Synthesizing MILP Constraints for Efficient and Robust Optimization

JINGBO WANG, University of Southern California, United States
AARTI GUPTA, Princeton University, United States
CHAO WANG, University of Southern California, United States

While mixed integer linear programming (MILP) solvers are routinely used to solve a wide range of important science and engineering problems, it remains a challenging task for end users to write correct and efficient MILP constraints, especially for problems specified using the inherently non-linear Boolean logic operations. To overcome this challenge, we propose a syntax guided synthesis (SyGuS) method capable of generating high-quality MILP constraints from the specifications expressed using arbitrary combinations of Boolean logic operations. At the center of our method is an extensible domain specification language (DSL) whose expressiveness may be improved by adding new integer variables as decision variables, together with an iterative procedure for synthesizing linear constraints from non-linear Boolean logic operations using these integer variables. To make the synthesis method efficient, we also propose an over-approximation technique for soundly proving the correctness of the synthesized linear constraints, and an under-approximation technique for safely pruning away the incorrect constraints. We have implemented and evaluated the method on a wide range of benchmark specifications from statistics, machine learning, and data science applications. The experimental results show that the method is efficient in handling these benchmarks, and the quality of the synthesized MILP constraints is close to, or higher than, that of manually-written constraints in terms of both compactness and solving time.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Applied computing** → **Operations research**; • **Theory of computation** → **Logic and verification**.

Additional Key Words and Phrases: Syntax Guided Synthesis, Data Science, Statistics, Machine Learning

ACM Reference Format:
Jingbo Wang, Aarti Gupta, and Chao Wang. 2023. Synthesizing MILP Constraints for Efficient and Robust Optimization. *Proc. ACM Program. Lang.* 7, PLDI, Article 184 (June 2023), 24 pages. https://doi.org/10.1145/3591298

1 INTRODUCTION

Many important science and engineering problems may be formulated as mixed integer linear programming (MILP) problems and then solved using off-the-shelf MILP solvers such as Gurobi [Bixby 2007]. Here, the word "mixed" means that the variables appearing in these linear constraints and the objective function are of either integer or real type, where the integer variables are often used to model binary decisions. Thus, MILP constraints are capable of expressing a wide range of decision problems and optimization problems, including binary classification, path planning in dynamical and partially known systems, and provisioning long-haul network capacity in cloud services. In

Authors' addresses: Jingbo Wang, University of Southern California, Los Angeles, United States, jingbow@usc.edu; Aarti Gupta, Princeton University, Princeton, United States, aartig@cs.princeton.edu; Chao Wang, University of Southern California, Los Angeles, United States, wang626@usc.edu.

2475-1421/2023/6-ART184
https://doi.org/10.1145/3591298

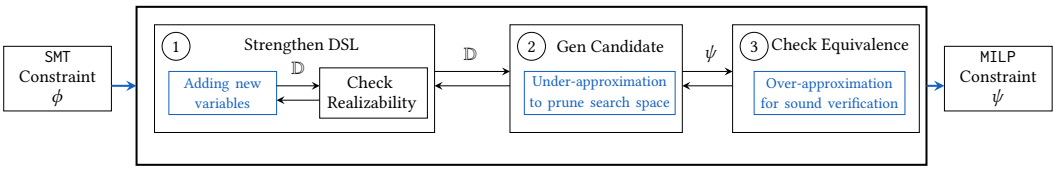


Fig. 1. SynMio – our method for synthesizing the MILP constraint ψ from the SMT specification ϕ.

addition, with the rapid advance in the runtime performance of modern MILP solvers, they are also increasingly used in statistics, machine learning, and data science applications.

However, writing *correct* and *efficient* MILP constraints remains a challenging task [Aghaei et al. 2019; Bertsimas and Dunn 2017; Chang et al. 2012; Chang 2012; Paulsen and Wang 2022a,b; Wang et al. 2022], for two reasons. First, many problems cannot be directly expressed using a conjunction of linear integer and real arithmetic constraints. Instead, they require the use of Boolean variables, together with Boolean logic operations such as AND ($\wedge$), OR ($\vee$), and Implication ($\rightarrow$) between the constraints. In the context of MILP, these Boolean logic operations are non-linear operations. Since the decision variant of MILP is NP-complete, which has the same complexity as Boolean satisfiability (SAT), it is possible to model Boolean variables using integer variables (with 0 and 1 values) and model Boolean logic operations using linear arithmetic constraints. However, in practice, the modeling process is labor intensive and error prone. Second, when domain experts write MILP constraints, they often do not use the objective function as is; instead, they use heuristic simplifications or convex proxies to replace the exact objective function with an alternative, easier-to-solve objective function. However, this kind of manual optimization is challenging for end users.

To overcome the aforementioned limitations, we propose a *solver-independent* and *generally-applicable* method to automate the process of transforming specifications with Boolean logic operations to *correct*, *efficient* and *robust* MILP constraints that can be solved using any MILP solver. By *generally-applicable*, we mean the method works for arbitrary combinations of Boolean logic operations. Fig. 1 shows the overall flow. The input of our method is a specification ϕ expressed in the LIA/LRA fragments of the SMT-LIB format. Here, LIA stands for linear integer arithmetic while LRA stands for linear real arithmetic; thus, ϕ consists of both linear arithmetic constraints and Boolean logic operations. The output is a constraint ψ in the MILP format; that is, ψ is a conjunction of linear integer/real arithmetic constraints.

Internally, our method uses *syntax guided synthesis* (SyGuS) [Alur et al. 2013], which relies on a domain specification language (DSL) to define the search space for ψ, and enumerative search to explore the candidates for ψ that are equivalent to ϕ. While we can fix the set of grammar rules of the DSL, denoted $\mathbb{D}$, the set of variables cannot be fixed *a priori*. Thus, we propose to iteratively strengthen the DSL. This is accomplished by starting from an initial DSL under which ψ may be unrealizable, and strengthening it until ψ becomes realizable. Strengthening is accomplished by adding new decision variables, which effectively decompose the input specification ϕ and improve the expressiveness of $\mathbb{D}$. We leverage the counterexamples (CEX) generated by both the realizability checking step and the verification subroutine as feedback, to decide which new variables to add, to maximize the improvement in each refinement step.

Given a sufficiently expressive DSL, there are still two challenges in synthesizing the MILP constraint ψ. First, due to the large search space, there may be too many candidates, each of which must be checked to see if it is equivalent to ϕ. Second, verifying the equivalence of ϕ and ψ may be time-consuming. To overcome these challenges, we propose an under-approximation technique to

quickly falsify the equivalence of ψ and ϕ, and then use the result to prune the search space. The under-approximation is to reduce *integer* variables to small *bit-vectors*. This drastically speeds up the falsification subroutine of SyGuS. We also propose an over-approximation technique to soundly verify the equivalence of ψ and ϕ. Since the goal of verification is to prove the equivalence of ϕ and ψ for any data element, rather than solving the optimization problem encoded by ψ over concrete data elements, we over-approximate the equivalence verification by rewriting complex operations in ψ and ϕ as uninterpreted functions (UF) and then connecting them using high-level relations.

Sometimes, there may be multiple candidates for ψ, all of which are equivalent to the specification ϕ, and yet some are significantly easier to solve than others. In such a case, it is important for our method to choose the most efficient candidate. Consider the specification $(x_8 = 0 \rightarrow x_{11} = 0) \wedge (x_8 = 0 \rightarrow x_{17} = 0)$ as an example of ϕ, where $0 \leq x_7,\ x_8,\ x_{11} \leq 1$, and $x_7,\ x_8,\ x_{11} \in \mathbb{Z}$. The two candidates may be $\psi : x_{11} + x_{17} \leq 2x_8$ and $\psi' : x_{11} \leq x_8 \wedge x_{17} \leq x_8$. While both of them are equivalent to ϕ, our experience with modern MILP solvers shows that ψ is significantly easier to solve. This is due to the fact that, while both correspond to the same discrete set $\mathcal{F}$ of feasible points, the polyhedra formed by linear relaxations (i.e., relaxing variables of ψ and ψ' from integer to continuous value) are different. The polyhedra for ψ is significantly closer to the convex hull of $\mathcal{F}$. To choose the high-quality candidate, we combine our heuristic decomposition of the input specification ϕ (used to add new variables to the DSL) with a *convex-hull* based optimization to allow the relaxation space to get closer to the convex hull. Details of this optimization are presented in Section 4.

Our method differs significantly from existing techniques. Broadly speaking, there are two lines of related work. One line of work, which is adopted by some MILP solvers, is to provide limited support for Boolean logic expressions internally. For example, both Gurobi [Bixby 2007] and CPLEX [CPLEX 2015] can linearize simple Boolean logic expressions by adding indicator variables and then explicitly branching on these variables. However, this may lead to exponential blowup; furthermore, since the linearization is solver-dependent, it cannot benefit other solvers. Another line of work is on linearizing Boolean logic expressions externally, but only for a restricted syntax. For example, the Big-M method [Cococcioni and Fiaschi 2021; Glover 1975] adds a large number for an artificial M variable, to ensure that a Boolean logic expression holds whenever a Boolean indicator variable evaluates to `False`. However, the M value has a significant and yet unpredictable impact on performance. Bertsimas et al. [Bertsimas et al. 2021] propose another way to reformulate simple Boolean logic expressions as convex binary optimization problems; however, it works only for a restricted syntax. To the best of our knowledge, our method is the first solver-independent and generally-applicable solution.

It is worth pointing out that the decision variant of an MILP problem can be solved using an SMT solver. However, its performance is not as competitive as MILP solvers. To demonstrate this, we have conducted an experiment using the *Protein Folding* example (also explained in Section 2.2). The state-of-the-art MILP solver, Gurobi [Bixby 2007], solved it in 0.23 seconds, while the state-of-the-art SMT solver, Z3 [Moura and Bjørner 2008], solved it in 18.27s. Although there have been efforts on extending SMT solvers to make them more efficient in solving both the decision and the optimization variants of the MILP problem, for example, in [Devriendt et al. 2021] and [King et al. 2014], the performance is still far from being competitive.

We have implemented our method in a tool and evaluated the tool on a diverse set of benchmarks. They include 38 specifications from statistics, machine learning, and data science applications [CPLEX 2015; Forrester and Greenberg 2008; Williams 2013]. To evaluate the quality of the synthesized MILP constraints, we have compared them with the MILP constraints manually written by domain experts. The results show that, in terms of compactness, the synthesized constraints are similar to the manually-written constraints; and in terms of MILP-solving performance, the

$$\phi \begin{cases} \phi_1 : a \geq 0 \rightarrow b = K1 \\ \phi_2 : a < 0 \rightarrow b = K2 \end{cases} \quad \psi^1 \begin{cases} a \geq minA * x \\ a \leq maxA - (maxA + 1) * x \\ b = K1 + (K2 - K1) * x \\ 0 \leq x \leq 1, x \in \mathbb{Z} \end{cases} \quad \psi^2 \begin{cases} \psi_1 : a \geq a * x \\ \psi_2 : a \leq a - x * (a + 1 + a) \\ \psi_3 : b = K1 + (K2 - K1) * x \\ \psi_4 : 0 \leq x \leq 1, x \in \mathbb{Z} \end{cases} \tag{1}$$

Fig. 2. An example SMT specification ϕ (left), the equivalent MILP constraint ψ^1 written by a domain expert (middle), and the equivalent MILP constraint ψ^2 synthesized by our method (right). Here, x is a new integer variable (whose value is either 0 or 1), while $minA$ and $maxA$ are constant values.

synthesized constraints are either similar to, or better than, the manually-written constraints. Specifically, our synthesized constraints are able to reduce the MILP-solving time by 9.8% on average and up to a maximum of 41%. To evaluate the efficiency of the synthesis tool, we have also analyzed its execution time, and quantified the impact of individual optimization techniques. Our experimental results show that the tool is able to synthesize all of the MILP constraints quickly, and our optimization techniques are effective in speeding up the synthesis process.

To summarize, this paper makes the following contributions:

- We propose a *solver-independent* and *generally-applicable* method for synthesizing *correct*, *efficient* and *robust* MILP constraints from specifications with Boolean logic operations.
- We propose an under-approximation technique to prune the search space, and propose an over-approximation technique to speed up equivalence verification.
- We propose a method for generating high-quality candidates by leveraging a *convex-hull* based optimization to decompose the input specification and strengthen the DSL.
- We implement our method and demonstrate its effectiveness on 38 statistical modeling benchmarks with a wide range of Boolean logic operations.

The remainder of this paper is organized as follows. First, we use examples to motivate our work in Section 2. Then, we present an overview of our method in Section 3. This is followed by the detailed algorithms for strengthening the DSL in Section 4, generating candidates in Section 5, verifying equivalence in Section 6, and domain-specific optimizations in Section 7. We present the experimental results in Section 8, the related work in Section 9, and the conclusions in Section 10.

2 MOTIVATION

In this section, we give an overview of our approach using two motivating examples.

2.1 Example 1: *From Online Q&A*

This example comes from the StackExchange website. The input specification can be expressed as **If** $a \geq 0$ **then** $b = K1$; **else** $b = K2$; where a and b are real-valued variables and $K1$ and $K2$ are constants. Given the input specification, it is easy to write down the formula expressed in the SMT-LIB format, $\phi := \phi_1 \wedge \phi_2$, where ϕ_1 and ϕ_2 are defined in Fig. 2 (left).

The four predicates $a \geq 0$, $a < 0$, $b = K1$ and $b = K2$ in ϕ are linear constraints that can be normalized to $\mathsf{a}x \leq \mathsf{b}$. The Boolean logic operator $\rightarrow$, which stands for "implies", can be transformed to elementary Boolean operations; that is, $A \rightarrow B$ is equivalent to $\neg A \vee B$.

While it is easy for end users to write the SMT specification ϕ, it is a challenging task to transform ϕ into a correct and efficient MILP constraint, since MILP does not allow disjunction ($\vee$) or implication ($\rightarrow$), or any combination of Boolean logic operators. Instead, an MILP constraint must be a conjunction of linear arithmetic constraints.

Boolean	ϕ	:=	$p \mid \phi \wedge \phi$	Arith Expr	a	:=	$a_0 \mid a + a \mid a * a \mid a\%a$
Atomic Pred	p	:=	$a \odot a$	Input Expr	a_0	:=	$c \mid var \mid \vec{A} \mid \texttt{ITE}(r,1,0)$
Comparator	$\odot$	:=	$= \mid \leq$	Array Expr	$\vec{A}$	:=	$\mathcal{A}[a] \mid \mathcal{A}[a][a]$
				Bool Relation	r	:=	$R(var, var) \mid R(var, \mathcal{A}) \mid R(\mathcal{A}, \mathcal{A})$

Fig. 3. Basic DSL for expressing the MILP constraint, where c denotes constant, var denotes variable and $\mathcal{A}$ denotes array. Both var and $\mathcal{A}$ are sets of elements extracted from the input SMT specification ϕ.

2.1.1 The MILP Constraints ψ^1 and ψ^2. Fig. 2 shows two equivalent MILP constraints. The one in the middle, denoted ψ^1, is written by a domain expert, while the one on the right, denoted ψ^2, is synthesized by our method.

Domain experts often adopt the Big-M method from the literature [Cococcioni and Fiaschi 2021; Glover 1975]. In this running example, x is a newly-added decision variable whose value is restricted to either 0 or 1, while $minA$ and $maxA$ are two constant values satisfying ($minA \leq a \wedge minA < 0 \ \wedge maxA \geq a \wedge maxA > 0$). In other words, $minA$ ($maxA$) must be a negative (positive) value smaller (bigger) than all possible values of the variable a. As such, the approach has two limitations. First, it requires *a priori* estimation of the minimal and maximal values of the variable a, which may not be easy to do. Second, the quality of the MILP constraint depends on the quality of these two bounds; loose bounds of $minA$ and $maxA$ will make the MILP constraint difficult to solve, or make the solving time unpredictable.

In contrast, our method is able to synthesize a correct, efficient and robust MILP constraint ψ^2, as shown in Fig. 2 (right). Unlike the manually-written constraint, it does not rely on the two constant values ($minA$ and $maxA$); thus, there is no need to estimate the minimal and maximal values of the variable a. Furthermore, our experience with modern MILP solvers shows that ψ^2 is significantly easier to solve than ψ^1. In the remainder of this subsection, we will explain, at a high level, how our method synthesizes the MILP constraint ψ^2 from the input specification ϕ.

2.1.2 Our Method for Synthesizing ψ^2. Our method starts by defining a domain specific language (DSL), and then strengthens the DSL until it is expressive enough to capture the MILP constraint ψ^2 equivalent to the input specification ϕ. During this strengthening process, the set of grammar rules of the DSL is fixed, while the set of variables of the DSL is expanded.

Fig. 3 shows the grammar rules of the DSL. Each rule maps a type (left-hand side of ":=") to a set of compatible values (right-hand side of ":="). For instance, the compatible values for atomic predicate are $\mathbb{D}[p] = \{a \odot a\}$, i.e., the linear arithmetic constraints. With the grammar rules fixed, the expressiveness of the DSL depends on c, var, r, and $\vec{A}$; they are the sets of constants, variables, relations, and arrays, respectively.

In this running example, the set of variables is $var = \{a, b, K1, K2\}$ initially. With these four variables, however, the equivalent MILP constraint ψ^2 is *unrealizable*. We must add new variables to the DSL to improve its expressiveness, until ψ^2 becomes realizable.

This is accomplished by first adding a Boolean indicator variable x, whose value is either `True` or `False`, to replace the predicates ($a \geq 0$) and ($a < 0$) in ϕ. We capture the relationship between a and x using ϕ_3: $(a < 0 \rightarrow x = \texttt{True}) \wedge (a \geq 0 \rightarrow x = \texttt{False})$.

With the addition of this new variable x, we have $var = \{a, b, K1, K2, x\}$, which improves the expressiveness of the DSL. Furthermore, the updated input specification is $\phi = \phi_1' \wedge \phi_2' \wedge \phi_3$, where ϕ_1' and ϕ_2' are new versions of ϕ_1 and ϕ_2 by replacing predicates $a \geq 0$ and $a < 0$ with $x =$`False` and $x =$`True`, respectively.

With the updated DSL $\mathbb{D}$ and input specification ϕ, our method checks the realizability of an equivalent MILP constraint again. This time, the answer becomes yes, and the resulting ψ^2 is shown

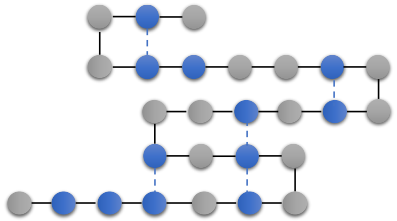


Fig. 4. The protein folding example.

For each pair of hydrophobic acids i and j,
we can match them if:
(1) they have an even number of acids between i and j
(2) there is exactly one fold between i and j
(3) i and j are not contiguous (i.e., $j > i+1$)

Fig. 5. Requirements for protein folding.

in Fig. 2 (right). While ψ^2 contains the product of program variables, it can still be modeled as linear constraints, since at least one of the two variables in a product is binary (e.g., x).

To understand why the synthesized MILP constraint ψ^2 is equivalent to the SMT specification ϕ, consider the following two cases:

- When $x = 0$, we can reduce ψ^2 to $\psi_1 : a \geq 0$, $\psi_2 : a \leq a$, and $\psi_3 : b = K1$.
- When $x = 1$, we can reduce ψ^2 to $\psi_1 : a \geq a$, $\psi_2 : 2a \leq -1$, and $\psi_3 : b = K2$.

In both cases, the result is consistent with $\phi_1 : a \geq 0 \rightarrow b = K1$ and $\phi_2 : a < 0 \rightarrow b = K2$.

The equivalence of ϕ and ψ^2 holds as long as a precondition $\Phi_{\leq}(a)$ holds. That is,

$$\forall a, b, K1, K2 \in \mathbb{R},\ x \in \{0, 1\}.\ \Phi_{\leq}(a) \rightarrow \phi(a, b, K1, K2, x) = \psi^2(a, b, K1, K2, x) \tag{2}$$

The precondition $\Phi_{\leq}(a)$ is defined as $a < 0 \rightarrow a \leq -0.5$ to remove strict inequality ($<$) in ϕ_2. Since strict inequality is not supported by the MILP solver, a tolerance value (-0.5) is introduced to convert $a < 0$ into $a \leq -0.5$.

One advantage of our method is that the correctness of ψ^2 is guaranteed by construction, since our method returns ψ^2 only after it formally proves the equivalence of ψ^2 and ϕ. This is in contrast to the manually-written ψ^1, for which domain experts must manually verify the correctness.

2.2 Example 2: Protein Folding

This is a molecular biology problem [Forrester and Greenberg 2008] where a chain of amino acids must be folded. Some of the amino acids are *hydrophobic* (water-hating) while others are *hydrophilic* (water-loving). The goal is to maximize the pairing of hydrophobic acids. Fig. 4 shows the folding in two dimensions, with hydrophobic amino acids marked by *blue* circles, and the matches marked by *blue* dashed lines.

Given a chain of 50 amino acids $A = \{1, 2, \ldots, 50\}$ and the subset of *hydrophobic* amino acids $H = \{2, 4, 5 \ldots\} \subset A$, for example, the goal here is to compute the best folding, to maximize the number of matchings of hydrophobic amino acids. Fig. 5 shows the matching constraints, where $i \in [1, 50]$ and $j \in [1, 50]$ are integer variables. Since the third constraint is straightforward, let us focus on the first two constraints (1) (2).

Based on the requirements in Fig. 5, end users will be able write the input specification $\phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \ldots$ shown in Fig. 6 (left). Here, ϕ_1 and ϕ_2 encode the first requirement of Fig. 5, while ϕ_3 encodes the second requirement, meaning that there is only one fold between hydrophobic acids i and j, if the two hydrophobic acids i and j match.

2.2.1 Using Boolean Relations. Inside ϕ, Boolean relations such as $M(i, j)$ and $f(k)$ are used to write the input specification symbolically. This is in contrast to the use of individual Boolean variables such as M_i_j for all $i \in [0, 50]$ and $j \in [0, 50]$. For example, $M(i, j) = \texttt{True}$ indicates that a hydrophobic acid i is matched with another hydrophobic acid j, for all hydrophobic acids

ϕ_1: $M(i,j) = 1 \rightarrow (i+j-1)\%2 = 0$
ϕ_2: $M(i,j) = 1 \wedge k = (i+j-1)/2 \rightarrow f(k) = 1$
ϕ_3: $M(i,j) = 1 \rightarrow \sum_{i \le k' < j} f(k') = 1$
ϕ_4: $x \leftrightarrow k = (i+j-1)/2$
ϕ_5: $y \leftrightarrow x \wedge (i+j-1)\%2 = 0$

ψ_1: $(i+j-1)\%2 \le 1 - M(i,j)$
ψ_2: $x + (i+j-1)\%2 \le 1,\ x + y \le 1$
ψ_3: $1 \le (i+j-1)\%2 + x + y,\ M(i,j) + x \le f(k) + 1$
ψ_4: $(i+j-1)/2 \le k + y * (i+j-1)/2$
ψ_5: $k \le (i+j-1)/2 + (k+1) * y$
ψ_6: $f(k) + M(i,j) + y \le 2$

Fig. 6. Input specifications are in $\{\phi_1, \phi_2, \phi_3\}$, specifications generated by our decomposition procedure are in $\{\phi_4, \phi_5\}$, and the synthesized MILP constraints are in $\{\psi_1 - \psi_6\}$.

$i + 1 < j \in H$, while $f(k) = \texttt{True}$ holds if and only if a fold occurs between the k-th and $(k+1)$-th amino acids in the chain, where $k \in A$.

One advantage of using Boolean relations in ϕ, as shown in Fig. 6, is that the specification ϕ is applicable to a chain of amino acids of any length, not just a chain of amino acids of length 50.

2.2.2 Rewriting the Accumulative Operations. Given this input specification ϕ, our method first simplifies ϕ, and then strengthens the DSL to ensure that the equivalent MILP constraint is realizable, before synthesizing ψ. Our simplification focuses on rewriting the accumulative operations such as $\sum_{i \in I}$ and $\bigwedge_{i \in I}$ using pre-defined rewriting rules.

These rewriting rules are derived from the semantics of the accumulative operators. For the $\sum_{i \in I}$ operator, there are two rules applicable to this example: $R_1 : \sum_{i \in I} g(i) \leftrightarrow g(i') + \sum_{i \in I \wedge i \neq i'} g(i)$ and $R_2 : \sum_{i \in I} g(i) = 0 \rightarrow g(i) = 0$.

With rule R_1, subformula ϕ_3 is rewritten as $f(k) + \sum_{i \le k' < j \wedge k' \neq k} f(k') = 1$. Specifically, assuming the LHS of ϕ_2 holds (e.g., $M(i,j) = \texttt{True}$, $k = (i+j-1)/2$), we obtain $f(k) = 1$ from the RHS of ϕ_2. Thus, ϕ_3 is rewritten as $\sum_{i \le k' < j \wedge k' \neq k} f(k') = 0$, implying $\sum_{i \le k' < j \wedge k' \neq (i+j-1)/2} f(k') = 0$. With rule R_2, ϕ_3 is finally simplified as follows: $M(i,j) \wedge k' \neq (i+j-1)/2 \rightarrow f(k') = 0$.

2.2.3 Checking Unrealizability. Next, our method checks if the equivalent MILP constraint ψ is realizable using the initial DSL ($\mathbb{D}$). This check is formulated as

$$\text{Find } \psi \in L(\mathbb{D}), \underline{\forall i, j, k, M(i,j), F(k).}\ \phi(i,j,k,M,F) = \psi(i,j,k,M,F) \tag{3}$$

Here, $L(\mathbb{D})$ represents the set of all possible formulas that can be expressed using the DSL $\mathbb{D}$. For this example, the result is *unrealizable*. To make it realizable, we must add new decision variables to the *var* set of the DSL. This is accomplished by replacing some of the predicates in ϕ with the new decision variables, until Eq. 3 says *realizable*. The detailed algorithm for checking realizability will be presented in Section 4.2.

There may be multiple ways of decomposing ϕ to add new variables. Below is an example.

$$x \leftrightarrow \underline{(i+j-1)\%2 = 0} \wedge y \leftrightarrow \underline{k = (i+j-1)/2} \wedge z \leftrightarrow \underline{M(i,j) \wedge \neg y} \wedge z1 \leftrightarrow \underline{M(i,j) \wedge y} \tag{4}$$

However, this may introduce too many new variables.

A better way is to introduce only two new variables x and y, as defined in the subformulas ϕ_4 and ϕ_5. Fig. 6 (right) shows the equivalent MILP constraint synthesized by our method.

3 OUR METHOD

The top-level procedure of our method, SynMio, is presented in Algorithm 1. It takes the SMT specification ϕ as input and returns the equivalent MILP constraint ψ as output.

Internally, it first identifies from the input specification ϕ the sets of constants, relations, variables, and arrays that appear in ϕ. Then, it uses these sets to initialize the DSL $\mathbb{D}$, whose grammar rules have been defined in Fig. 3. It also initializes S_E, which will be used to store the set of negative

Algorithm 1 Our synthesis method $\psi \leftarrow \textsc{SynMio}(\phi)$.

```
1: Let c, r, var and A be the sets of constants, relations, variables, and arrays appeared in specification φ
2: D ← InitDsl(c, r, var, A)
3: S_E ← ∅                                          ▷ S_E is the set of negative examples
4: do
5:     ⟨D, φ⟩ ← EnrichDsl(D, S_E, φ)                ▷ Strengthen DSL if S_E contains evidence that ψ is unrealizable
6:     do
7:         ψ ← GenCandi(D, S_E, φ)                  ▷ using under-approximation to prune the SyGuS search
8:         E ← VerifyEq(φ, ψ)                       ▷ using over-approximation to speed up verification
9:         S_E ← S_E ∪ {E}
10:    while E ≠ ∅ ∧ runtime < threshold
11: while E ≠ ∅
12: return ψ
```

examples. Here, a negative example is a set of concrete values for variables in the DSL, to show $\psi \neq \phi$. Initially, S_E is an empty set.

Inside the first loop (Line 4), our method first checks if the equivalent MILP constraint ψ is *realizable*. If there is evidence in S_E that ψ *may be unrealizable*, the DSL is strengthened until such negative examples no longer exist. This is done inside the subroutine EnrichDsl, which returns the strengthened $\mathbb{D}$ and the updated ϕ.

While the detailed algorithm of EnrichDSL will be presented in Section 4, here, it suffices to say that the *realizability-checking* procedure is designed to be sound but not necessarily complete, for efficiency reasons. That is, when it reports *unrealizable*, it means the equivalent MILP constraint is definitely *unrealizable*; however, when it reports *unknown*, the result is inconclusive. In the latter case, EnrichDsl returns the current $\mathbb{D}$ and ϕ without modification.

With the strengthened DSL, our method goes into the second loop, to generate ψ using the syntax guided synthesis (SyGuS) framework. Specifically, it uses the subroutine GenCandi to generate a candidate ψ, and then uses the subroutine VerifyEq to prove the equivalence of ψ and ϕ.

To improve performance, it uses under-approximation inside GenCandi to quickly prune the bad candidates, and uses over-approximation inside VerifyEq to speed up verification. Any negative examples generated by these two subroutines will be added to the set S_E.

Whenever ϕ and ψ are proved equivalent, VerifyEq returns an empty set (E), which allows our method to jump out of the loops and return the synthesized MILP constraint ψ (Line 12).

If the running time of the inner loop exceeds a predefined threshold, then the subroutine EnrichDsl is used to strengthen the DSL again, using the accumulated negative examples in S_E.

In the next three sections, we will present the detailed algorithms inside the three subroutines EnrichDsl (Sec 4), GenCandi (Sec 5), and VerifyEq (Sec 6).

4 STRENGTHENING THE DSL

In this section, we present the algorithm inside the subroutine EnrichDsl. The pseudo code is shown in Algorithm 2. The input of this subroutine consists of the DSL $\mathbb{D}$, the negative example set S_E, and the specification ϕ. The output consists of the updated versions of $\mathbb{D}$ and ϕ.

Internally, there are three steps. The first step, Unrealizable, checks whether the example set S_E contains any evidence to show that an MILP constraint equivalent to ϕ is unrealizable. The second step, Decomp, strengthens the DSL by decomposing ϕ to replace some of its predicates with new decision variables; adding these new variables to the *var* set of the DSL will improve its expressiveness. The third step, ConvexHullSplit, further decomposes ϕ to add more variables. However, unlike Decomp, the goal here is to increase the chance of generating a high-quality MILP constraint. In the remainder of this section, we explain these three steps in detail.

Algorithm 2 $\langle \mathbb{D}, \phi \rangle \leftarrow$ EnrichDsl$(\mathbb{D}, S_E, \phi)$

1: **while** Unrealizable$(\mathbb{D}, S_E, \phi)$ **do**
2: $\quad \langle \mathbb{D}, \phi \rangle \leftarrow$ Decomp$(\mathbb{D}, S_E, \phi)$
3: **end while**
4: $\langle \mathbb{D}, \phi \rangle \leftarrow$ ConvexHullSplit$(\mathbb{D}, \phi)$
5: **return** $\langle \mathbb{D}, \phi \rangle$

$$\phi \begin{cases} \phi_1 : x \geq 0 \wedge x < 1 \rightarrow & y \leq 10 \\ \phi_2 : x \geq 1 \wedge x < 2 \rightarrow & y \leq 5 \\ \phi_i : \ldots & \textit{(up to 10 implications)} \\ \phi_n : x \geq 10 \rightarrow & y \leq 10 \end{cases}$$

Fig. 7. Specification ϕ for *StackExchange 2873*.

4.1 Decomposing the Specification

We start with the subroutine Decomp. Instead of showing the pseudo code, which is straightforward, we illustrate the process using an example. The specification of this example, named *StackExchange 2873*, is shown in Fig. 7. That is, $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \phi_i \wedge \ldots \phi_n$ contains a conjunction of subformulas, each of which may have both linear constraints and Boolean logic operations such as Implication ($\rightarrow$). In our method, each subformula, ϕ_i, is treated as an input, for which an equivalent MILP constraint is synthesized.

$$\begin{array}{r|l} x \geq 0 \wedge x < 1 \rightarrow y \leq 10 & \phi_{1a} : \; x \geq 0 \Leftrightarrow w_1 \\ \hline w_1 \wedge x < 1 \rightarrow y \leq 10 & \phi_{1b} : \; x < 1 \Leftrightarrow w_2 \\ \hline w_1 \wedge w_2 \rightarrow y \leq 10 & \phi_{1c} : \; w_1 \wedge w_2 \Leftrightarrow z \\ \hline & \phi_{1d} : z \rightarrow y \leq 10 \end{array}$$

Fig. 8. The decomposed specifications from ϕ_1.

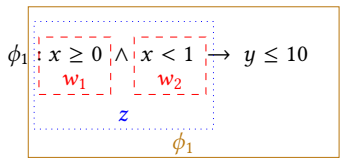


Fig. 9. Decomposing ϕ_1.

Consider ϕ_1 as our running example, Fig. 9 shows several ways of decomposing the antecedent, to replace predicates in the set $\{x \geq 0 \wedge x < 1,\ x \geq 0,\ x < 1\}$ with new decision variables. For example, Decomp may introduce a new variable z to replace ϕ_1.left() $= x \geq 0 \ \wedge \ x < 1$. Decomp may also introduce two more variables w_1 and w_2 to replace the children of ϕ_1.left() such that $w_1 = x \geq 0$ and $w_2 = x < 1$.

After introducing these three new variables, the specification ϕ_1 becomes $\phi_{1a} \wedge \phi_{1b} \wedge \phi_{1c} \wedge \phi_{1d}$, where the subformulas are defined in Fig. 8.

Our method will synthesize an equivalent MILP constraint for each of these subformulas. Thus, the final MILP constraint for ϕ_1 will be $\psi_1 = \psi_{1a} \wedge \psi_{1b} \wedge \psi_{1c} \wedge \psi_{1d}$, where ψ_{1d} is defined as $y \leq 10 + W * (1 - z)$ and the other subformulas are defined as follows:

$$\psi_{1a} \begin{cases} W * w_1 \geq x + eps \\ W * (1 - w_1) \geq 0 - x \end{cases} \quad \psi_{1b} \begin{cases} W * w_2 \geq 1 - x \\ W * (1 - w_2) \geq x - 1 + eps \end{cases} \quad \psi_{1c} \begin{cases} z \leq w_1 \wedge z \leq w_2 \\ w_1 + w_2 - 1 \leq z \end{cases} \tag{5}$$

Here, W and eps are symbolic variables such that W is the largest possible value among all expressions in ψ, and eps is a small constant used to make the inequality non-strict, similar to the tolerance value used in in Equation 2. For eps, we always add the bound constraint $0 < eps < 1$.

For W, we first extract all the subexpressions (i.e., $x + eps$ and $0 - x$ from ψ_{1a}) and then impose the constraint $W \geq x + eps \wedge W \geq 0 - x$.

4.2 Checking for Unrealizability

Now, we explain the subroutine Unrealizable, which decides whether S_E contains any evidence showing that a ϕ-equivalent MILP constraint is unrealizable. The verification problem is defined as

follows:

$$\text{Find } \psi \in L(\mathbb{D}),\ \underline{\forall var, r, \vec{A}.}\ \phi(var, r, \vec{A}) = \psi(var, r, \vec{A}) \tag{6}$$

However, this problem is difficult to solve.

Instead, we leverage the *unrealizability* verification technique proposed by Hu et al. [Hu et al. 2019, 2020]. The core idea is to reduce the *unrealizability* proof over all inputs (Eq. 6) to a proof over the finite number of examples in S_E:

$$\text{Find } \psi \in L(\mathbb{D}),\ \bigwedge\nolimits_{var, r, \vec{A} \in S_E}\ \phi(var, r, \vec{A}) = \psi(var, r, \vec{A}) \tag{7}$$

This is a sound (and not necessarily complete) procedure in that, if it is unrealizable according to Eq. 7, it is unrealizable according to Eq. 6; however, the reverse is not always true.

This procedure is practically fast, because the set S_E of negative examples is finite, which means the *unrealizability* proof of Eq. 7 can be encoded as a *reachability* problem in a non-deterministic program P_n, as shown by Hu et al. [Hu et al. 2019, 2020]. Each program path in P_n models a possible candidate expression that the DSL $\mathbb{D}$ can represent. Furthermore, P_n has an assertion (Eq. 8) such that, if the assertion cannot be falsified, it means the synthesis instance Eq. 7 is *unrealizable*.

$$\neg \bigwedge\nolimits_{var, r, \vec{A} \in S_E}\ \phi(var, r, \vec{A}) = \psi(var, r, \vec{A}) \tag{8}$$

4.3 Convex-hull Based Splitting

Finally, we explain the subroutine ConvexHullSplit. At this moment, the DSL has been strengthened such that there is no longer evidence in S_E showing that the ϕ-equivalent MILP constraint is unrealizable. Thus, the goal of adding new variables is not improving the expressiveness; instead, the goal is improving the quality of the synthesized MILP constraint.

Our method for adding new variables in ConvexHullSplit is inspired by the fact that, within MILP solvers, the original problem is often reformulated into a more relaxed problem before it is solved. There are many such reformulations, all of which keep the original part of the solution space, but may add more solutions to make the solution space convex. In this context, a *convex hull* is defined as the smallest convex solution space for the relaxed problems [Belotti et al. 2011].

Consider an example whose input specification ϕ is given as follows:

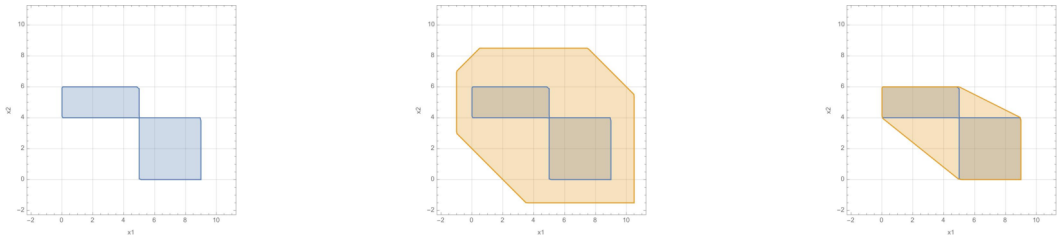
$$\phi \begin{cases} \phi_1 : k \rightarrow 5 \leq x1 \leq 9 \wedge 0 \leq x2 \leq 4 \\ \phi_2 : \neg k \rightarrow 0 \leq x1 \leq 5 \wedge 4 \leq x2 \leq 6 \end{cases} \qquad \psi^1 \begin{cases} \psi_1 : 5 - 7 * (1 - k) \leq x1 \leq 9 + 7 * (1 - k) \\ \psi_2 : -7 * (1 - k) \leq x2 \leq 4 + 7 * (1 - k) \\ \psi_3 : -7 * k \leq x1 \leq 5 + 7 * k \\ \psi_4 : 4 - 7 * k \leq x2 \leq 6 + 7 * k \end{cases} \tag{9}$$

The solution space of ϕ is shown as the blue shaded areas in Figure 10a, where $x1$ is the horizontal axis and $x2$ is the vertical axis.

Without adding any new variable, our method will synthesize ψ^1 as shown in the above Eq. 9. The solution space of ψ^1 is shown as the orange region in Figure 10b, which contains the blue region (actual solution space).

However, this is not a tight relaxation since the orange region over-approximates the blue region too much. This may lead to a long solving time, because MILP solvers routinely search for a solution in the relaxation space and then check if it also belongs to the actual solution space. Thus, a smaller relaxation space (*orange*) is more likely to reduce the solving time. However, without adding new variables, it is not possible for our synthesis method to get a tighter relaxation.

The reason is because ψ^1 is synthesized from ϕ by treating each of the subformulas (ϕ_1 and ϕ_2) in isolation. In other words, the synthesizer fails to capture and then utilize the correlation between



(a) ϕ (b) ψ^1 (c) ψ^2

Fig. 10. The solution spaces for ϕ, and two possible equivalent MILP constraints ψ^1 and ψ^2.

them. To see why ϕ_1 and ϕ_2 are correlated, consider the fact that they are defined over the same set of variables $\{x1, x2\}$.

One way to capture the correlation is to split variable $x1$ into $x1_k$ and $x1_{\neg k}$ such that $x1 = x1_k + x1_{\neg k}$. If $x1_k$ is activated, $x1_{\neg k}$ is equivalent to 0 (deactived); and vice versa. ConvexHullSplit is designed to handle this kind of scenarios, to split each (non-decision) variable to two new variables such that only one of them can be activated.

For the running example, the *var* set of the DSL will be augmented by the set $\{x1_k, x1_{\neg k}, x2_k, x2_{\neg k}\}$ of new variables. With the updated DSL, our method will synthesize the MILP constraint ψ^2 defined as follows:

$$\psi^2 \begin{cases} \psi_1 : x1 = x1_k + x1_{\neg k} \wedge x2 = x2_k + x2_{\neg k} \\ \psi_2 : 0 \leq x1_k \wedge 0 \leq x1_{\neg k} \\ \psi_3 : 5k \leq x1_k \leq 9k \wedge 0 \leq x1_{\neg k} \leq 5(1-k) \\ \psi_4 : 0 \leq x2_k \leq 4k \wedge 4(1-k) \leq x2_{\neg k} \leq 6(1-k) \end{cases} \tag{10}$$

Figure 10c shows the region of ψ^2, where the relaxation space (orange) is reduced to the *convex hull* of the blue region.

To summarize, the subroutine ConvexHullSplit is designed to introduce new variables by splitting the existing variables with the goal of capturing the relation between two subformulas ϕ_i and ϕ_j, where the antecedent of ϕ_i is the negation of the antecedent of ϕ_j. Our observation is that, by extracting the correlation between them and make the information available during synthesis, our method will be able to synthesize high-quality candidates.

5 GENERATING THE MILP CANDIDATES

In this section, we present the algorithm implemented in the subroutine GenCandi, which takes the DSL ($\mathbb{D}$), the example set (S_E) and the specification (ϕ) as input, and returns a candidate ψ as output. The pseudo code is shown in Algorithm 3. The baseline is a counterexample guided inductive synthesis (CEGIS) procedure [Alur et al. 2013], which enumerates candidates in a search space defined by $\mathbb{D}$ and, for each candidate ψ, checks whether it is equivalent to ϕ. To improve performance, however, we use under-approximation during the equivalence checking.

5.1 Under-approximation for Search Space Pruning

While variables in ϕ and ψ are either integer or real-valued variables, we redefine them as bit-vectors during the verification step. By reducing the length of the bit-vectors to 2 or 3, for example, the verification time will be drastically reduced.

Algorithm 3 Subroutine for generating candidate: $\psi \leftarrow \textsc{GenCandi}(\mathbb{D}, S_E, \phi)$.

```
1: Φ ← True; b ← Init();
2: while runtime < threshold do
3:     ψ ← Assemble(𝔻, S_E, φ, b)              ▷ ⋀_{E∈S_E} . Φ(E) → φ(E) = ψ(E)
4:     E ← FalsifyBit(φ, ψ, b)                 ▷ ∀E, 0 ≤ E ≤ 2^{b−1} − 1. Φ(E) → φ(E) = ψ(E)
5:     if E ≠ ∅ then                           ▷ E : counterexample
6:         if CheckCex(E, φ, ψ) then
7:             S_E ← S_E ∪ {E}                 ▷ E is a real counterexample – add to S_E
8:         else
9:             Φ ← UpdatePre(Φ, ψ, b, E)       ▷ E is a bogus counterexample – update precondition Φ
10:        end if
11:    else
12:        return ψ
13:    end if
14: end while
15: return ∅
```

Our verification step has three phases, each of which is slower and yet more accurate than the previous one. In the first phase, we check if the equivalence of ϕ and ψ holds on all examples $E \in S_E$. Since S_E is a finite set, this is captured by Ψ_1, defined as follows:

$$\Psi_1(\psi) := \bigwedge\nolimits_{E \in S_E} \Phi(E) \ \rightarrow \phi(E) = \psi(E) \tag{11}$$

Here, $\phi(E)$ denotes the value of ϕ on the example E, and $\phi(E) = \psi(E)$ means ϕ and ψ are equivalent for this particular example E.

$\Phi(E)$ is the precondition that is necessary to establish the equivalence in this bit-vector domain, defined as $\Phi(E) = \Phi_{\leq}(E) \wedge \Phi_{\uparrow}(E)$. The subformula $\Phi_{\leq}$ is used to remove strict inequality as explained using the example $\Phi_{\leq}(a)$ in Equation 2. The subformula $\Phi_{\uparrow}$, on the other hand, accounts for overflow/underflow in the bit-vector domain. Details of $\Phi_{\uparrow}$ will be explained in Section 5.2.

As shown in Line 3 of Algorithm 3, the subroutine Assemble returns the candidate only when ψ and ϕ satisfy Eq. 11.

In the second phase, we check if the equivalence holds for all values in the bit-vector domain; this is captured by Ψ_2, defined as follows:

$$\Psi_2(\psi) := \forall E,\ 0 \leq E \leq 2^{b-1} - 1\ .\ \Phi(E) \rightarrow \phi(E) = \psi(E) \tag{12}$$

Here, b is the length of the bit-vector. Again, the number of E is finite. As shown in Line 4 of Algorithm 3, the subroutine FalsifyBit is used to check Eq. 12.

In the third phase, we check if the equivalence holds in the unbounded integer/real domain $\mathbb{Z}/\mathbb{R}$. Since $\Phi_{\uparrow}$ is no longer needed to account for overflow and underflow in the bit-vector domain, the precondition $\Phi = \Phi_{\leq} \wedge \Phi_{\uparrow}$ reduces to $\Phi_{\leq}$. This is captured by Ψ_3, defined as follows:

$$\Psi_3(\psi) := \forall E \in \mathbb{Z},\ E' \in \mathbb{R}\ .\ \Phi_{\leq}(E, E') \rightarrow\ \phi(E, E') = \psi(E, E') \tag{13}$$

The reason why we need the third phase is because, even if $\Psi_2(\psi)$ holds, $\Psi_3(\psi)$ may not hold. The third phase is implemented in VerifyEq (used in Algorithm 1), and it will be discussed again in Section 6.

The three-phase approach presented above is designed to significantly speed up verification, by quickly falsifying equivalence in smaller domains (S_E and $[0, 2^{b-1} - 1]$) before falsifying in the unbounded domain. This effectively prunes the redundant search space.

The effectiveness of pruning is affected by the length of bit-vector. In our implementation, we set the length to 2 initially, and then keep increasing it as long as the equivalence checking is falsified in the third phase. This can be illustrated using the *Decentralized Planning* example below.

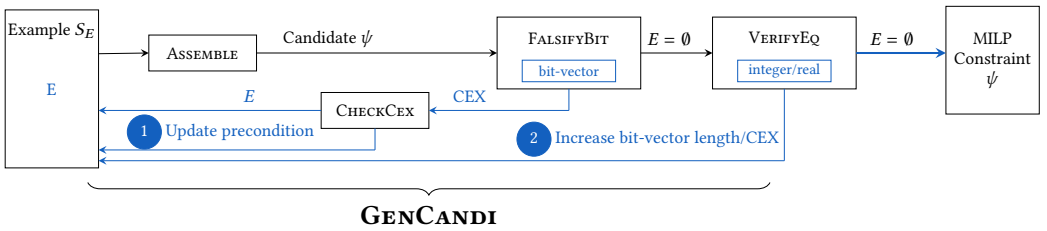


Fig. 11. The overall flow of the subroutine GenCandi presented in Algorithm 3.

In this example, the input specification ϕ is given as follows, together with the synthesized MILP constraint ψ^2:

$$\phi \begin{cases} mov(d,c,d2,c2) \rightarrow loc(d,c) \\ mov(d,c,d2,c2) \rightarrow loc(d2,c2) \\ loc(d,c) \wedge loc(d2,c2) \rightarrow mov(d,c,d2,c2) \end{cases} \qquad \psi^2 \begin{cases} mov(d,c,d2,c2) \leq loc(d,c) \\ mov(d,c,d2,c2) \leq loc(d2,c2) \\ loc(d,c) + loc(d2,c2) \leq 1 + mov(d,c,d2,c2) \end{cases} \tag{14}$$

Here, both loc and mov are Boolean relations, where $loc(d,c)$ means a factory d is located at a city c, and $mov(d,c,d2,c2)$ means moving d from c to $c2$ and renaming it as $d2$.

While ψ^2 is the equivalent MILP constraint synthesized by our method, it is not synthesized in one iteration. Initially, we use length-2 bit-vectors, and GenCandi produces a different candidate ψ^1, shown as follows:

$$mov(d,c,d2,c2) + loc(d2,c2) + mov(d,c,d2,c2) + loc(d,c) \leq (\#b01 + \#b01 + \#b01 + \#b01 + \#b01)$$

Here, $\#b01$ stands for a bit-vector constant of value 1. While this candidate ψ^1 satisfies Ψ_2 defined in Eq. 12, it does not satisfy Ψ_3 defined in Eq. 13.

That is why we increase the bit-vector length to 3, and then to 4, which leads to the correct candidate ψ^2.

5.2 Defining Preconditions

Recall that in Eq. 11 and Eq. 12, the preconditon $\Phi_{\uparrow}(E)$ is used to establish the equivalence in the bit-vector domain, to account for overflow and underflow. For example, the summation of two bit-vectors of length 2 may create a positive value that is too big, and the overflow may cause the value to become negative. Both overflow and underflow may lead to incorrect verification results.

Consider the example below, which illustrates the overflow. The input specification ϕ and the equivalent MILP constraint ψ are shown as follows:

$$\phi \begin{cases} \phi_1 : x > LB \rightarrow w1 = \texttt{True} \\ \phi_2 : x \leq LB \rightarrow w1 = \texttt{False} \end{cases} \qquad \psi \begin{cases} \psi_1 : W * w1 + LB \geq x \\ \psi_2 : W * (1 - w1) + x \geq LB + eps \end{cases} \tag{15}$$

While ψ is equivalent to ϕ in the integer and real domains, when we check them in bit-vector domain, the equivalence may no longer hold.

Thus, FalsifyBit produces a counterexample $E = \langle W = 7,\ w1 = \texttt{True},\ LB = 2,\ x = 4 \rangle$. While both $\phi(E)$ and $\psi(E)$ evaluate to `True`, in the length-4 bit-vector domain (signed), $\psi(E)$ evaluates to `False`. This is because 7 is the maximum positive value in this bit-vector domain. Thus $W * w1 + LB$, which is supposed to be 9, becomes -7. As a result, $\psi(E)$ evaluates to `False`, thus (mistakenly) failing Ψ_2 defined in Eq. 12. The consequence is that the correct candidate is missed.

To avoid the above scenario, we must add the precondition $\Phi_{\uparrow}$ into Eq. 11 and Eq. 12. Instead of enforcing constraints on a particular variable, we propose a general constraint to eliminate the overflow case of the summation over k-bit domain.

Assuming $k = 4$, we add to $\Phi_{\uparrow}$, for each arithmetic operation of the form $(+\ op1\ op2)$, the following constraint: $op1 \geq 0_4 \wedge op2 \geq 0_4 \rightarrow op1 + op2 \geq 0_4$. Similarly, for each $(+\ v1\ v2)$, we add the following constraint: $v1 \geq 0_4 \wedge v2 \geq 0_4 \rightarrow v1 + v2 \geq 0_4$.

If FalsifyBit generates a counterexample E, it is evident that $\phi(E) = \psi(E)$ fails in the bit-vector domain. However, it remains unclear if E is a valid counterexample in the integer and real domains. To make sure that E is a real counterexample, inside CheckCex (Line 6 of Algorithm 3), we check whether $\phi(E) = \psi(E)$ fails in the integer and real domains.

Figure 11 shows the details of generating and then validating the counterexample E, before adding it to the set S_E. In addition to FalsifyBit, the counterexample may also be generated by VerifyEq. In both cases, E is added to S_E only when it is confirmed to be a valid counterexample.

If, on the other hand, E is found to be a *spurious* counterexample, we identify the arithmetic operations causing the discrepancy (E being valid in the bit-vector domain but spurious in the integer/real domain). These arithmetic operations, e.g., $(+\ op1\ op2)$ and $(+\ v1\ v2)$, will be used to update the precondition $\Phi_{\uparrow}$, as explained in the paragraphs above.

To summarize, in Algorithm 3, the first step, Assemble, produces a candidate ψ that is equivalent to ϕ for all the examples $E \in S_E$ (Eq. 11). The second step, FalsifyBit, further checks the validity of ψ in the bit-vector domain (Eq. 12). If ψ is validated by FalsifyBit, it would be returned for further checking in VerifyEq (Line 12). Otherwise, a counterexample E is generated. As E is not guaranteed to be a real counterexample, CheckCex checks the validity of E. Based on the result of CheckCex, it would be either added to S_E (*real* counterexample) or used to define the precondition (*spurious* counterexample).

6 VERIFYING THE EQUIVALENCE

We now present the algorithm implemented in the subroutine VerifyEq, which is used in Algorithm 1 to soundly prove that ψ and ϕ are equivalent under the precondition $\Phi_{\leq}$ as defined in Eq. 13. This is accomplished by leveraging an off-the-self SMT solver. However, instead of proving the validity of

$$(\phi \equiv \psi) := (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi), \tag{16}$$

we use the SMT solver to check the satisfiability of the negated formula $(\phi \not\equiv \psi)$, which is equivalent to $(\phi \wedge \neg\psi \vee \neg\phi \wedge \psi)$. We say that ψ and ϕ are equivalent if and only if the negated formula is unsatisfiable (UNSAT).

However, directly applying the SMT solver to check the UNSAT of the negated formula may still be costly. As the sizes of ϕ and ψ increase, the verification time increases rapidly, which prevents our method from handling larger synthesis problems. To overcome this limitation, we propose to verify the equivalence of ϕ and ψ at a higher level of abstraction, i.e., by treating complex arithmetic operations as uninterpreted functions (UF). This is motivated by the fact that, during equivalent verification, our goal is to prove the equivalence of ϕ and ψ, without having to solve these arithmetic constraints on the potentially large amount of concrete data.

6.1 An Example

To understand the idea of verifying equivalence at a higher level of abstraction, let us revisit the protein folding example. Recall that, in this example, both ϕ and ψ are constraints over index variables such as i, j and k. Neither ϕ nor ψ refers to the concrete data. Here, the concrete data refers to the chain of 50 amino acids $A = \{1, 2, \ldots, 50\}$ and the subset of *hydrophobic* amino acids

ϕ_1: $M(i,j) \rightarrow E(i,j)$
ϕ_2: $M(i,j) \wedge S(i,j,k) \rightarrow f(k)$
ϕ_3: $M(i,j) \wedge \neg S(i,j,k1) \rightarrow \neg f(k1)$
ϕ_4: $x \leftrightarrow E(i,j) \wedge S(i,j,k)$

ψ_1: $M(i,j) \leq E(i,j), M(i,j) \leq S(i,j,k)$
ψ_2: $x \leq E(i,j), x \leq S(i,j,k), E(i,j) + S(i,j,k) \leq x + 1$
ψ_3: $M(i,j) + x \leq f(k) + 1$
ψ_4: $f(k1) + M(i,j) \leq 1 + S(i,j,k1)$

Fig. 12. Abstracted specification ϕ for protein folding (left) and ψ synthesized by our method (right). Arithmetic operations in $\phi_1 - \phi_3$ are replaced by UFs, and ϕ_4 is created by Decomp to add the new decision variable x.

$H = \{2, 4, 5, \ldots\}$. As a result, both ϕ and ψ are valid for arbitrarily long chains of amino acids, not merely the chain of length 50.

During equivalence verification, we want to keep i, j, k as unbounded integers in $[1, +\infty]$, as opposed to integers in a concrete range such as $[1, 50]$. In other words, the relations defined on these index variables must be kept symbolic.

- $E(i,j)$: `True` if $(i + j - 1)\%2 = 0$; `False` otherwise.
- $S(i,j,k)$: `True` if $(i + j - 1)/2 = k$; `False` otherwise.

During equivalence verification, we replace these symbolic relations as uninterpreted functions (UFs), to abstract away the correlation between the input (i, j) and the output of $E(i, j)$, except that $(i = i' \wedge j = j') \rightarrow E(i,j) = E(i',j')$. This is a sound over-approximation in that, as long as we prove the equivalence of ϕ and ψ using these UFs, the equivalence of ϕ and ψ in the integer/real domain is guaranteed.

Another benefit of verifying the equivalence at a higher level of abstraction is that our synthesis method can produce more compact candidates for ψ, e.g., by replacing the actual arithmetic computations in ϕ with UFs. Consider the input specification ϕ on the left-hand side of Figure 12, where the arithmetic computation has been abstracted away. For this ϕ, the number of new variables that needs to be added to the *var* set of the DSL will be drastically reduced.

Indeed, after adding one new variable, x, which is shown in ϕ_4 on the left-hand side of Figure 12, our method is able to synthesize the MILP constraint ψ shown on the right-hand side of Figure 12. This is significantly simpler than the one shown on the right-hand side of Figure 6.

6.2 Abstracting with Uninterpreted Functions

In our implementation of the subroutine VerifyEq, given the input specification ϕ, we first transform ϕ to the conjunctive normal form (CNF), consisting of $\phi_1 \wedge \phi_2 \wedge \ldots, \wedge \phi_n$, and then try to remove the redundant subformulas. For example, if the end user includes another subformula $\phi_5 : \neg E(i,j) \rightarrow \neg M(i,j)$ in the original specification ϕ, upon detecting that ϕ_5 is equivalent to $\phi_1 : M(i,j) \rightarrow E(i,j)$ and thus is redundant, we will remove it from ϕ.

Then, we use syntactic-level rewriting rules to replace accumulative operations (such as $\sum_{i \in I}$ and $\bigwedge_{i \in I}$) with functionally equivalent, non-accumulative operations. This has been explained in Section 2.2.2, while we introduced the protein folding example.

Next, we start abstracting the arithmetic operations to uninterpreted functions (UF). For example, in protein folding, the uninterpreted function $E(i,j)$ is introduced to replace the actual predicate $(i + j - 1)\%2 = 0$? `True` : `False`. We also use $\mathsf{UF}(i,j)$ to replace the array $\vec{A}[i][j]$ for efficient verification.

This leads to a sound over-approximation. By sound, we mean that the equivalence of ϕ and ψ using these UFs implies the equivalence of ϕ and ψ in the concrete domain. However, the reverse is not necessarily true: it is possible that ϕ and ψ are equivalent, and yet VerifyEq cannot prove the equivalence. Nevertheless, our experimental evaluation shows that, in practice, VerifyEq is effective in proving equivalence of ϕ and ψ in most cases.

Boolean	ϕ	:=	$p \mid \phi \wedge \phi \mid \phi_n \rightarrow \phi$	Arith Expr ND	a_n	:=	$a_0^n \mid a_n + a_n \mid a_n * a_n \mid a_n \% a_n$
Boolean ND	ϕ_n	:=	$p_n \mid \phi_n \wedge \phi_n$	Input Expr ND	a_0^n	:=	$c \mid var_n$
Atomic Pred ND	p_n	:=	$a_n \odot a_n$				

Fig. 13. DSL $\mathbb{D}_t$ for *non-decision* (ND) variables, which may occur as the antecedent of the Implication ($\rightarrow$) operator whereas *decision* variables cannot.

7 OPTIMIZATIONS

While the method presented so far has all the functionalities, we can make it produce a more compact MILP constraint using a DSL annotated with the type information.

7.1 Type Annotations for the DSL

While the DSL shown in Figure 3 is general enough, individual MILP solvers may have their own syntactic restrictions, e.g., over the *index* variables (ND) and *decision* variables. To speed up the synthesis using such information, we introduce a variant of the DSL, denoted $\mathbb{D}_t$, to capture user-provided syntactic restrictions. Our method combines $\mathbb{D}_t$, which is defined in Figure 13, with the DSL $\mathbb{D}$ defined in Figure 3, to take these type annotations into consideration.

Figure 3 captures the syntactic restrictions for both ND and *decision* variables while Figure 13 focuses exclusively on ND variables. In Figure 13, var_n represents ND variables, the values of which are decided by the dataset. In contrast, the values of decision variables are decided after the optimization is finished. For instance, in the *protein folding* example, i, j and k are ND variables while M and F are decision variables.

In Figure 13, each rule maps a type (left-hand side of ":=") to a set of compatible values (right-hand side of ":="). For instance, the compatible values for Boolean formula are $\mathbb{D}[\phi] = \{p, \phi \wedge \phi, \phi_n \rightarrow \phi\}$. Here ϕ_n represents the Boolean formula consisting of pure ND variables, which may be the antecedent of $\rightarrow$ operator, whereas a formula consisting of *decision* variable is prohibited.

7.2 Leveraging the Type Information

Next, we explain why the combined DSL allows the synthesis procedure to generate a more compact candidate ψ. The reason is because, by allowing more operations to be associated with ND variables, it may prevent some *unrealizability* cases from happening. As $\mathbb{D}_t$ relaxes the syntactic restrictions on ND variables, it enlarges the set of syntactically-valid formulas, which reduces the probability of being *unrealizable*. Hence, our synthesizer is able to reduce the newly-created variables and obtain a more compact solution.

Again, in the *protein folding* example, type information allows the synthesis method to generate the following candidate:

$$\underline{\neg E(i,j) \rightarrow \neg M(i,j)} \wedge \underline{E(i,j) \wedge S(i,j,k) \rightarrow M(i,j) \leq f(k)} \wedge \underline{\neg S(i,j,k1) \rightarrow f(k1) + M(i,j) \leq 1} \tag{17}$$

This is compact and does not rely on any new variables.

8 EXPERIMENTS

We have implemented our method in a software tool (SynMio), which uses Rosette [Torlak and Bodik 2013] to generate candidates in small bit-vector domain. We choose Rosette over other synthesis tools, such as cvc4sy [Reynolds et al. 2019] and EUSolver [Alur et al. 2017], since it directly supports uninterpreted functions as a syntactic construct for syntax-guided synthesis (SyGuS) [Alur et al. 2013]. Our verification subroutines, FalsifyBit and VerifyEq, are implemented using the Z3 SMT solver [Moura and Bjørner 2008]. The difference is that, while FalsifyBit uses the bit-vector (BV) domain, VerifyEq uses the linear integer/real arithmetic (LIA/LRA) domain.

Name	Description of the Problem	Name	Description of the Problem
O1	Protein Folding	O15	Continuous If2 NonStrict
O2	Decentralized Planning	O16	Continuous If Combine SN
O3	Protein Comparison	O17	Continuous If Combine NS
O4	Task Scheduling	O18	Continuous If Combine SS
O5	Food Manufacture	O19	Continuous If Combine NN
O6	ILOG CPLEX OR	Q1-11	StackOverflow/StackExchange
O7	Cardinality Problem	M1	Decision Tree Learning
O8	Minimize Num of Workers	M2	Ranking Function-AUC
O9	Traffic Scheduling	M3	Ranking Function-RRF
O10	Warehouse locating	M4	Sparse PCA
O11	Factory Production Planning	M5	Branch Constraints
O12	Continuous If1 Strict	M6	Bipartite Ranking
O13	Continuous If1 NonStrict	M7	Best Subset Selection
O14	Continuous If2 Strict	M8	Associate Classification

Fig. 14. The list of input specifications[1].

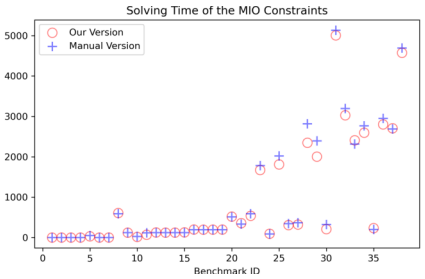


Fig. 15. The MILP-solving time.

We ran all of our experiments on a computer with 2.9 GHz Intel Core i5 CPU and 64 GB RAM. Our experiments were designed to answer the following questions:

- What is the quality of the MILP constraints synthesized by our method, measured in terms of both compactness and the MILP-solving time?
- How efficient is our method in synthesizing MILP constraints, measured in terms of both the size of the input specifications and the synthesis time?

8.1 Benchmarks

As shown in Fig. 14, our benchmarks include 38 mixed-integer optimization problems specified using a variety of Boolean logic operations. They come from statistics, machine learning, and data science applications. Broadly speaking, they fall into three categories. The first category ($O1$ to $O19$) are problems from transportation [Fourer 2014], financial services, supply network design [Belotti et al. 2011], and as well as biomedical research [Forrester and Greenberg 2008]. The second category ($Q1$ to $Q11$) are problems collected from various online Q&A platforms, including the *StackExchange* website. Due to difficulties in manually linearizing Boolean logic operations, novice data analysts frequently seek assistance from domain experts on these platforms. The third category ($M1$ to $M8$) are problems created by researchers who leverage MILP solvers in a variety of machine learning applications, including ranking [Chang 2012], decision tree learning [Bertsimas and Dunn 2017], and sparse PCA [Bertsimas et al. 2022].

All of these problems have the corresponding, manually-written MILP constraints. They come from three sources: online Q&A forums, textbooks, and specialists. These manually-written MILP constraints serve as a baseline for evaluating the quality of the solutions produced by our method.

8.2 Results: The Quality of Synthesized MILP Constraints

To evaluate the quality of the synthesized MILP constraints, we compared them with the MILP constraints manually written by domain experts. The results are shown in Table 1. The first four columns show, for each benchmark, the name, the size of the input specification ϕ, the number of variables in ϕ, and the size of the concrete dataset.

Since input specifications may be expressed symbolically using index variables, such as i, j, and k in the protein folding example, concrete datasets, such as $A = \{1, 2, \ldots, 50\}$ and $H = \{2, 4, \ldots\}$, must be used to concretize them before giving the flattened formula to MILP solvers. This is a common practice in the Liver Disorders [McDermott and Forsyth 2016], MicroMass [Mahe et al.

[1]Optimization benchmarks include O1-2, O4-5 [Williams 2013], O3, 8-9 [Fourer 2014], O6-7 [CPLEX 2015], O10-11 [Belotti et al. 2011]. Machine learning benchmarks include M1,5 [Bertsimas and Dunn 2017], M2-3 [Chang 2012] and M4 [Bertsimas et al. 2022].

Table 1. The quality of the synthesized MILP constraints in terms of compactness.

Name	#formula	#var	|C|	Synthesized		Manually-Written		Name	#formula	#var	|C|	Synthesized		Manually-Written	
				#formula	#var	#formula	#var					#formula	#var	#formula	#var
O1	3	6	197	4	8	3	6	Q1	2	4	1700	3	5	3	7
O2	1	6	69	3	6	3	6	Q2	2	5	1647	2	5	2	5
O3	1	6	99	3	6	3	6	Q3	2	4	1600	3	4	5	5
O4	1	4	96	2	6	4	6	Q4	21	13	3800	96	38	103	38
O5	5	11	90	20	20	50	40	Q5	1	4	160	1	4	1	4
O6	1	2	200	2	3	3	3	Q6	24	15	2500	101	25	101	25
O7	2	4	200	4	6	4	6	Q7	5	5	1100	21	11	7	9
O8	2	2	252	2	4	2	4	Q8	7	9	1300	19	13	14	14
O9	1	6	200	3	6	3	6	Q9	27	12	2600	107	20	125	29
O10	3	7	27	6	8	15	10	Q10	30	16	1880	134	22	144	27
O11	5	13	156	9	15	5	13	Q11	22	19	240	40	21	50	30
O12	1	4	200	2	5	2	5	M1	2	8	2.6M	5	10	5	10
O13	1	4	200	2	5	2	5	M2	1	5	5700	2	6	4	6
O14	1	4	200	2	5	2	5	M3	2	11	3366	4	11	4	11
O15	1	4	200	2	5	2	5	M4	2	6	5586	5	6	5	6
O16	2	5	280	4	7	4	7	M5	2	5	280	4	7	4	7
O17	2	5	280	4	7	4	7	M6	4	10	4200	8	14	10	15
O18	2	5	280	4	7	4	7	M7	5	9	4530	7	11	7	11
O19	2	5	280	4	7	4	7	M8	4	8	3702	6	9	8	11

2014], Adult [Kohavi et al. 1996] and Haberman Surv. datasets. We show $|C|$ in Column 4, which is the number of *pre-simplified decision variables* generated by `Gurobi` solver; $|C|$ is a frequently-used indicator for estimating the size of the search space. Here, *M1* is an outlier: it has an unusually-large value for $|C|$ because it takes the entire Adult [Kohavi et al. 1996] dataset as input, consisting of 48,842 concrete data elements.

The remainder of Table 1 compares the size of the synthesized MILP constraint with the size of the manually-written constraint. The size is measured in terms of both the number of subformulas in ψ and the number of variables in ψ.

8.2.1 Compactness. The results in Table 1 show that, overall, the synthesized constraints are as compact as the manually-written constraints. This is a significant achievement on its own, considering the amount of time and expertise needed to write the equivalent MILP constraints manually. Except for a few cases such as *Q7* and *Q8*, the synthesized constraints are either as compact as, or more compact than, the manually-written constraints. In some cases, the synthesized constraints are significantly more compact. For *Q9*, the number of subformulas is 14% smaller. For *O5*, in particular, the number of subformulas is less than half of the manually-written version.

8.2.2 MILP-solving Time. We also compared the synthesized constraint with the manually-written constraints in terms of the MILP-solving time. For consistency, we encoded all problems in Julia [Bezanson et al. 2017] and then solved them using the state-of-the-art `Gurobi` solver [Gurobi Optimization 2018]. The results are shown in Fig. 15, which represents the running time of the synthesized constraints using "∘", and the running time of the manually-written constraints using "+". Here, the x-axis is the benchmark index for *O1-O19*, *Q1-Q11*, and *M1-M8*, while the y-axis is the time in seconds.

The results show that, for more than 50% of the benchmarks, the synthesized constraint has almost the same MILP-solving time as the manually-written constraint. While some manually-written constraints have shorter MILP-solving time (e.g., *O8, M3*), the difference is less than 4%. For the remainder of the benmarks, the synthesized constraints have signficantly shorter MILP-solving time. For *Q3-4, Q6, Q9-10*, the synthesized constraints have much shorter MILP-solving time. Overall, the average performance improvement is more than 9.8%.

This improvement can be attributed in part to the ConvexHullSplit procedure, which splits variables to capture the correlation between subformulas in ϕ and thus significantly reduces the

ID	D	Synthesis Time				ID	D	Synthesis Time			
		None	Decomp	Falsify	Both			None	Decomp	Falsify	Both
O1	6	7689s	7689s	181s	181s	Q1	5	T/O	2293s	T/O	133s
O2	4	57s	37s	3s	3s	Q2	4	2517s	846s	95s	95s
O3	4	133s	53s	6s	6s	Q3	5	T/O	615s	112s	112s
O4	6	T/O	2409s	T/O	158s	Q4	6	T/O	T/O	T/O	4212s
O5	5	T/O	4785s	T/O	552s	Q5	3	174s	174s	3s	3s
O6	5	T/O	1836s	T/O	217s	Q6	6	T/O	T/O	T/O	5522s
O7	5	T/O	3439s	T/O	351s	Q7	7	T/O	7131s	T/O	651s
O8	3	T/O	2049s	T/O	37s	Q8	6	T/O	7845s	T/O	379s
O9	4	1893s	931s	10s	10s	Q9	6	T/O	T/O	T/O	4503s
O10	7	T/O	7309s	T/O	612s	Q10	6	T/O	T/O	T/O	4822s
O11	6	T/O	4968s	3727s	430s	Q11	4	T/O	T/O	T/O	1265s
O12	3	T/O	526s	T/O	3s	M1	7	T/O	4125s	T/O	373s
O13	3	T/O	519s	T/O	8s	M2	6	T/O	2323s	T/O	206s
O14	3	T/O	534s	T/O	9s	M3	7	T/O	3579s	826s	335s
O15	3	T/O	507s	T/O	9s	M4	4	T/O	2964s	685s	244s
O16	4	T/O	1207s	T/O	138s	M5	4	T/O	2554s	T/O	167s
O17	4	T/O	1225s	T/O	163s	M6	6	T/O	3213s	T/O	188s
O18	4	T/O	1246s	T/O	74s	M7	6	T/O	4062s	T/O	275s
O19	4	T/O	1288s	T/O	94s	M8	6	T/O	3491s	T/O	210s

Fig. 16. The performance of our synthesis tool. Here, T/O means > 3 hours.

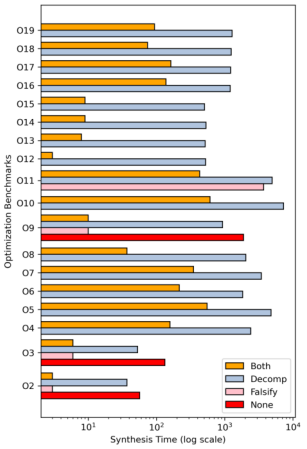


Fig. 17. The impact of each optimization technique.

relaxed solution space. Although ConvexHullSplit may add more variables, these variables are effective in constraining the solution space, allowing the MILP solver to converge faster. This is the case for *O11*, in particular: although the splitting has led to more subformulas (9 versus 5), the synthesized constraint reduces the MILP-solving time by more than 41%, (69.42s versus 118.17s).

8.3 Results: The Efficiency of the Synthesis Tool

To evaluate the efficiency of the synthesis tool, we analyzed its running time for all benchmarks. We also investigated the impact of two important components of our method, by comparing the performance with and without them. The results are shown in Fig. 16. Column 1 shows the benchmark name, and Column 2 shows the maximal depth of the intermediate AST (corresponding to a decomposed constraint ϕ_i). The next four columns show the running time of four variants of our synthesis tool. **None** uses neither EnrichDsl nor FalsifyBit subroutines; **Decomp** uses EnrichDsl but not FalsifyBit; **Falsify** uses FalsifyBit but not EnrichDsl; and **Both** uses both components.

The results in Fig. 16 show that our complete method (**Both**) is efficient in synthesizing MILP constraints and scalable for handling real-world specifications. For most of the benchmarks, it finishes quickly. For more than half of the benchmarks, it finishes within 5 minutes. For a few benchmarks, e.g., $Q4$ and $Q6$, it takes more than an hour; however, considering how labor-intensive and time-consuming it is for end users to write the MILP constraints manually, the time taken by the synthesis tool is acceptable. In terms of scalability, since the benchmark set consists of a diverse set of specifications collected from various sources, the fact that all of them can be handled by our tool means the tool is scalable enough in practice.

The results also demonstrate the impact of the two main algorithmic innovations in our method. Without **Decomp** and **Falsify**, the synthesis tool cannot finish most of the benchmarks within the time limit. Here, T/O stands for *timed out after 3 hours*. With either of them, while the performance improves significantly, there are still many benchmarks that cannot be finished within the time limit. However, it is clear that they have complementary strengths in speeding up the synthesis process. With both of them enabled, our tool is able to finish all benchmarks.

Table 2. Results for benchmarks that have no manually-written solutions.

Name	Description of the Problem	#formula	#var	|C|	Synthesized		MILP-solving Time	D	Synthesis Time			
					#formula	#var			None	Decomp	Falsify	Both
Q12	Implication Constraints (SO)	3	4	216	12	7	195s	5	T/O	3209s	T/O	212s
Q13	Implication of Int/Bool/Real (MathSE)	1	7	459	11	12	421s	6	T/O	4261s	T/O	359s
Q14	Chain of Implications (OR SE)	3	7	392	5	11	136s	5	T/O	4350s	T/O	297s
Q15	Disjunctive Expression (OR SE)	3	5	180	6	7	72s	5	T/O	2062s	T/O	183s
Q16	Disjunctive Expression in DAG (OR SE)	5	11	1940	21	17	1618s	7	T/O	6598s	T/O	589s
Q17	Chain of Disjunctions (OR SE)	1	6	270	3	6	186s	5	T/O	2795s	678.4s	241s
Q18	Exclusive Conditions (OR SE)	3	6	954	7	9	487s	6	T/O	2928s	T/O	263s
Q19	Robust Implication (OR SE)	3	2	1650	3	3	711s	4	T/O	1448s	T/O	125s
Q20	Union-closed Sets Conjecture (OR SE)	1	4	1800	3	4	846s	4	T/O	814s	132s	132s
Q21	Max in Constraints (OR SE)	1	4	2395	12	7	2024s	6	T/O	3850s	T/O	324s
Q22	Absolute in Constraints (OR SE)	1	3	476	2	5	315s	4	T/O	1266s	T/O	107s
Q23	Compact Continuous If (SO)	2	2	150	1	3	69s	4	T/O	1741s	T/O	143s

The two optimizations are also synergistic in that, when using both of them, the performance improvement is significantly more than the addition of what can be achieved by each in isolation. To illustrate this observation, we plot in Fig. 17 the running time of the four variants of our synthesis tool for a subset of the benchmarks, namely *O2-O19*. We omit *O1* since the numbers are too big to fit in the same scale as those of the other benchmarks. For `T/O` case, we plot zero as its value. First, let us focus on the blue (Decomp) and pink (Falsify) bars in Fig. 17: they show that **Decomp** is more effective than **Falsify** when each of them is used in isolation (the latter of which still corresponds to many T/O cases). Next, let us focus on the blue (Decomp) and orange (Both) bars: they show that, when **Falsify** is used together with **Decomp**, the reduction (from the blue bars to the orange bars) is drastic.

8.4 Results: Benchmarks without Manually-Written Solutions

In addition to the 38 benchmarks shown in Fig. 14, we also evaluated SynMio on 12 benchmarks for which manually-written solutions do not exist. In other words, answers to these 12 problems collected from various online forums are either empty, incomplete, incorrect or only applicable to a certain optimizer.

Table 2 shows the benchmark name and the type of logical constraints in Columns 1-2, where `SO`, `MathSE`, and `ORSE` stands for the online forums StackOverflow, MathStackExchange and OperationsResearchStackExchange, respectively.

Columns 3-7 are similar to Columns 2-6 of Table 1. They indicate that, for most benchmarks, new variables are needed in the synthesized constraints to linearize the input logical constraints. Furthermore, the synthesized constraint may have many more formulas than the input constraint (e.g., Q13 and Q21). The reason is that the input constraint has to be decomposed to many subconstraints, or the input constraint contains operations such as `max(x1,x2,x3)`. In order to remove the `max()` function, our method needs to introduce multiple variables `v1,v2,v3` and the corresponding predicates `v1`↔`(x1>x2)`, `v2`↔`max(x1,x2)` and `v3`↔`(max(x1,x2)>x3)`.

Columns 9-13 shows that our method is efficient and scalable in synthesizing MILP constraints for these benchmarks. They also demonstrate the effectiveness of our two main algorithmic innovations. Specifically, our complete method (**Both**) is the most efficient and, for more than 70% of the benchmarks, it is able to finish synthesis within 5 minutes.

9 RELATED WORK

There are two lines of prior work on converting Boolean logic operations to MILP constraints, which are the most closely related to ours. However, as we have already mentioned briefly, they both have severe limitations in the kinds of Boolean logic constraints that they can handle. In particular, Bertsimas et al. [Bertsimas et al. 2021; Bertsimas and Van Parys 2020] impose a regularization

term to their objective functions, with the goal of allowing Boolean logic relations; however, the method works only for a restricted subclass of problems and thus cannot linearize arbitrary Boolean expressions. The Big-M methods [Cococcioni and Fiaschi 2021; Fourer et al. 1987; Glover 1975] introduce a large number associated with each artificial variable, called M, to ensure that the Boolean logic constraint holds when the Boolean indicator variable is `False`. However, the M value has a significant and often unpredictable impact on the MILP-solving performance, which may lead to numerical instability. In contrast, our method does not have such problems since it linearizes the Boolean logic operations without using the M value at all.

Traditionally, MILP solvers and SMT solvers focus on problems in different niche applications. However, in recent years, there are efforts on bridging the gap. On the MILP solver side, there are efforts on extending the core MILP-solving algorithms to handle a limited number of Boolean logic operations natively. For example, IBM ILOG CPLEX [CPLEX 2015] automatically transforms logical constraints into equivalent linear formulations via automatic creation of new indicator variables. However, it explicitly branches on the indicator variable inside its solving procedure, without inferring the linearized formulation. As a result, this linearization only works for CPLEX and cannot be applied to other solvers that do not have such an inner solving algorithm to support the logical constraints. Gurobi [Bixby 2007] restricts the antecedent of implication operator to be a singular Boolean variable, as opposed to equality or inequality constraints. In contrast, our method can handle arbitrary combinations of Boolean logic operators. It is also a solver-independent way of transforming logical constraints to equivalent MILP constraints, before solving the MILP constraints using any MILP optimization tool.

On the SMT solver side, there are efforts on extending the SMT/Pseudo-Boolean solving paradigm to more efficiently solve subclassess of MILP problems [Devriendt et al. 2021; King et al. 2014; Nuzzo et al. 2010; Shoukry et al. 2018]. For example, Pseudo-Boolean solvers [Chai and Kuehlmann 2003; Devriendt et al. 2021] optimize 0-1 integer linear programs (ILP) by interleaving LP solving with conflict-driven pseudo-Boolean search. King et al. [King et al. 2014] integrate MIP solvers with SMT, to improve its optimization performance. However, these techniques focus exclusively on enhancing SMT solvers with optimization modules. None of them paid attention to the linearized formulas translated from Boolean constraints. Other works [Nuzzo et al. 2010; Shoukry et al. 2018] use a lazy combination of SMT solving and convex programming to determine the satisfiability of logic formulas over Boolean variables and convex constraints; however, they do not focus on optimizing the MILP problems.

While Mixed Boolean-Arithmetic (MBA) expressions [Feng et al. 2020; Liem et al. 2008; Liu et al. 2021; Shen and Ming 2021; Zhou et al. 2007] appear to be similar to MILP constraints, there are signficant differences. MBA expressions can directly combine arithmetic operations (in the integer modular ring $\mathbb{Z}/2^n$) with Boolean operations, but they focus on the discrete domain. In contrast, MILP contains arithmetic operations for both discrete and continuous values, but does not allow arbitrary bitwise Boolean operations. In practice, MBA-based techniques have been used primarily for obfuscation as well as deobfuscation [Blazytko et al. 2017; Shen and Ming 2021; Zhou et al. 2007], where the focus is on converting computations into MBA expressions, and vice versa, instead of supporting the optimization goals such as maximization/minimization in MILP.

Since our method relies on the popular SyGuS [Alur et al. 2013] framework for synthesizing MILP constraints, it is related to a large body of work on partitioning or pruning the search space and improving the efficiency of SyGuS [Alur et al. 2017; Eldib and Wang 2014; Eldib et al. 2016; Feng et al. 2018; Feser et al. 2015; Guo et al. 2019; Polikarpova et al. 2016; Reynolds et al. 2019; Wang et al. 2021, 2017]. Some of them rely on partitioning [Eldib and Wang 2014; Eldib et al. 2016] or utilize type-directed pruning techniques to avoid infeasible programs [Feser et al. 2015; Frankle et al. 2016; Guo et al. 2019; Osera and Zdancewic 2015; Polikarpova et al. 2016], while others leverage semantic

information of the DSL to check the feasibility of partial programs [Feng et al. 2018, 2017]. There is also work on using abstract interpretation to build the space of feasible programs [Wang et al. 2017]. However, the difference is that, while these prior works start from an infinitely large search space and gradually shrink it until the desired solution is obtained, our method starts from an under-approximated search space and we gradually enlarge it, which significantly speeds up the enumeration.

In the context of improving SyGuS, there are bottom-up enumeration techniques [Alur et al. 2017; Lee 2021] that recursively decompose a given large synthesis problem into smaller subproblems, and produce small program subexpressions via enumerative search. However, unlike our method that decomposes the input specification to make the synthesis problem *realizable*, they are not concerned with *realizability*; instead, they work on a sub-problem satisfying a subset of examples.

We have used existing program synthesis tools as part of our tool. In this context, EUsolver [Alur et al. 2017] and cvc4Syn [Reynolds et al. 2019] are the most closely related synthesis tools, as they both target SMT expressions. However, our approach goes far beyond, by supporting the uninterpreted function (UF) symbols as syntactic constructs during syntax-guided synthesis, whereas EUsolver and cvc4Syn are unable to manipulate UF during syntax-guided synthesis.

10 CONCLUSIONS

We have proposed a *solver-independent* and *generally-applicable* method for synthesizing *correct*, *efficient* and *robust* MILP constraints from a specification expressed using linear integer/real arithmetic constraints together with arbitrary combinations of Boolean logic operations. Starting from the input specification, our method first creates a domain specific language that is expressive enough, then uses syntax-guided synthesis (SyGuS) to assemble a candidate, and finally proves that the candidate is equivalent to the input specification. To improve performance, our method uses an under-approximation technique to quickly prune the search space, and uses an over-approximation technique to speed up equivalence verification. Our experiments on a diverse set of benchmarks show that the quality of the synthesized constraints are comparable to constraints written manually by domain experts. Furthermore, the synthesis tool is fast and scalable enough for handling real-world applications.

ACKNOWLEDGMENTS

We thank our shepherd Rastislav Bodik and the anonymous reviewers for their helpful feedback. This work was partially funded by the U.S. National Science Foundation grant CCF-2220345.

REFERENCES

Sina Aghaei, Mohammad Javad Azizi, and Phebe Vayanos. 2019. Learning optimal and fair decision trees for non-discriminative decision-making. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1418–1426. https://doi.org/10.1609/aaai.v33i01.33011418

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer Aided Design*. IEEE. https://doi.org/1721.1/90876

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Pietro Belotti, Leo Liberti, Andrea Lodi, Giacomo Nannicini, Andrea Tramontani, et al. 2011. Disjunctive inequalities: applications and extensions. *Wiley Encyclopedia of Operations Research and Management Science* 2 (2011), 1441–1450.

Dimitris Bertsimas, Ryan Cory-Wright, and Jean Pauphilet. 2021. A unified approach to mixed-integer optimization problems with logical constraints. *SIAM Journal on Optimization* 31, 3 (2021), 2340–2367. https://doi.org/1721.1/144075

Dimitris Bertsimas, Ryan Cory-Wright, and Jean Pauphilet. 2022. Solving Large-Scale Sparse PCA to Certifiable (Near) Optimality. *J. Mach. Learn. Res.* 23 (2022), 13–1.

Dimitris Bertsimas and Jack Dunn. 2017. Optimal classification trees. *Machine Learning* 106, 7 (2017), 1039–1082. https://doi.org/10.1007/s10994-017-5633-9

Dimitris Bertsimas and Bart Van Parys. 2020. Sparse high-dimensional regression: Exact scalable algorithms and phase transitions. *The Annals of Statistics* 48, 1 (2020), 300–323. https://doi.org/10.1214/18-AOS1804

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

Bob Bixby. 2007. The gurobi optimizer. *Transp. Re-search Part B* 41, 2 (2007), 159–178.

Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code.. In *USENIX Security Symposium*. 643–659. https://doi.org/10.5555/3241189.3241240

Donald Chai and Andreas Kuehlmann. 2003. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th annual Design Automation Conference*. 830–835. https://doi.org/10.1145/775832.776041

Allison Chang, Dimitris Bertsimas, and Cynthia Rudin. 2012. An integer optimization approach to associative classification. In *Advances in neural information processing systems*. 269–277.

Allison An Chang. 2012. *Integer optimization methods for machine learning*. Ph. D. Dissertation. Massachusetts Institute of Technology. https://doi.org/1721.1/72643

Marco Cococcioni and Lorenzo Fiaschi. 2021. The Big-M method with the numerical infinite M. *Optimization Letters* 15, 7 (2021), 2455–2468. https://doi.org/10.1007/s11590-020-01644-6

IBM ILOG CPLEX. 2015. V12. 6 User's Manual for CPLEX 2015. *CPLEX division* (2015).

Jo Devriendt, Ambros Gleixner, and Jakob Nordström. 2021. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints* 26, 1 (2021), 26–55. https://doi.org/10.1007/s10601-020-09318-x

Hassan Eldib and Chao Wang. 2014. Synthesis of Masking Countermeasures against Side Channel Attacks. In *International Conference on Computer Aided Verification*. Springer, 114–130. https://doi.org/10.1007/978-3-319-08867-9_8

Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *International Conference on Computer Aided Verification*. Springer, 343–363. https://doi.org/10.1007/978-3-319-41540-6_19

Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. 2020. Neureduce: Reducing mixed boolean-arithmetic expressions by recurrent neural network. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 635–644.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436. https://doi.org/10.1145/3062341.3062351

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239. https://doi.org/10.1145/2813885.2737977

Richard John Forrester and Harvey J Greenberg. 2008. Quadratic binary programming models in computational biology. *Algorithmic Operations Research* 3, 2 (2008).

Robert Fourer. 2014. Strategies for not linear optimization. In *5th INFORMS Optimization Society Conference*. 6–8March.

Robert Fourer, David M Gay, and Brian W Kernighan. 1987. *AMPL: A mathematical programming language*. AT & T Bell Laboratories Murray Hill, NJ.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. *ACM Sigplan Notices* 51, 1 (2016), 802–815. https://doi.org/10.1145/2914770.2837629

Fred Glover. 1975. Improved linear integer programming formulations of nonlinear integer problems. *Management science* 22, 4 (1975), 455–460. https://doi.org/10.1287/mnsc.22.4.455

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28. https://doi.org/10.1145/3371080

LLC Gurobi Optimization. 2018. Gurobi optimizer reference manual.

Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving unrealizability for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 335–352. https://doi.org/10.1007/978-3-030-25540-4_18

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1128–1142. https://doi.org/10.1145/3385412.3385979

Tim King, Clark Barrett, and Cesare Tinelli. 2014. Leveraging linear and mixed integer programming for SMT. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 139–146. https://doi.org/10.1109/FMCAD.2014.6987606

Ron Kohavi et al. 1996. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid.. In *Kdd*, Vol. 96. 202–207.

Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434335

Clifford Liem, Yuan Xiang Gu, and Harold Johnson. 2008. A compiler-based infrastructure for software-protection. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. 33–44. https://doi.org/10.1145/1375696.1375702

Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. Software Obfuscation with Non-Linear Mixed Boolean-Arithmetic Expressions. In *International Conference on Information and Communications Security*. Springer, 276–292. https://doi.org/10.1007/978-3-030-86890-1_16

Pierre Mahe, Maud Arsac, Sonia Chatellier, Valérie Monnin, Nadine Perrot, Sandrine Mailler, Victoria Girard, Mahendrasingh Ramjeet, Jérémy Surre, Bruno Lacroix, et al. 2014. Automatic identification of mixed bacterial species fingerprints in a MALDI-TOF mass-spectrum. *Bioinformatics* 30, 9 (2014), 1280–1286.

James McDermott and Richard S Forsyth. 2016. Diagnosing a disorder in a classification benchmark. *Pattern Recognition Letters* 73 (2016), 41–43. https://doi.org/10.1016/j.patrec.2016.01.004

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Pierluigi Nuzzo, Alberto Puggelli, Sanjit A Seshia, and Alberto Sangiovanni-Vincentelli. 2010. CalCS: SMT solving for non-linear convex constraints. In *Formal Methods in Computer Aided Design*. IEEE, 71–79.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630. https://doi.org/10.1145/2737924.2738007

Brandon Paulsen and Chao Wang. 2022a. Example Guided Synthesis of Linear Approximations for Neural Network Verification. In *International Conference on Computer Aided Verification*. Springer, 149–170. https://doi.org/10.1007/978-3-031-13185-1_8

Brandon Paulsen and Chao Wang. 2022b. LinSyn: Synthesizing Tight Linear Bounds for Arbitrary Neural Network Activation Functions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 357–376. https://doi.org/10.1007/978-3-030-99524-9_19

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538. https://doi.org/10.1145/2980983.2908093

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

Junfu Shen and Jiang Ming. 2021. Mba-blast: unveiling and simplifying mixed boolean-arithmetic obfuscation. (2021).

Yasser Shoukry, Pierluigi Nuzzo, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, George J Pappas, and Paulo Tabuada. 2018. SMC: Satisfiability modulo convex programming. *Proc. IEEE* 106, 9 (2018), 1655–1679. https://doi.org/10.1109/JPROC.2018.2849003

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 135–152. https://doi.org/10.1145/2509578.2509586

Jingbo Wang, Yannan Li, and Chao Wang. 2022. Synthesizing Fair Decision Trees via Iterative Constraint Solving. In *International Conference on Computer Aided Verification*. Springer, 364–385. https://doi.org/10.1007/978-3-031-13188-2_18

Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-Driven Synthesis of Provably Sound Side Channel Analyses. In *IEEE/ACM International Conference on Software Engineering*. IEEE, 810–822. https://doi.org/10.1109/ICSE43902.2021.00079

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://doi.org/10.1145/3158151

H Paul Williams. 2013. *Model building in mathematical programming*. John Wiley & Sons.

Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. 2007. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*. Springer, 61–75. https://doi.org/10.1007/978-3-540-77535-5_5

Received 2022-11-10; accepted 2023-03-31