ILP Modulo Theories

Panagiotis Manolios and Vasilis Papavasileiou

Northeastern University
{pete,vpap}@ccs.neu.edu

Abstract. We present Integer Linear Programming (ILP) Modulo Theories (IMT). An IMT instance is an Integer Linear Programming instance, where some symbols have interpretations in background theories. In previous work, the IMT approach has been applied to industrial synthesis and design problems with real-time constraints arising in the development of the Boeing 787. Many other problems ranging from operations research to software verification routinely involve linear constraints and optimization. Thus, a general ILP Modulo Theories framework has the potential to be widely applicable. The logical next step in the development of IMT and the main goal of this paper is to provide theoretical underpinnings. This is accomplished by means of BC(T), the Branch and Cut Modulo T abstract transition system. We show that BC(T) provides a sound and complete optimization procedure for the ILP Modulo T problem, as long as T is a decidable, stably-infinite theory. We compare a prototype of BC(T) against leading SMT solvers.

1 Introduction

The primary goal of this paper is to present the theoretical underpinnings of the Integer Linear Programming (ILP) Modulo Theories (IMT) framework for combining ILP with background theories. The motivation for developing the IMT framework comes from our previous work, where we used an ILP-based synthesis tool, CoBaSA (Component-Based System Assembly), to algorithmically synthesize architectural models using the actual production design data and constraints arising during the development of the Boeing 787 Dreamliner [16]. According to Boeing engineers, previous methods for creating architectural models required the *"cooperation of multiple teams of engineers working over long periods of time."* We were able to synthesize architectures in minutes, directly from the high-level requirements. What made this possible was the combination of ILP with a custom decision procedure for hard real-time constraints [16], *i.e.*, an instance of IMT.

ILP has been the subject of intensive research for more than five decades [13]. ILP solvers [1, 2] are routinely used to solve practical optimization problems from a diverse set of fields including operations research, industrial engineering, artificial intelligence, economics, and software verification. Based on our successful use of the IMT approach to solve architectural synthesis problems and the widespread applicability of ILP and optimization, we hypothesize that IMT has the potential to enable interesting new applications, analogous to what is currently happening with Satisfiability Modulo Theories [3, 10, 27, 8].

We introduce the theoretical underpinnings of IMT via the BC(T) framework (Branch and Cut Modulo T). BC(T) can be thought of as the IMT counterpart to the DPLL(T) architecture for lazy SMT [27]. BC(T) models the branch-and-cut family of algorithms for integer programming as an abstract transition system and allows plugging in theory solvers. Building on classical results on combining decision procedures [23, 31, 19], we show that BC(T) provides a sound and complete optimization procedure for the combination of ILP with stably-infinite theories. As a side-product of our theoretical study of IMT, we show how to bound variables while preserving optimality modulo the combination of Linear Integer Arithmetic and a stably-infinite theory.

The rest of the paper is organized as follows. In Section 2, we formally define IMT and provide an abstract BC(T) architecture for solving IMT problems. IMT can be seen as SMT with a more expressive core than propositional logic. We elaborate on the relationship between IMT and SMT in Section 3. We have implemented BC(T), using the SCIP MIP solver [2] as the core solver. We carried out a sequence of experiments, as outlined in Section 4. The first experiment shows that for our synthesis problems, ILP solvers [1, 2] outperform the Z3 SMT solver [8]. In the second experiment, we compared our prototype implementation with state-of-the-art SMT solvers [8, 14] on SMT-LIB benchmarks. An analysis of the results suggests that BC(T) is an interesting future alternative to the DPLL(T) architecture. We provide an overview of related work in Section 5 and conclude with Section 6.

2 BC(T)

In this section, we formally define IMT. We also provide a general BC(T) architecture for solving IMT problems. We describe BC(T) by means of a *transition system*, similar in spirit to DPLL(T) [27]. The BC(T) architecture allows one to obtain a solver for ILP Modulo T by combining a branch-and-cut ILP solver with a background solver for T.

2.1 Formal Preliminaries

An *integer linear expression* is a sum of the form $c_1 v_1 + \cdots + c_n v_n$ for integer constants c_i and variable symbols v_i. An *integer linear constraint* is a constraint of the form $e \bowtie r$, where e is an integer linear expression, r is an integer constant, and $\bowtie$ is one of the relations $<$, $\leq$, $=$, $>$, and $\geq$. An *integer linear formula* is a set of (implicitly conjoined) integer linear constraints. We will use propositional connectives over integer linear constraints and formulas as appropriate and omit $\wedge$ when this does not cause ambiguity (*i.e.*, juxtaposition will denote conjunction). An *integer linear programming (ILP) instance* is a pair C, O, where C is an integer linear formula, and the *objective function* O is an integer linear expression. Our goal will always be *minimizing* the objective function.

We assume a fixed set of variables $\mathcal{V}$. An *integer assignment* A is a function $\mathcal{V} \rightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. We say that an assignment A *satisfies* the constraint $c = (c_1 v_1 + \cdots + c_n v_n \ \bowtie \ r)$ (where $\bowtie$ is one of the relations $<$, $\leq$, $=$, $>$, $\geq$, and every v_i is in $\mathcal{V}$) if $\sum_i c_i \cdot A(v_i) \ \bowtie \ r$. An assignment A satisfies a formula C if it satisfies every constraint $c \in C$. A formula C is *integer-satisfiable* or *integer-consistent* if there is an assignment A that satisfies C. Otherwise, it is called *integer-unsatisfiable* or *integer-inconsistent*.

A signature Σ consists of a set Σ^C of constant symbols, a set Σ^F of function symbols, a set Σ^P of predicate symbols, and a function $ar : \Sigma^F \cup \Sigma^P \rightarrow \mathbb{N}^+$ that assigns a non-zero natural number (the arity) to every function and predicate symbol. A Σ-formula is a first-order logic formula constructed using the symbols in Σ. A Σ-theory T is a closed set of Σ-formulas (*i.e.*, T contains no free variables). We will write theory in place of Σ-theory when Σ is clear from the context (similarly for terms and formulas).

Example 1. Let $\Sigma_{\mathcal{A}}$ be a signature that contains a binary function read, a ternary function write, no constants, and no predicate symbols. The theory $T_{\mathcal{A}}$ of arrays (without extensionality) is defined by the following formulas [21]:

$$\forall a \ \forall i \ \forall e \ [\mathsf{read}(\mathsf{write}(a,i,e),i) = e]$$
$$\forall a \ \forall i \ \forall j \ \forall e \ [i \neq j \Rightarrow \mathsf{read}(\mathsf{write}(a,i,e),j) = \mathsf{read}(a,j)].$$

A formula F is T-satisfiable or T-consistent if $F \wedge T$ is satisfiable in the first-order sense (*i.e.*, there is an interpretation that satisfies it). A formula F is called T-unsatisfiable or T-inconsistent if it is not T-satisfiable. For formulas F and G, F T-entails G (in symbols $F \models_T G$) if $F \wedge \neg G$ is T-inconsistent.

Definition 1. *Let $\Sigma_{\mathcal{Z}}$ be a signature that contains the constant symbols $\{0, \pm 1, \pm 2, \ldots\}$, a binary function symbol $+$, a unary function symbol $-$, and a binary predicate symbol $\leq$. The theory of Linear Integer Arithmetic, which we will denote by $\mathcal{Z}$, is the $\Sigma_{\mathcal{Z}}$-theory defined by the set of closed $\Sigma_{\mathcal{Z}}$-formulas that are true in the standard model (an interpretation whose domain is $\mathbb{Z}$, in which the symbols in $\Sigma_{\mathcal{Z}}$ are interpreted according to their standard meaning over $\mathcal{Z}$).*

We will use relation symbols like $<$ that do not appear in $\Sigma_{\mathcal{Z}}$, and also multiplication by a constant (which is to be interpreted as repeated addition); these are only syntactic shorthands. We will frequently view an integer assignment A as the set of formulas $\{v = A(v) \mid v \in \mathcal{V}\}$, where $A(v)$ is viewed as a $\Sigma_{\mathcal{Z}}$-term. An integer assignment A viewed as a set of formulas is always $\mathcal{Z}$-consistent. If A is an integer assignment and A satisfies an integer linear formula C, it is also the case that $A \models_{\mathcal{Z}} C$. If A is an integer assignment, T is a theory and F is a formula, we will say that A is a T-model of F if A is T-consistent and $A \models_{\mathcal{Z} \cup T} F$. Note that a T-model is not a first-order model.

A *Σ-interface atom* is a Σ-atomic formula (*i.e.*, the application of a predicate symbol or equality), possibly annotated with a variable symbol, *e.g.*, $(x = y)^v$. The meaning of a Σ-interface atom with no annotation remains the same. An annotated Σ-interface atom ϕ^v denotes $\phi \Leftrightarrow v > 0$. A set of Σ-interface atoms will often be used to denote their conjunction.

Definition 2 (ILP Modulo T Instance). *An* ILP Modulo (Theory) T instance*, where the signature Σ of T is disjoint from $\Sigma_{\mathcal{Z}}$, is a triple of the form C, I, O, where C is an integer linear formula, I is a set of Σ-interface atoms, and O is an objective function. The variables that appear in both C and I are called interface variables.*

An ILP Modulo T instance can be thought of as an integer linear program that contains terms which have meaning in T. In Definition 2, the interface atoms (elements of I) are separated from the linear constraints, *i.e.*, there are no Σ-terms embedded within integer linear constraints. This is not a restriction, because every set of $(\Sigma \cup \Sigma_{\mathcal{Z}})$-atomic formulas can be written in separate form [19, "Variable Abstraction"].

Example 2. Let Σ be a signature that contains the unary function symbol f. The formula $f(x+1) + f(y+2) \geq 3$ (where x and y are variable symbols) can be written in separate form as $C = \{v_3 + v_4 \geq 3, v_1 = x + 1, v_2 = y + 2\}$ and $I = \{v_3 = f(v_1), v_4 = f(v_2)\}$. C is an integer linear formula; I is a set of Σ-interface atoms; and Σ is disjoint from $\Sigma_{\mathcal{Z}}$. Variable abstraction introduced new variables, $v_1, \ldots, v_4$. C and I only share variable symbols.

Let A be the assignment $\{x = 2, y = 1, v_1 = 3, v_2 = 3, v_3 = 3, v_4 = 0\}$. Clearly $A \models_{\mathcal{Z}} C$. However, $A \not\models_{\emptyset} I$, where $\emptyset$ stands for the *theory of uninterpreted functions* (also called the empty theory, because it has an empty set of formulas). The reason is that $v_1 = v_2$ but $f(v_1) \neq f(v_2)$. In contrast, the assignment $A' = \{x = 2, y = 1, v_1 = 3, v_2 = 3, v_3 = 3, v_4 = 3\}$ is a $\emptyset$-model of $C \wedge I$ per our definition, as A' is $\emptyset$-consistent and $A' \models_{\mathcal{Z} \cup \emptyset} C \wedge I$.

2.2 Transition System

Definition 3 (Difference Constraint). *A difference constraint is a constraint of the form $v_i \leq c$ or $v_i - v_j \leq c$, where v_i and v_j are integer variables and c is an integer constant.*

Definition 4 (Subproblem). *A subproblem is a pair of the form $\langle C, D \rangle$, where C is a set of constraints and D is a set of difference constraints.*

In a subproblem $\langle C, D \rangle$, we distinguish between the arbitrary constraints in C and the simpler constraints in D in order to provide a good interface for the interaction between the core ILP solver and background theory solvers that only understand difference logic, *i.e.*, a limited fragment of $\mathcal{Z}$. It is the responsibility of the core solver to notify the theory solver about the difference constraints that hold. Difference constraints are clearly a special case of integer linear constraints.

Definition 5 (State). *A state of* BC(T) *is a tuple $P \parallel A$, where P is a set of subproblems, and A is either the constant* `None`*, or an assignment. If A is an assignment, it can optionally be annotated with the superscript $-\infty$.*

Our abstract framework maintains a list of open subproblems, because it is designed to allow different branching strategies. This is in contrast to an algorithm like CDCL that does not keep track of subproblems explicitly. There, subproblems are implicit, *i.e.*, backtracking can reconstruct them. ILP solvers branch over non-Boolean variables in arbitrary ways, thus mandating that we explicitly record subproblems.

In a state $P \parallel A$, the assignment A is the best known (T-consistent) solution so far, if any. It has a superscript $-\infty$ if it satisfies all the constraints, but is not optimal because the IMT instance admits solutions with arbitrarily low objective values. If this is the case, it is useful to provide an assignment and to also report that no optimal assignment exists.

The interface atoms I and the objective function O are not part of the BC(T) states because they do not change over time. $\mathsf{obj}(A)$ denotes the value of the objective function O under assignment A: if $O = \sum_i c_i v_i$, then $\mathsf{obj}(A) = \sum_i c_i \cdot A(v_i)$. The objective function itself is not an argument to obj because it will be clear from the context which objective function we are referring to. For convenience, we define $\mathsf{obj}(\mathtt{None}) = +\infty$ and $\mathsf{obj}(A^{-\infty}) = -\infty$. Function $\mathsf{lb}(\langle C, D\rangle)$ returns a lower bound for the possible values of the objective function O for the subproblem $\langle C, D\rangle$: by definition, there is no A such that A satisfies $C \wedge D$ and $\mathsf{obj}(A) < \mathsf{lb}(\langle C, D\rangle)$.

Figure 1 defines the *transition relation* $\longrightarrow$ of BC(T) (a binary relation over states). In the rules, c and d always denote integer linear constraints and difference constraints. C (possibly subscripted) denotes an integer linear formula (set of integer linear constraints), while D denotes a set of difference constraints. $C\ c$ stands for the set union $C \cup \{c\}$, under the implicit assumption that $c \notin C$; similarly for $D\ d$. C and D are always well-formed sets, *i.e.*, they contain no syntactically duplicate elements. P and P' stand for sets of syntactically distinct subproblems, while A and A' are integer assignments. $P \uplus Q$ denotes the union $P \cup Q$, under the implicit assumption that the two sets are disjoint. The intuitive meaning of the different BC(T) rules is the following:

`Branch`
: Case-split on a subproblem $\langle C, D\rangle$, by replacing it with two or more different subproblems $\langle C_i, D\rangle$. If there is a satisfying assignment for $C \wedge D$, this assignment will also satisfy $C_i \wedge D$ for some i, and conversely.

`Learn`, T-`Learn`, `Propagate`
: Add an entailed constraint (in the case of `Learn` and T-`Learn`) or difference constraint (`Propagate`) to a subproblem. T-`Learn` takes the theory T into account. T-`Learn` is strictly more powerful than `Learn`. We retain the latter as a way to denote transitions that do not involve theory reasoning.

`Forget`
: Remove a constraint entailed by the remaining constraints of a subproblem.

`Drop`, `Prune`
: Eliminate a subproblem either because it is unsatisfiable (`Drop`), or because it cannot lead to a solution better than the one already known.

`Branch`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C_i, D\rangle \mid 1 \le i \le n\} \parallel A$$
$$\text{if } \begin{cases} n > 1 \\ D \models_{\mathcal{Z}} (C \Leftrightarrow \bigvee_{1\le i\le n} C_i) \\ C_i \text{ are syntactically distinct} \end{cases}$$

`Learn`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C\ c, D\rangle\} \parallel A$$
$$\text{if } C \wedge D \models_{\mathcal{Z}} c$$

T-`Learn`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C\ c, D\rangle\} \parallel A$$
$$\text{if } C \wedge D \wedge I \models_{\mathcal{Z}\cup T} c$$

`Propagate`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C, D\ d\rangle\} \parallel A$$
$$\text{if } C \wedge D \models_{\mathcal{Z}} d$$

`Forget`

$$P \uplus \{\langle C\ c, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C, D\rangle\} \parallel A$$
$$\text{if } C \wedge D \models_{\mathcal{Z}} c$$

`Drop`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \parallel A$$
$$\text{if } C \wedge D \text{ is integer-inconsistent}$$

`Prune`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \parallel A$$
$$\text{if } \begin{cases} A \ne \texttt{None} \\ \mathsf{lb}(\langle C, D\rangle) \ge \mathsf{obj}(A) \end{cases}$$

`Retire`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \parallel A'$$
$$\text{if } \begin{cases} A' \text{ is a } T\text{-model of } C \wedge D \wedge I \\ \mathsf{obj}(A') < \mathsf{obj}(A) \\ \text{for any } T\text{-model } B \text{ of } C \wedge D \wedge I,\ \mathsf{obj}(A') \le \mathsf{obj}(B) \end{cases}$$

`Unbounded`

$$P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow \emptyset \parallel A'^{\,-\infty}$$
$$\text{if } \begin{cases} A' \text{ is a } T\text{-model of } C \wedge D \wedge I \\ \mathsf{obj}(A') \le \mathsf{obj}(A) \\ \text{for any integer } k, \text{ there exists a } T\text{-model } B \text{ of } C \wedge D \wedge I \\ \quad \text{such that } \mathsf{obj}(B) < k \end{cases}$$

Fig. 1. The BC(T) Transition System

`Retire`, `Unbounded`
: The solution to a subproblem becomes the new incumbent solution, as long as it improves upon the objective value of the previous solution. If there are solutions with arbitrarily low objective values, we don't need to consider other subproblems.

The observant reader will have noticed that the T-`Learn` rule is very powerful, *i.e.*, it allows for combined $\mathcal{Z} \cup T$-entailment. This is in pursuit of generality. Our completeness strategy (Theorem 3) will not depend in any way on performing combined arithmetic and theory reasoning, but only on extracting equalities and disequalities from the difference constraints. Entailment modulo $\mathcal{Z} \cup T$ is required if we want to learn clauses, because they are represented as linear constraints. Interesting implementations of BC(T) may go beyond clauses and apply T-`Learn` for theory-specific cuts.

We define the binary relations $\longrightarrow^{+}$ and $\longrightarrow^{*}$ over BC(T) states as follows: $S \longrightarrow^{+} S'$ if $S \longrightarrow S'$, or there exists some state Q such that $S \longrightarrow^{+} Q$ and $Q \longrightarrow S'$. $S \longrightarrow^{*} S'$ if $S = S'$ or $S \longrightarrow^{+} S'$. When convenient, we will annotate a transition arrow between two BC(T) states with the name of the rule that relates them, for example $S \underset{\texttt{Branch}}{\longrightarrow} S'$.

A *starting state* for BC(T) is a state of the form $\{\langle C, \emptyset \rangle\} \parallel \texttt{None}$, where C is the set of integer linear constraints of an ILP Modulo T instance. A *final state* is a state of the form $\emptyset \parallel A$ (A can also be `None`, or an assignment annotated with $-\infty$).

2.3 Soundness and Completeness

Throughout this Section, we assume an IMT instance with objective function O and a set of interface atoms I. Theorems 1 and 2 characterize BC(T) soundness. A version of this paper with proofs is available through arXiv [20].

Theorem 1. *For a formula C, if $\{\langle C, \emptyset \rangle\} \parallel \texttt{None} \longrightarrow^{*} \emptyset \parallel \texttt{None}$, then $C \wedge I$ is $\mathcal{Z} \cup T$-unsatisfiable.*

Theorem 2. *For a formula C and an assignment A, if*

$$\{\langle C, \emptyset \rangle\} \parallel \texttt{None} \longrightarrow^{*} \emptyset \parallel A$$

where $A \neq \texttt{None}$, then (a) A is a T-model of $C \wedge I$, and (b) there is no assignment B such that B is a T-model of $C \wedge I$ and $\mathsf{obj}(B) < \mathsf{obj}(A)$.

Definition 6 (Stably-Infinite Theory). *A Σ-theory T is called stably-infinite if for every T-satisfiable quantifier-free Σ-formula F there exists an interpretation satisfying $F \wedge T$ whose domain is infinite.*

Definition 7 (Arrangement). *Let E be an equivalence relation over a set of variables V. The set*

$$\alpha(V, E) = \ \{x = y \mid xEy\} \ \cup \ \{x \neq y \mid x, y \in V \text{ and not } xEy\}$$

is the arrangement of V induced by E.

Note that $\mathcal{Z}$ is a stably-infinite theory. We build upon the following result on the combination of signature-disjoint stably-infinite theories:

Fact 1 (Combination of Stably-Infinite Theories [23, 31, 19]) *Let T_i be a stably-infinite Σ_i-theory, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. Also, let Γ_i be a conjunction of Σ_i literals. $\Gamma_1 \cup \Gamma_2$ is $(T_1 \cup T_2)$-satisfiable iff there exists an equivalence relation E of the variables shared by Γ_1 and Γ_2 such that $\Gamma_i \cup \alpha(V, E)$ is T_i-satisfiable, for $i = 1, 2$.*

Decidability for the combination of $T_1 = \mathcal{Z}$ and another stably-infinite theory follows trivially, as we can pick an arrangement over the variables shared by the two sets of literals non-deterministically and perform two T_i-satisfiability checks. We show that BC(T) can be applied in a complete way by meeting the hypotheses of Fact 1.

Theorem 3 (Completeness). BC(T) *provides a complete optimization procedure for the ILP Modulo T problem, where T is a decidable stably-infinite theory.*

Proof (Sketch). Let $\mathcal{C}, I, O$ be an ILP Modulo T instance. Assume that

$$\{\langle \mathcal{C}, \emptyset \rangle\} \parallel \texttt{None} \longrightarrow^* P \parallel A,$$

and that for every $\langle C, D \rangle \in P$ the following conditions hold: (a) there is an equivalence relation E_D over the set of interface variables V of the ILP Modulo T instance, such that $D \models_{\mathcal{Z}} \alpha(V, E_D)$, and (b) either $D \models_{\mathcal{Z}} v > 0$ or $D \models_{\mathcal{Z}} v \leq 0$ for every v that appears as the annotation of an interface atom in I. Then we can solve the IMT instance to optimality as follows. For every subproblem $\langle C, D \rangle \in P$, $C \wedge D \wedge I$ $\mathcal{Z} \cup T$-entails the following set of literals:

$$\begin{aligned} \{\phi \mid \phi \in I \text{ and } \phi \text{ is not annotated}\} &\cup \\ \{\phi \mid \phi^v \in I \text{ and } D \models_{\mathcal{Z}} v > 0\} &\cup \\ \{\neg\phi \mid \phi^v \in I \text{ and } D \models_{\mathcal{Z}} v \leq 0\} &\cup \\ \alpha(V, E_D) & \end{aligned}$$

If the set of literals is T-unsatisfiable, then $C \wedge D \wedge I$ is $\mathcal{Z} \cup T$-unsatisfiable. If it is T-satisfiable, any integer solution for $C \wedge D$ will be a T-model. For the T-unsatisfiable subproblems, we apply T-`Learn` to learn an integer-infeasible constraint (*e.g.*, $0 < 0$) and subsequently apply `Drop`. If all the subproblems are T-unsatisfiable, we reach a final state $\emptyset \parallel A$. If there are T-satisfiable subproblems, it suffices to let a (complete) branch-and-cut ILP algorithm run to optimality, as we have already established T-consistency. The basic steps of such algorithms can be described by means of BC(T) steps. Note that unbounded objective functions do not hinder completeness: it suffices to recognize an unbounded subproblem [4] and apply `Unbounded`.

A systematic branching strategy can guarantee that after a finite number of steps, the difference constraints of every subproblem entail an arrangement. For every pair of interface variables x and y and every subproblem, we apply the `Branch` rule to obtain three new subproblems, each of which contains one of the constraints $x - y < 0$, $x - y = 0$, and $x - y > 0$. The `Propagate` rule then applies to all three subproblems. Similarly, we branch to obtain a truth value for $v > 0$ for every v that appears as the annotation of an interface atom.

3 SMT as IMT

In Section 2, we provided a sound and complete optimization procedure for the combination of ILP and a stably-infinite theory (Theorems 1, 2, and 3). We will now demonstrate how to deal with propositional structure, so that we can use this procedure for SAT Modulo $\mathcal{Z} \cup T$ problems, where T is stably-infinite. In essence, our goal is to flatten propositional structure into linear constraints.

3.1 Bounding $\mathcal{Z} \cup T$ Instances

As a prerequisite for dealing with propositional structure, we show how to bound integer terms in quantifier-free formulas while preserving $\mathcal{Z} \cup T$-satisfiability. We build upon well-known results for ILP [5]. Similar ideas have been applied to $\mathcal{Z}$ [30]. Our results go beyond the bounds for $\mathcal{Z}$, in that we take into account background theories and objective functions.

We will say that a term is Σ-rooted if (at its root) it is an application of a function symbol from the signature Σ. Let Σ_0 and Σ_1 be signatures such that $\Sigma_0 \cap \Sigma_1 = \emptyset$. Given a $\Sigma_0 \cup \Sigma_1$-formula F, we will refer to the Σ_i-rooted terms that appear directly under predicate and function symbols from Σ_{1-i} as the Σ_i-interface terms in F. Interface terms are the ones for which variable abstraction (Example 2) introduces fresh variables.

Let Σ be a signature such that $\Sigma_{\mathcal{Z}} \cap \Sigma = \emptyset$, and F be a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula. We denote by $\mathsf{intf}_{\mathcal{Z}}(F)$ and $\mathsf{intf}_{\Sigma}(F)$ the sets of $\Sigma_{\mathcal{Z}}$-interface terms and Σ-interface terms in F, and by $\mathsf{intf}(F)$ the union $\mathsf{intf}_{\Sigma}(F) \cup \mathsf{intf}_{\mathcal{Z}}(F)$. Let $\mathsf{atoms}_{\mathcal{Z}}(F)$ be the set of atomic formulas in F that are applications of $\leq$; without loss of generality, we will assume that formulas contain no arithmetic equalities or other kinds of inequalities. Also, let $\mathsf{maxc}(F)$ be the maximum absolute value among integer coefficients in F plus one, and $\mathsf{vars}_{\mathcal{Z}}(F)$ be the set of variable symbols that appear directly under predicate and function symbols from $\Sigma_{\mathcal{Z}}$. By $o(M)$ we denote the interpretation of linear expression o under the first-order model M. We finally define $\mathsf{bounds}(F, \rho) = \{-\rho \leq t \wedge t \leq \rho \mid t \in \mathsf{vars}_{\mathcal{Z}}(F) \cup \mathsf{intf}(F)\}$, for positive integers ρ.

Theorem 4. *Let Σ be a signature such that $\Sigma_{\mathcal{Z}} \cap \Sigma = \emptyset$ and T be a stably-infinite Σ-theory. Let F be a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula and o an objective function. Let $k = |\,\mathsf{atoms}_{\mathcal{Z}}(F)| + |\,\mathsf{intf}(F)| + |\,\mathsf{vars}_{\mathcal{Z}}(F)| - 1$, $m = |\,\mathsf{intf}_{\mathcal{Z}}(F)| + k$, and $n = |\,\mathsf{intf}(F)| + |\,\mathsf{vars}_{\mathcal{Z}}(F)|$. Finally, let*

$$\rho \;=\; (2n+k)^3[(m+2)\,\mathsf{maxc}(F)]^{4m+12}.$$

*If there is a first-order model M such that $M \models F \wedge \mathcal{Z} \wedge T$ and M is a finite optimum for F with respect to o (*i.e.*, there is some integer constant c such that $M \models o = c$ and there is no model M' such that $M' \models F \wedge \mathcal{Z} \wedge T$ and $M' \models o < c$), then $\{F\} \cup \mathsf{bounds}(F, \rho) \cup \{o = o(M)\}$ is $\mathcal{Z} \cup T$-satisfiable.*

We provide a proof through arXiv [20]. Intuitively, given a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula F and an objective function o, Theorem 4 allows us to bound the integer terms of F while preserving (finite) optimality. [1]

3.2 Propositional Structure

Let Σ be a signature such that $\Sigma_{\mathcal{Z}} \cap \Sigma = \emptyset$ and T be a stably-infinite Σ-theory. Throughout this Section, F will be a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula and O will be an objective function. We show how to encode F as the conjunction of a set C of integer linear constraints and a set I of Σ-interface atoms, while preserving optimality with respect to O. We apply a Tseitin-like algorithm, *i.e.*, we recursively introduce $\{0, 1\}$-constrained variables for subformulas of F.

The most interesting part is dealing with predicate symbols from Σ and $\Sigma_{\mathcal{Z}}$. For the former we simply introduce annotated Σ-interface atoms, *e.g.*, $[p(x)]^v$. For $\Sigma_{\mathcal{Z}}$, we can assume that we are only confronted with inequalities of the form $\phi = (\sum_i \; c_i \cdot v_i \leq r)$, because other relations can be expressed in terms of $\leq$ and the propositional connectives. Also, we only have to deal with sums over variable symbols, because variable abstraction takes care of terms that involve Σ. We define a variable $v(\phi)$ such that $v(\phi) \Leftrightarrow \phi$ as follows. By bounding all variables as per Theorem 4, we compute m and k such that $m < \sum_i c_i \cdot v_i \leq k$ always holds. The direction $v(\phi) \Rightarrow \phi$ can be expressed as $\sum_i c_i \cdot v_i \leq r + (k - r) \cdot (1 - v(\phi))$; for the opposite direction we have $\sum_i c_i \cdot v_i > r + (m - r) \cdot v(\phi)$.

With atomic formulas taken care of, what remains is propositional connectives; we encode them by using clauses in the standard fashion. Clauses appear as part of our collection of ILP constraints: $\vee_i l_i$ is equivalent to $\sum_i l_i \geq 1$. (For translating a clause to a linear expression, a negative literal $\neg v_i$ appears as $1 - v_i$ while a positive literal remains intact.)

Note that the (possibly astronomical) coefficients we compute only serve the purpose of representing formulas as sets of linear constraints. Their magnitude does not necessarily have algorithmic side-effects. In the worst case, the initial continuous relaxation will be weak, but relaxations will become stronger once we start branching on the Boolean variables. This is no worse than Lazy SMT, where linear constraints are only applicable once the SAT core assigns the corresponding Boolean variables.

4 Implementation and Experiments

IMT first appeared in the context of architectural synthesis for aerospace systems [16]. Our approach combined an ILP solver with a custom decision procedure for real-time constraints. We implemented the combination in the CoBaSA tool. The CoBaSA manifestation of IMT predates BC(T). More recently, we implemented a BC(T)-based solver, which we call Inez.

[1] A solver that relies on Theorem 4 for bounding can detect unboundedness by imposing the additional constraint $o < o(M)$, re-computing bounds, and solving the resulting instance. If the updated instance is satisfiable, the original is unbounded.

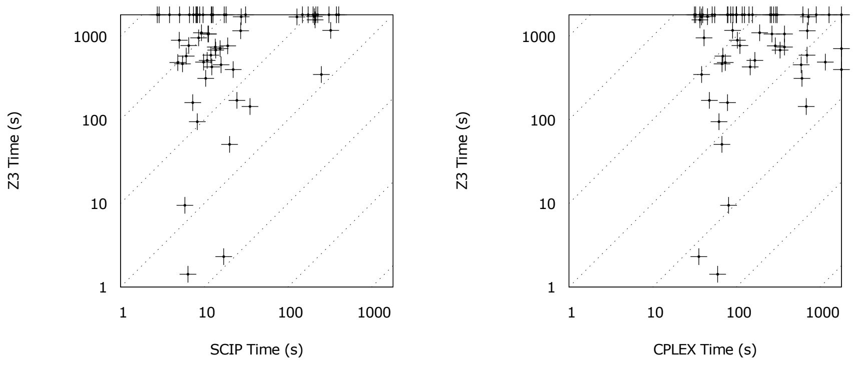


Fig. 2. Z3 versus SCIP and CPLEX (Synthesis Instances)

Our experimental evaluation is twofold. First, we show that an ILP core is essential for the practicality of our synthesis approach. This part of the evaluation does not deal with BC(T) in any way, but it nevertheless provides evidence that IMT enables new applications. Second, we compare our BC(T) prototype against Z3 [8] and MathSAT [14] using benchmarks from the SMT-LIB.

4.1 Motivation

In the past, we applied CoBaSA to solve the architectural synthesis problem for the real, production data from the 787, which was provided to us by Boeing [16]. We have made available a family of 60 benchmark instances derived from Boeing problems, 47 of which are unsatisfiable. [2] We will use these instances to evaluate the suitability of SMT and ILP solvers as the core of a combination framework for synthesis, which is a key application area for IMT.

We briefly describe the synthesis problem that gives rise to our benchmarks. The basic components for this problem are cabinets (providing resources like CPU time, bandwidth, battery backup, and memory), software applications (that consume resources), and global memory spaces (that also consume resources). Applications and memories have to be mapped to cabinets subject to various constraints, *e.g.*, resource allocation and fault tolerance. Applications communicate via a publish-subscribe network. Messages are aggregated into virtual links that are multicast. The network and messages are subject to various constraints, *e.g.*, bandwidth and scheduling constraints. The instances differ in the numbers of different components, the amounts of different resources, and the collection of structural and scheduling requirements they encode.

The instances are $\{0, 1\}$-ILP (also known as Pseudo-Boolean). There are multiple ways to encode $\{0, 1\}$-ILP problems as SMT-LIB instances. A direct translation led to SMT problems that Z3 could not solve, so we tried several encodings, most of which yielded similar results. One encoding was significantly

[2] `http://www.ccs.neu.edu/home/vpap/benchmarks.html`

better than the rest, and it works as follows. Some of the linear constraints are clauses, *i.e.*, of the form $\sum l_i \geq 1$ for literals l_i. It makes sense to help SMT solvers by encoding such constraints as disjunctions of literals instead of inequalities. To do this, we declare all variables to be Boolean. Since almost all variables also appear in arithmetic contexts where they are multiplied by constants greater than 1, we translate such constraints as demonstrated by the following example: the linear constraint $x+y+2z \geq 2$ becomes `(>= (+ (ite x 1 0) (ite y 1 0) (ite z 2 0)) 2)`.

Figure 2 visualizes the behavior of Z3 versus SCIP and CPLEX. SCIP solves all instances, while CPLEX solves all but 3. Z3 solves 5 out of 13 satisfiable and 30 out of 47 unsatisfiable instances. Strictly speaking, the only theory involved is $\mathcal{Z}$. However, the instances do contain collections of scheduling theory lemmas [16] recorded by CoBaSA in the process of solving synthesis problems. Therefore, our setup simulates the kinds of queries a core solver would be confronted with, when coupled with our scheduling solver. With suitability for synthesis as the evaluation criterion, this is the most rigorous comparison we can perform without implementing and optimizing the combination of SMT with scheduling. Both ILP solvers significantly outperform Z3, demonstrating the potential of a general ILP-based combination framework.

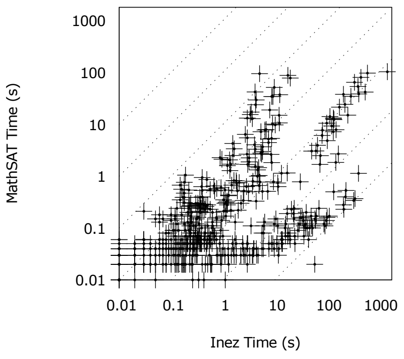
4.2 BC(T) Implementation

Inez is implemented as an unobtrusive extension of SCIP. Namely, we have extended SCIP with a congruence closure procedure (*constraint handler* in SCIP terms), and also provide an SMT-LIB frontend. The overall architecture of SCIP matches BC(T). Subproblems (called *nodes*) are created by branching (`Branch`) and eliminated by operations semantically very similar to `Drop`, `Prune`, `Retire`, and `Unbounded`. SCIP employs various techniques for cut generation (`Learn`).

Like most modern MIP solvers, SCIP relies heavily on linear relaxations. While not explicitly mentioned in BC(T), linear relaxations fit nicely: (a) lb relies on continuous relaxations, as the best integral solution can be at most as good as the best non-integral solution. (b) Solutions to relaxations frequently guide branching, *e.g.*, if a solution assigns a non-integer value r to variable v, it makes sense to branch around r ($v \geq \lceil r \rceil$ or $v \leq \lfloor r \rfloor$). (c) If some relaxation is infeasible, then the corresponding subproblem is infeasible and `Drop` applies, while (d) `Retire` or `Unbounded` applies to T-consistent integer solutions.

BC(T) proposes difference constraints as a channel of communication with theory solvers (`Propagate` rule). Inez implements `Propagate` as follows. For every pair of variables x and y whose (dis)equality is of interest to the theory solver, Inez introduces a variable $d_{x,y}$ and imposes the constraint $d_{x,y} = x - y$. When SCIP fixes the lower bound of $d_{x,y}$ to l, the theory solver is notified of the difference constraint $l \leq x-y$ (similarly for the upper bound). We generally need quadratically many such auxiliary variables. This is not necessarily a practical issue, because most pairs of variables are irrelevant.

Our congruence closure procedure takes offsets into account [25]. In addition to standard propagation based on congruence closure, Inez applies techniques



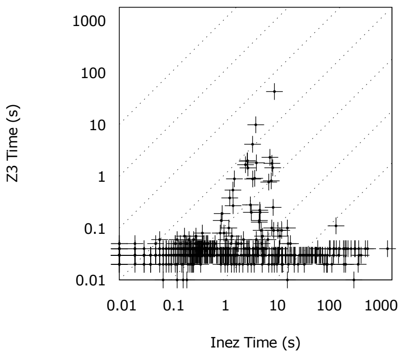


Fig. 3. Inez versus Z3 and MathSAT (SMT-LIB Instances)

specific to the integer domain. Notably, if x is bounded between a and b, and for every value of k in $[a, b]$, $f(k)$ is bounded between l and u, it follows that $l \leq f(x) \leq u$, *i.e.*, we can impose bounds on $f(x)$. a and b do not have to be constants, *e.g.*, it may be the case that $m \leq d_{x,y} = x - y \leq n$ and $f(y + m), \ldots, f(y + n)$ are bounded. We apply this idea dynamically (to benefit from local bounds) and not just as preprocessing.

BC(T) does not preclude techniques that target special classes of linear constraints. For example, an implementation can use the two-watched-literal scheme to accelerate Boolean Constraint Propagation on clauses. SCIP implements such techniques. Note that IMT does not strive to replace propositional reasoning, but rather to shift a broader class of constraints to the core solver.

4.3 BC(T) on SMT-LIB

We experimentally evaluate Inez against MathSAT and Z3, based on the most relevant SMT-LIB category, which is `QF_UFLIA` (Quantifier-Free Linear Integer Arithmetic with Uninterpreted Functions). Z3 and MathSAT solve all 562 benchmarks, and so does Inez. While Inez is generally slower than the more mature SMT solvers, the majority of the benchmarks (338) require less than a second, 462 benchmarks require less than 10 seconds, and 528 less than 100 seconds. The integer-specific kind of propagation outlined in Section 4.2 is crucial; we only solve 490 instances with this technique disabled. Figure 3 visualizes our experiments.

Interestingly, the underlying SCIP solver learns no cutting planes whatsoever for 362 out of the 562 instances. For the remaining instances the number of cuts is limited. Namely, 126 instances lead to a single cut, 61 lead to 2 cuts, and the remaining 13 instances lead to 9 cuts or less. Based on this observation, the branching part of Inez's branch-and-cut algorithm is being stressed here. We have not yet tried to optimize branching heuristics, so there is plenty of room for improvement. More importantly, the instances are not representative of arithmetic-heavy optimization problems, where we would expect more cuts.

A final observation is that SCIP performs floating-point (FP) arithmetic, which may lead to wrong answers. Interestingly, Inez provides no wrong answers for the benchmark set in question, *i.e.*, the instances do not pose numerical difficulties. The fact that we learn very few cutting planes partially explains why. There is little room for learning anything at all, let alone for learning something unsound.

5 Related Work

Branch-and-Cut: Branch-and-cut algorithms [22] combine branch-and-bound with cutting plane techniques, *i.e.*, adding violated inequalities (cuts) to the linear formulation. Different cut generation methods have been studied for general integer programming problems, starting with the seminal work of Gomory [13]. Cuts can also be generated in a problem-specific way, *e.g.*, for TSP [15]. Problem-specific cuts are analogous to theory lemmas in IMT.

Nelson-Oppen: The seminal work of Nelson and Oppen [23] provided the foundations for combining decision procedures. Tinelli and Harandi [31] revisit the Nelson-Oppen method and propose a non-deterministic variant for non-convex stably-infinite theories. Manna and Zarba provide a detailed survey of Nelson-Oppen and related methods [19].

SMT: ILP Modulo Theories resembles Satisfiability Modulo Theories, with ILP as the core formalism instead of SAT. SMT has been the subject of active research over the last decade [3, 10, 27, 8]. Nieuwenhuis, Oliveras and Tinelli [27] present the abstract DPLL(T) framework for reasoning about lazy SMT. Different fragments of Linear Arithmetic have been studied as background theories for SMT [11, 14]. Extensions of SMT support optimization [26, 6, 29].

Generalized CDCL: A family of solvers that generalize CDCL-style search to richer logics recently emerged [17, 28, 18, 9]. This research direction can be viewed as progress towards SMT with a non-propositional core. Our work is complementary, in the sense that we do not focus on the core solver, but rather provide a way to combine a non-Boolean core with theories.

Inexact Solvers: Linear and integer programming solvers generally perform FP (and thus inexact) calculations. Faure et al. experiment with the inexact CPLEX solver as a theory solver [12] and observe wrong answers. For many applications, numerical inaccuracies are not a concern, *e.g.*, the noise in the model overshadows the floating point error intervals. However, accuracy is often critical. Recent work [24, 7] proposes using FP arithmetic as much as possible (especially for solving continuous relaxations) while preserving safety. IMT solvers can be built on top of both exact and inexact solvers.

6 Conclusions and Future Work

We introduced the ILP Modulo Theories (IMT) framework for describing problems that consist of linear constraints along with background theory constraints. We did this via the BC(T) transition system that captures the essence of branch-and-cut for solving IMT problems. We showed that BC(T) is a sound and complete optimization procedure for the combination of ILP with stably-infinite theories. We conducted a detailed comparison between SMT and IMT.

Many interesting research directions now open up. We could try to relax requirements on the background theory (*e.g.*, stably-infiniteness, signature disjointness) while preserving soundness and completeness. We anticipate interesting connections between IMT and other paradigms, *e.g.*, SMT, constraint programming, cut generation, and decomposition. Additionally, the BC(T) architecture seems to allow for significant parallelization. Finally, we believe that IMT has the potential to enable interesting new applications.

Acknowledgements

We would like to thank Harsh Raju Chamarthi, Mitesh Jain, and the anonymous reviewers for their valuable comments and suggestions.

References

1. CPLEX. See `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.
2. T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universitat Berlin, 2007.
3. C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *CAV*, 2002.
4. R. H. Byrd, A. J. Goldman, and M. Heller. Recognizing Unbounded Integer Programs. *Operations Research*, 35(1):140–142, 1987.
5. Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 2nd edition, 1998.
6. A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS*, 2010.
7. W. Cook, T. Koch, D. Steffy, and K. Wolter. An Exact Rational Mixed-Integer Programming Solver. In *IPCO*, 2011.
8. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
9. L. de Moura and D. Jovanovic. A Model-Constructing Satisfiability Calculus. In *VMCAI*, 2013.
10. L. de Moura and H. Ruess. Lemmas on Demand for Satisfiability Solvers. In *SAT*, 2002.
11. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, 2006.
12. G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodriguez-Carbonell. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In *SAT*, 2008.

13. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the AMS*, 64:275–278, 1958.
14. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, 2012.
15. M. Grotschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.
16. C. Hang, P. Manolios, and V. Papavasileiou. Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints. In *CAV*, 2011.
17. D. Jovanovic and L. de Moura. Cutting to the Chase: Solving Linear Integer Arithmetic. In *CADE*, 2011.
18. A. Kuehlmann, K. McMillan, and M. Sagiv. Generalizing DPLL to Richer Logics. In *CAV*, 2009.
19. Z. Manna and C. Zarba. Combining Decision Procedures. In *10th Anniversary Colloquium of UNU/IIST*, 2002.
20. P. Manolios and V. Papavasileiou. ILP Modulo Theories. *CoRR*, abs/1210.3761, 2012.
21. J. McCarthy. Towards a Mathematical Science of Computation. In *Congress IFIP-62*, 1962.
22. J. E. Mitchell. Branch-and-Cut Algorithms for Combinatorial Optimization Problems. In *Handbook of Applied Optimization*, pages 223–233. Oxford University Press, 2000.
23. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *TOPLAS*, 1:245–257, 1979.
24. A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming*, 99:283–296, 2004.
25. R. Nieuwenhuis and A. Oliveras. Congruence Closure with Integer Offsets. In *LPAR*, 2003.
26. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *SAT*, 2006.
27. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
28. Scott Cotton. Natural domain SMT: a preliminary assessment. In *FORMATS*, 2010.
29. R. Sebastiani and S. Tomasi. Optimization in SMT with LA(Q) Cost Functions. In *IJCAR*, 2012.
30. S. A. Seshia and R. E. Bryant. Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds. In *LICS*, 2004.
31. C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In *FroCoS*, 1996.