3 (a.k.a. Z3OPT) [22,23], OPTIMATHSAT [73,72], and SYMBA [51]. Remarkably, BCLT, Z3 and OPTIMATHSAT currently implement also specialized procedures for MaxSMT, leveraging to SMT level state-of-the-art MaxSAT procedures; in addition, Z3 features a Pseudo-Boolean *$\mathcal{T}$-solver* which can generate sorting circuits on demand for Pseudo-Boolean inequalities featuring sums with small coefficients when a Pseudo-Boolean inequality is used some times for unit propagation/conflicts [23,21].

MILP and LGDP. Mixed Integer Linear Programming (MILP) is an extension of *Linear Programming (LP)* involving both discrete and continuous variables. A large variety of techniques and tools for MILP are available, mostly based on efficient combinations of

LP, *branch-and-bound* search mechanism and *cutting-plane* methods (see e.g. [52]). SAT techniques have also been incorporated into these procedures for MILP (see [13]). *Linear Disjunctive Programming (LDP)* problems are LP problems where linear constraints are connected by conjunctions and disjunctions [15]. Closest to our domain, *Linear Generalized Disjunctive Programming (LGDP)*, is a generalization of LDP which has been proposed in [65] as an alternative model to the MILP problem. Unlike MILP, which is based entirely on algebraic equations and inequalities, the LGDP model allows for combining algebraic and logical equations with Boolean propositions through Boolean operations, providing a much more natural representation of discrete decisions. Current approaches successfully address LGDP by reformulating and solving it as a MILP problem [65, 78, 67, 68]; these reformulations focus on efficiently encoding disjunctions and logic propositions into MILP, so as to be fed to an efficient MILP solver like CPLEX. An extensive empirical comparison of OMT($\mathcal{LRA}$) vs LGDP was presented in [71].

To this extent, we believe that OMT solvers can play as an interesting alternative to MILP ones when the addressed problems (i) have a strong Boolean component (see e.g. the empirical results in [71]), or (ii) require dealing with other theories, in particular non-numerical ones, or (iii) require incremental calls –or in any combination of the above cases.

FLATZINC *and* MINIZINC. The study in [26, 24, 27] represents one of the earliest attempts to investigate the effectiveness of SMT on problems typically solved with CP and MILP techniques. In those papers, Bofill et Al. presented SIMPLY, a compiler for translating CSP problems into the SMT-LIBV2 format and showed that SMT tools can scale up well on benchmarks requiring substantial Boolean reasoning. In a following study [25], Bofill et Al. presented a novel translation framework achieving the same goal, but this time targeting MINIZINC, a widely adopted high-level declarative language for modeling CSP problems. Remarkably, MINIZINC supports an extensive library of global constraints, several data types (floats, bools, integers, sets and multidimensional arrays), user-defined predicates, if-then-else and let expressions. MINIZINC is also the language used at the MINIZINC Challenge [56], an yearly competition among CSP tools dealing with a vast library of benchmarks about planning, scheduling, logistics and more. Differently than with SIMPLY, the approach adopted in [25] is to combine an existing MZN2FZN tool, which compiles MINIZINC problems into an easily-parsable format called FLATZINC, with a novel FZN2SMT compiler mapping FLATZINC models into SMT-LIBV2 formulas. In order to deal with optimization goals, the latter tool was complemented with an optimization procedure built on top of an external SMT solver used as a black-box. Remarkably, FZN2SMT–paired with the SMT solver YICES [40]– was able to score two silver medals for two years consecutively in the MINIZINC Challenge (2011 and 2012).

Other Related Approaches. In the literature, there exist several research attempts to integrate the very successful achievements obtained in the last decades by SAT communities within Finite Domain Propagator (FDP) and Integer Linear Programming (ILP) solvers. For instance, both [63, 64] and [42] presented an FDP solver with a SAT engine, used for lazy clause generation, fast back-jumping and recording of clauses learned along the search. In [53], instead, Manolios et Al. presented Integer Linear Programming Modulo Theories (IMT), a sound and complete framework for combining ILP with a background solver for a theory T, and Inez, a novel IMT tool. Differently than SMT, which is centered around the SAT solver, the search in an IMT solver is guided by a Branch and Cut procedure that communicates with some $\mathcal{T}$*-solver* by means of interface difference logic inequalities. An experimental evaluation, conducted over a set of benchmarks derived from the problem of synthesizing architectural models for a Boeing 787 Dreamliner, showed that IMT can be competitive wrt. state-of-the-art SMT solvers.

7 Conclusions and Future Developments

In this paper, we presented OPTIMATHSAT, a state-of-the-art OMT solver which supports both single- and multi-objective optimization of $\mathcal{LRA}$, $\mathcal{LIA}$, $\mathcal{LIRA}$ or PB/MAXSMT objectives. In addition, we explored the concept of compositional definition of objectives, currently one of the distinguished features of OPTIMATHSAT, that we believe it should become a standard for all OMT solvers in the future. In our overview of OPTIMATHSAT, we focused on its high-level architecture and its tool-set of functionalities. To this extent, the presentation is complemented with both an introduction to OMT, tailored around OPTIMATHSAT specific set of techniques and heuristics, and a selected review of experimental results published in previous papers.

Optimization Modulo Theories is an exciting, albeit young, technology with large margins for improvement. We are planning to to investigate novel OMT techniques and hence to extend OPTIMATHSAT capabilities along several directions.

First, we plan to extend OPTIMATHSAT to handle objectives defined on other theories than linear arithmetic that are already supported by MATHSAT5: Bit-Vectors [57] and floating-point arithmetic [44,29,30]. Also, we plan to investigate –and to integrate into OPTIMATHSAT if successful– OMT versions of the novel SMT procedures for $\mathcal{NRA}$ [33] and for $\mathcal{NRA}$ plus transcendental functions [34] which have been implemented on top of MATHSAT5.

Another opportunity is represented by the introduction of a Pareto optimization scheme in OPTIMATHSAT, similarly to what is already available in Z3 [22,23], possibly adapting some well-known algorithm to our needs (e.g., [49,66,50]). This extension could be matched with a more general combination of OMT with the so-called all-SMT functionality [47], currently supported by MATHSAT5, which would allow for easily enumerating all the equivalently-optimal solutions of a given problem.

Finally, we plan to further expand our support for MINIZINC, making OPTIMATHSAT the bridge of two worlds that are now seemingly distinctly separated. In order to do so, we strive for improving the efficiency and expressiveness of OPTIMATHSAT so as to cover the largest possible subset of the FLATZINC language. This may require extending our OMT tool with dedicated procedures for dealing with global constraints and sets, and possibly investigating more efficient techniques or heuristics for dealing with heavily ILP-focused problems. This work should be validated with a serious extensive empirical comparison and possibly by making OPTIMATHSAT participate to the annual MINIZINC competition [56], if deemed eligible.

References

1. CGM-Tool. http://www.cgm-tool.eu.
2. D-wave 2x tecnology overview. https://www.dwavesys.com/sites/default/files/D-Wave%202X%20Tech%20Collateral_0915F.pdf.
3. Experimental data. http://disi.unitn.it/trentin/resources/jar2017.tar.gz.
4. FlatZinc 1.6. http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf.
5. LMT. http://disi.unitn.it/~teso/lmt/lmt.tgz.
6. MiniZinc 1.6. http://www.minizinc.org/downloads/doc-1.6/zinc-spec.pdf.
7. OptiMathSAT. http://optimathsat.disi.unitn.it.
8. PyLMT. http://www.bitbucket.org/stefanoteso/pylmt.
9. FLATZINC support in OPTIMATHSAT. http://optimathsat.disi.unitn.it/pages/fznreference.html.
10. WCET OMT. https://github.com/PatrickTrentin88/wcet_omt.
11. I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints. In *19th International Conference on Principles and Practice of Constraint Programming*, CP'13, 2013.
12. T. Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 07 2009.
13. T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: a new approach to integrate CP and MIP. In *Proc. CPAIOR'08*, LNCS, pages 6–20. Springer, 2008.
14. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. Satisfiability Modulo Theories: An Efficient Approach for the Resource-Constrained Project Scheduling Problem. In *SARA*, 2011.
15. E. Balas. Disjunctive programming: Properties of the convex hull of feasible points. *Discrete Applied Mathematics*, 89(1-3):3 – 44, 1998.
16. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The satisfiability modulo theories library (smt-lib). http://www.smtlib.org, 2010.
17. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. In Biere et al. [20], February 2009.
18. Z. Bian, F. Chudak, R. Israel, B. Lackey, W. G. Macready, and A. Roy. Discrete optimization using quantum annealing on sparse ising models. *Frontiers in Physics*, (56), 2014.
19. Z. Bian, F. Chudak, W. Macready, A. Roy, R. Sebastiani, and S. Varotti. Solving SAT and MaxSAT with a Quantum Annealer: Foundations and a Preliminary Report. In *Frontiers of Combining Systems*, volume 10483 of *LNCS*, pages 153–171. Springer, 2017.
20. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, February 2009.
21. N. Bjorner. personal communication, 02 2016.
22. N. Bjorner and A.-D. Phan. νZ - Maximal Satisfaction with Z3. In *Proc International Symposium on Symbolic Computation in Software Science*, Gammart, Tunisia, December 2014. EasyChair Proceedings in Computing (EPiC). http://www.easychair.org/publications/?page=862275542.
23. N. Bjorner, A.-D. Phan, and L. Fleckenstein. Z3 - An Optimizing SMT Solver. In *Proc. TACAS*, volume 9035 of *LNCS*. Springer, 2015.
24. M. Bofill, M. Palahı, J. Suy, and M. Villaret. Simply: a compiler from a csp modeling language to the smt-lib format. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation*, pages 30–44, 2009.
25. M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.
26. M. Bofill, M. Palahı, and M. Villaret. A system for csp solving through satisfiability modulo theories. *IX Jornadas sobre Programación y Lenguajes (PROLE09)*, pages 303–312, 2009.
27. M. Bofill, J. Suy, and M. Villaret. A system for solving constraint satisfaction problems with smt. *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 300–305, 2010.
28. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Theory Combination via Boolean Search. *Information and Computation*, 204(10):1493–1525, 2006.
29. M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Interpolation-Based Verification of Floating-Point Programs with Abstract CDCL. In *SAS*, pages 412–432, 2013.
30. M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014.
31. R. H. Byrd, A. J. Goldman, and M. Heller. Technical Note– Recognizing Unbounded Integer Programs. *Operations Research*, 35(1), 1987.

32. A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In *TACAS*, volume 6015 of *LNCS*, pages 99–113. Springer, 2010.
33. A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani. *Invariant Checking of NRA Transition Systems via Incremental Reduction to LRA with EUF*, pages 58–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.
34. A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani. Satisfiability Modulo Transcendental Functions via Incremental Linearization. In *Proc. Int. Conference on Automated Deduction, CADE-26*, LNCS. Springer, 2017.
35. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. A Modular Approach to MaxSAT Modulo Theories. In *International Conference on Theory and Applications of Satisfiability Testing, SAT*, volume 7962 of *LNCS*, July 2013.
36. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT 5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'13.*, volume 7795 of *LNCS*, pages 95–109. Springer, 2013.
37. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM Transaction on Computational Logics – TOCL*, 12(1), October 2010.
38. A. Cimatti, A. Griggio, and R. Sebastiani. Computing Small Unsatisfiable Cores in SAT Modulo Theories. *Journal of Artificial Intelligence Research, JAIR*, 40:701–728, April 2011.
39. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, 2006.
40. B. Dutertre and L. de Moura. System Description: Yices 1.0. In *Proc. on 2nd SMT competition, SMT-COMP'06*, 2006. Available at `yices.csl.sri.com/yices-smtcomp06.pdf`.
41. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.
42. T. Feydy and P. J. Stuckey. Lazy clause generation reengineered. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP'09, pages 352–366, Berlin, Heidelberg, 2009. Springer-Verlag.
43. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation - JSAT*, 8:1–27, 2012.
44. L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding Floating-Point Logic with Systematic Abstraction. In *Proc. of FMCAD*, 2012. To Appear.
45. J. Henry, M. Asavoae, D. Monniaux, and C. Maïza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. *SIGPLAN Not.*, 49(5):43–52, 06 2014.
46. M. W. Johnson, M. H. S. Amin, S. Gildert, T. Lanting, F. Hamze, N. Dickson, R. Harris, A. J. Berkley, J. Johansson, P. Bunyk, E. M. Chapple, C. Enderud, J. P. Hilton, K. Karimi, E. Ladizinsky, N. Ladizinsky, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, C. J. S. Truncik, S. Uchaikin, J. Wang, B. Wilson, and G. Rose. Quantum annealing with manufactured spins. *Nature*, 473(7346):194–198, 2011.
47. S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Proc. CAV*, LNCS 4144. Springer, 2006.
48. D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Minimal-Model-Guided Approaches to Solving Polynomial Constraints and Extensions. In *SAT*, 2014.
49. M. Laumanns, L. Thiele, and E. Zitzler. An adaptive scheme to generate the pareto front based on the epsilon-constraint method. In J. Branke, K. Deb, K. Miettinen, and R. E. Steuer, editors, *Practical Approaches to Multi-Objective Optimization*, number 04461 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
50. J. Legriel, C. Le Guernic, S. Cotton, and O. Maler. Approximating the pareto front of multi-criteria optimization problems. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 69–83. Springer Berlin Heidelberg, 2010.
51. Y. Li, A. Albarghouthi, Z. Kincad, A. Gurfinkel, and M. Chechik. Symbolic Optimization with SMT Solvers. In *POPL*, 2014.
52. A. Lodi. Mixed Integer Programming Computation. In *50 Years of Integer Programming 1958-2008*, pages 619–645. Springer-Verlag, 2009.
53. P. Manolios and V. Papavasileiou. Ilp modulo theories. In *CAV*, pages 662–677, 2013.
54. J. Marques-Silva, J. Argelich, A. Graa, and I. Lynce. Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4):317–343, 2011.
55. J. P. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. In Biere et al. [20], February 2009.
56. MiniZinc. `www.minizinc.org`.

57. A. Nadel and V. Ryvchin. Bit-vector optimization. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2016*, volume 9636 of *LNCS*. Springer, 2016.
58. N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2717–2723. AAAI Press, 2014.
59. C. M. Nguyen, R. Sebastiani, P. Giorgini, and J. Mylopoulos. Multi-object reasoning with constrained goal models. *Requirements Engineering*, 2016. In print. Published online 24 December 2016. DOI: `http://dx.doi.org/10.1007/s00766-016-0263-5`.
60. C. M. Nguyen, R. Sebastiani, P. Giorgini, and J. Mylopoulos. Requirements Evolution and Evolution Requirements with Constrained Goal Models. In *Proceedings of the 37nd International Conference on Conceptual Modeling - ER16*, LNCS. Springer, 2016.
61. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *Proc. Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *LNCS*. Springer, 2006.
62. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.
63. O. Ohrimenko, P. J. Stuckey, and M. Codish. *Propagation = Lazy Clause Generation*, pages 544–558. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
64. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
65. R. Raman and I. Grossmann. Modelling and computational techniques for logic based integer programming. *Computers and Chemical Engineering*, 18(7):563 – 578, 1994.
66. D. Rayside, H.-C. Estler, and D. Jackson. The Guided Improvement Algorithm for Exact, General-Purpose, Many-Objective Combinatorial Optimization. Technical report, Massachusetts Institute of Technology, Cambridge, 07 2009.
67. N. W. Sawaya and I. E. Grossmann. A cutting plane method for solving linear generalized disjunctive programming problems. *Computing Chemical Engineering*, 29(9):1891–1913, 2005.
68. N. W. Sawaya and I. E. Grossmann. A hierarchy of relaxations for linear generalized disjunctive programming. *European Journal of Operational Research*, 216(1):70–82, 2012.
69. R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 3(3-4):141–224, 2007.
70. R. Sebastiani and S. Tomasi. Optimization in SMT with LA(Q) Cost Functions. In *IJCAR*, volume 7364 of *LNAI*, pages 484–498. Springer, July 2012.
71. R. Sebastiani and S. Tomasi. Optimization Modulo Theories with Linear Rational Costs. *ACM Transactions on Computational Logics*, 16(2), March 2015.
72. R. Sebastiani and P. Trentin. OptiMathSAT: A Tool for Optimization Modulo Theories. In *Proc. International Conference on Computer-Aided Verification, CAV 2015*, volume 9206 of *LNCS*. Springer, 2015.
73. R. Sebastiani and P. Trentin. Pushing the Envelope of Optimization Modulo Theories with Linear-Arithmetic Cost Functions. In *Proc. Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'15*, volume 9035 of *LNCS*. Springer, 2015.
74. R. Sebastiani and P. Trentin. On optimization modulo theories, maxsmt and sorting networks. In *Proc. Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'17*, volume 10205 of *LNCS*. Springer, 2017.
75. C. Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In P. van Beek, editor, *CP*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005.
76. S. Teso, R. Sebastiani, and A. Passerini. Structured learning modulo theories. *Artificial Intelligence*, 244:166–187, 2017.
77. A. Van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Conference on Requirements Engineering*, RE'01, pages 249–. IEEE Computer Society, 2001.
78. A. Vecchietti and I. Grossmann. Computational experience with logmip solving linear and nonlinear disjunctive programming problems. In *Proc. of FOCAPD*, pages 587–590, 2004.