Condition Synthesis Realizability via Constrained Horn Clauses

Bat-Chen Rothenberg, Orna Grumberg, Yakir Vizel, and Eytan Singher

Technion - Israel Institute of Technology
Haifa, Israel

Abstract. *Condition synthesis* takes a program in which some of the conditions in conditional branches are missing, and a specification, and automatically infers conditions to fill-in the holes such that the program meets the specification.
In this paper, we propose CoSyn, an algorithm for determining the realizability of a condition synthesis problem, with an emphasis on proving unrealizability efficiently. We use the novel concept of a *doomed* initial state, which is an initial state that can reach an error state along *every* run of the program. For a doomed initial state σ, there is no way to make the program safe by forcing σ (via conditions) to follow one computation or another. CoSyn checks for the existence of a doomed initial state via a reduction to Constrained Horn Clauses (CHC).
We implemented CoSyn in SeaHorn using Spacer as the CHC solver and evaluated it on multiple examples. Our evaluation shows that CoSyn outperforms the state-of-the-art syntax-guided tool Cvc5 in proving both realizability and unrealizability. We also show that joining forces of CoSyn and Cvc5 outperforms Cvc5 alone, allowing to solve more instances, faster.

1 Introduction

The automated synthesis of imperative programs from specifications is a very fruitful research area [26,1,27,18,9,28,22,25,16]. Our paper focuses on the important subproblem of condition synthesis. Condition synthesis receives as input a partial program, where conditions are missing in conditional branches (e.g., `if` statements), and a specification. A solution to this problem is a set of conditions to fill-in the holes such that the resulting program meets the specification. If such a solution exists, the problem is *realizable*, otherwise it is *unrealizable*.

The main motivation for condition synthesis is automated program repair. The problem naturally arises whenever the source of a bug is believed to be in a conditional branch, and the condition has to be replaced for the program to be correct. Studies on repair have shown that many real-life bugs indeed occur due to faulty conditions [29]. Several program repair methods focus on condition synthesis [19,6,29]. These algorithms, however, do not guarantee formal verification of the resultant program, but only that it passes a certain set of tests used as a specification.

In this work, we propose CoSyn, a novel algorithm for determining the realizability of a condition synthesis problem, with an emphasis on proving unrealizability efficiently. We use a formal safety specification and conduct a search guided by semantics rather than syntax. Importantly, CoSyn's (un)realizability results are accompanied by an *evidence* to explain them.

Our semantics-guided search is based on the novel concept of *doomed* initial states. An initial state is called *doomed* if it eventually reaches an error state along *every* run of the program. For a doomed initial state σ, there is no way to make the program safe by forcing σ (via conditions) to follow one computation or another. It will lead to a failure anyway. Thus, the existence of such a state constitutes a proof that conditions cannot be synthesized at all, regardless of syntax.

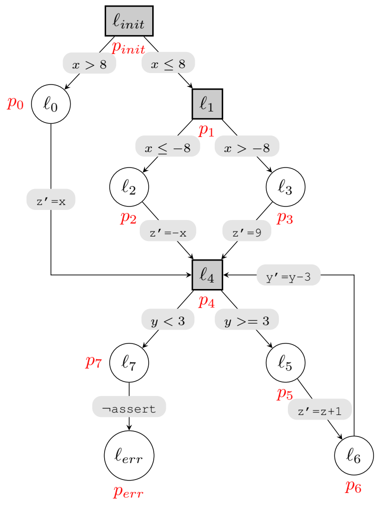
To check for the existence of a doomed initial state, CoSyn uses a reduction to Constrained Horn Clauses (CHC). CHC is a fragment of First-Order Logic, associated with effective solvers [4]. Our reduction constructs a set of CHCs that are satisfiable iff the condition synthesis problem is realizable, and utilizes a CHC solver to solve them.

When CoSyn finds a problem unrealizable, its answer is accompanied by a *witness*: an initial doomed state. When it finds a problem realizable, it returns a *realizability proof*. A realizability proof consists of two parts: a constraint defining a range of conditions *for each hole* in the program, and a correctness certificate. The range of conditions for a hole in location l is defined using two logical predicates, $\Psi^f(l)$ and $\Psi^t(l)$. Every predicate $\Psi(l)$ for which the implication $\Psi^f(l) \implies \Psi(l) \implies \Psi^t(l)$ holds (in particular $\Psi^f(l)$ and $\Psi^t(l)$), is a valid solution for the hole in location l. Moreover, the certificate is a proof for the safety of the program when using $\Psi(l)$ as a solution.

An important feature of CoSyn is that it can complement existing synthesis algorithms such as syntax-guided-synthesis (SyGuS) [1]. SyGuS limits the search-space to a user-defined grammar, hence ensuring that if a solution is found, it is of a user-desired shape. However, if a SyGuS algorithm determines the problem is unrealizable, it is with respect to the given grammar. Instead, one can use CoSyn to determine if the problem is realizable or not. In the case that the problem is realizable, the implication $\Psi^f(l) \implies \Psi(l) \implies \Psi^t(l)$, and a grammar can be given to a SyGuS algorithm, which then synthesizes a solution that conforms with the given grammar. Note that the input problem to the SyGuS algorithm is now much simpler (as evident in our experimental evaluation). This strategy can assist users as it can indicate if a solution exists at all, or if debugging of the specification is required (when unrealizable). Moreover, it can reduce the burden from an iterative synthesis process that searches for a solution in the presence of increasingly many examples or increasingly complex grammars. This is achieved by detecting unrealizability up-front.

We implemented CoSyn in an open-source tool on top of SeaHorn, a program verification tool for C programs. We created a collection of 125 condition synthesis problems by removing conditions from verification tasks in the TCAS and SVCOMP collections and by implementing several introductory programming assignments with missing conditions. We conducted an empirical evaluation of CoSyn against the state-of-the-art SyGuS engine implemented in Cvc5 on our benchmark collection. Two different variants were compared. In the first, we compare CoSyn and Cvc5 without a grammar. In the second, a grammar was supplied, and we compare the performance of CoSyn in conjunction with Cvc5[1] against Cvc5 alone. The experiments show that in both variations, with and without grammar, CoSyn solves more instances, both realizable and unrealizable. The advantage of CoSyn is most noticeable on the unrealizable

[1] Where CoSyn is executed, and Cvc5 is then invoked on the given grammar and implication $\Psi^f(l) \implies \Psi(l) \implies \Psi^t(l)$.



(a) $\mathcal{G}_{ex1}$

$$\top \rightarrow p_{init}$$
$$p_{init} \wedge (x > 8) \wedge (X' = X) \rightarrow p'_0$$
$$p_{init} \wedge (x \leq 8) \wedge (X' = X) \rightarrow p'_1$$
$$p_0 \wedge (z' = x) \wedge (x' = x) \wedge (y' = y) \rightarrow p'_4$$
$$p_1 \wedge (x <= -8) \wedge (X' = X) \rightarrow p'_2$$
$$p_1 \wedge (x > -8) \wedge (X' = X) \rightarrow p'_3$$
$$p_2 \wedge (z' = -x) \wedge (x' = x) \wedge (y' = y) \rightarrow p'_4$$
$$p_3 \wedge (z' = 9) \wedge (x' = x) \wedge (y' = y) \rightarrow p'_4$$
$$p_4 \wedge (y \geq 3) \wedge (X' = X) \rightarrow p'_5$$
$$p_4 \wedge (y < 3) \wedge (X' = X) \rightarrow p'_7$$
$$p_5 \wedge (z' = z + 1) \wedge (x' = x) \wedge (y' = y) \rightarrow p'_6$$
$$p_6 \wedge (y' = y - 3) \wedge (z' = z) \wedge (x' = x) \rightarrow p'_4$$
$$p_7 \wedge (\neg(z \geq 9 \wedge z \geq x \wedge z \geq -x)) \wedge (X' = X) \rightarrow p'_{err}$$
$$p_{err} \rightarrow \bot$$

(b) $\mathit{\Pi}_{\mathcal{G}_{ex1}}$

Fig. 1: The CFG $\mathcal{G}_{ex1}$ (left) and the set of CHCs $\mathit{\Pi}_{\mathcal{G}_{ex1}}$ (right).

problems. Further, CoSyn, in both variants, performs better w.r.t. runtime. This leads us to conclude that CoSyn can be an important addition to existing SyGuS tools.

To summarize, the main contributions of our work are:

- A novel algorithm, called CoSyn, for solving the (un)realizability problem of condition synthesis via a non-standard reduction to Constrained Horn Clauses (CHC). To the best of our knowledge, CoSyn is the first algorithm to determine that a condition synthesis problem is unrealizable w.r.t. any grammar.
- CoSyn's results are supported by an *evidence*: either a doomed initial state (for an unrealizable problem), or a realizability proof (for a realizable problem) that can be used by a synthesis tool to generate a solution w.r.t. a given grammar.

2 Preliminaries

2.1 Program Safety

To represent programs, we use control-flow-graphs with transitions encoded as logical formulas. We consider First Order Logic modulo a theory $\mathcal{T}$ and denote it by $FOL(\mathcal{T})$. $\mathcal{T}$ is defined over signature $\Sigma_{\mathcal{T}}$. We denote by X a set of variables representing program variables. A valuation σ of X is called a *program state*. We use the set $X^i = \{x^i \mid x \in X\}$ to represent variable values after i computation steps, where $i \geq 1$. For the special case of $i = 1$ (one computation step) we also use the set $X' = \{x' \mid x \in X\}$. A *state formula* is a (quantifier-free) formula in $FOL(\mathcal{T})$ defined over the signature $\Sigma_{\mathcal{T}} \cup X$. A *transition formula* is a (quantifier-free) formula in $FOL(\mathcal{T})$ defined over the signature $\Sigma_{\mathcal{T}} \cup X \cup X'$.

A *control-flow-graph (CFG)* is a tuple $\mathcal{G} = (\Lambda, \Delta, l_{init}, l_{err}, \Lambda_{cond})$, where Λ is a finite set of program locations, Δ is a set of transitions, $l_{init} \in \Lambda$ is the initial location and $l_{err} \in \Lambda$ is the error location. A *transition* τ is a triple $\langle l, \varphi, m \rangle$, where $l, m \in \Lambda$ are respectively the entry and exit locations of the transition, and φ is a transition formula. The set $\Lambda_{cond} \subset \Lambda$ is a set of locations, called *condition locations*, each of which having exactly two outgoing transitions in Δ, representing a condition and its negation. Formally, for every condition location $l_c \in \Lambda_{cond}$, there exist two distinct locations $l_c^f, l_c^t \in \Lambda$ and a state formula θ_c such that the only two outgoing transitions from l_c in Δ are $\langle l_c, \theta_c \wedge (X' = X), l_c^t \rangle$ and $\langle l_c, \neg\theta_c \wedge (X' = X), l_c^f \rangle$ (where the notation $X' = X$ is short for the conjunction of equalities between each variable and its primed version). A *path* π in the CFG is a sequence of transitions from Δ of the form

$$\pi = \langle l_0, \varphi_0, l_1 \rangle \langle l_1, \varphi_1, l_2 \rangle \langle l_2, \varphi_2, l_3 \rangle \cdots$$

The path is an *error path* if it is finite and, in addition, $l_0 = l_{init}$ and $l_n = l_{err}$ for some $n \geq 0$. Let α_π be a sequence of formulas representing π. That is,

$$\alpha_\pi = \varphi_0(X^0, X^1), \varphi_1(X^1, X^2), \varphi_2(X^2, X^3) \cdots$$

A *run* along path π from state σ is a sequence of states $r = \sigma_0, \sigma_1, \sigma_2 \ldots$, where $\sigma = \sigma_0$ and for every $i \geq 0$, σ_i is a valuation of variables X^i, such that $(\sigma_i, \sigma_{i+1}) \models \varphi_i(X^i, X^{i+1})$. In that case, we say that r starts at l_0. Path π is *feasible* if there is a run along it. If a run $r = \sigma_0, \sigma_1, \sigma_2 \ldots$ along π starts at l_{init} (i.e., $l_0 = l_{init}$) then for every $i \geq 0$ we say that state σ_i is *reachable* at l_i.

A *safety verification problem* is to decide whether a CFG $\mathcal{G}$ is SAFE or UNSAFE. $\mathcal{G}$ is UNSAFE if there exists a feasible error path in $\mathcal{G}$. Otherwise, it is SAFE.

Example 1. The CFG $\mathcal{G}_{ex1}$ is presented in Figure 1(a), where $\Lambda_{cond} = \{\ell_{init}, \ell_1\, \ell_4\}$. The Assertion at l_7 is $(z \geq 9 \wedge z \geq x \wedge z \geq -x)$. The path $\pi = \langle l_{init}, x \leq 8, l_1 \rangle \langle l_1, x \leq -8, l_2 \rangle \langle l_2, z' = -x, l_4 \rangle \langle l_4, y < 3, l_7 \rangle \langle l_7, \neg assert, l_{err} \rangle$ is a feasible error path in $\mathcal{G}_{ex1}$: there is a run along π from state σ, where $\sigma(x) = -8$ and $\sigma(y) = \sigma(z) = 0$. Consequently, the CFG $\mathcal{G}_{ex1}$ is UNSAFE.

2.2 Constrained Horn Clauses

Given the sets $\mathcal{F}$ of function symbols, $\mathcal{P}$ of uninterpreted predicate symbols, and $\mathcal{V}$ of variables, a *Constrained Horn Clause (CHC)* is a First Order Logic (FOL) formula of the form:

$$\forall \mathcal{V} \cdot (\phi \wedge p_1(X_1) \wedge \cdots \wedge p_k(X_k) \rightarrow h(X)), \text{ for } k \geq 1$$

where: ϕ is a constraint over $\mathcal{F}$ and $\mathcal{V}$ with respect to some background theory $\mathcal{T}$; $X_i, X \subseteq \mathcal{V}$ are (possibly empty) vectors of variables; $p_i(X_i)$ is an application $p(t_1, \ldots, t_n)$ of an n-ary predicate symbol $p \in \mathcal{P}$ for first-order terms t_i constructed from $\mathcal{F}$ and X_i; and $h(X)$ is either defined analogously to p_i or is $\mathcal{P}$-free (i.e., no $\mathcal{P}$ symbols occur in h). Here, h is called the *head* of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is called the *body*. A clause is called a *query* if its head is $\mathcal{P}$-free, and otherwise, it is called a *rule*. A rule with body $\mathtt{true}$ is called a *fact*. We say a clause is *linear* if its body contains at most one predicate symbol, otherwise, it is called *non-linear*. For convenience, given a CHC C of the form $\phi \land p_1(X_1) \land \cdots \land p_k(X_k) \rightarrow h(X))$, we will use $head(C)$ to denote its head $h(X)$. We refrain from explicitly adding the universal quantifier when the set of variables is clear from the context.

A set Π of CHCs is *satisfiable* iff there exists an interpretation $\mathcal{I}$ such that all clauses in Π are valid under $\mathcal{I}$. For $p \in \mathcal{P}$ we denote by $\mathcal{I}[p]$ the interpretation of p in $\mathcal{I}$.

2.3 Program Safety as CHC Satisfiability

Given a CFG $\mathcal{G} = (\Lambda, \Delta, l_{init}, l_{err}, \Lambda_{cond})$, checking its safety can be reduced to checking the satisfiability of a set $\Pi_{\mathcal{G}}$ of CHCs [4], as described below. For each program location $l \in \Lambda$, define an uninterpreted predicate symbol p_l. $\Pi_{\mathcal{G}}$ is then defined as the set of the following CHCs:

1. $\top \rightarrow p_{init}(X)$
2. $p_l(X) \land \varphi \rightarrow p_m(X')$ for every $\langle l, \varphi, m\rangle \in \Delta$
3. $p_{err}(X) \rightarrow \bot$

Note that this formulation assumes there are no function calls in the CFG, and that all function calls in the original program are inlined. This also implies that the resulting CHCs are linear. When clear from the context, we omit X and X' from $p_l(X)$, $p_l(X')$ and $\varphi(X, X')$. Instead, we write p_l, p'_l and φ, respectively.

Example 2. Consider again the CFG $\mathcal{G}_{ex1}$, presented in Figure 1(a), and its corresponding set of CHCs, $\Pi_{\mathcal{G}_{ex1}}$, given in Figure 1(b). For brevity, we write p_i as short for $p_i(x, y)$ and p'_i as short for $p_i(x', y')$. As shown in Example 1, $\mathcal{G}_{ex1}$ is UNSAFE and therefore there is no satisfying interpretation for its predicate symbols.

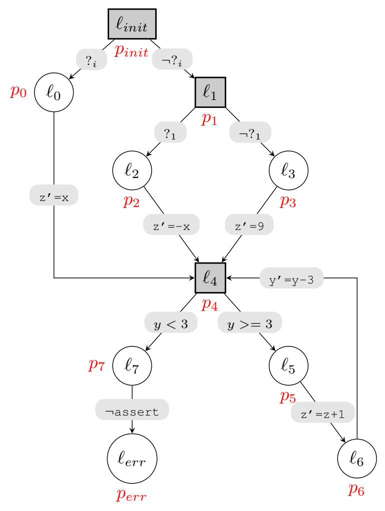
Lemma 1. *Let p_i be the predicate symbol associated with location l_i in a CFG $\mathcal{G}$. Assume that $\Pi_{\mathcal{G}}$ is satisfiable by the interpretation $\mathcal{I}$. Then, the interpreted predicate $\mathcal{I}[p_i]$ has the property that for every state σ, if σ is reachable at l_i (from l_{init}), then $\sigma \models \mathcal{I}[p_i]$.*

3 From Realizability to CHC Satisfaibility

In this section we describe the synthesis problem we solve, named *condition synthesis*. We also show how realizability of this problem can be reduced to satisfiability of a set of CHCs. From this point on, we assume that all function calls in the original program are inlined. This implies that the set of CHCs representing the program's CFG contains only linear clauses.

3.1 Defining the Condition Synthesis Problem

Given a set of condition locations specified by the user, the goal of condition synthesis is to automatically find conditions to be placed in these locations so that the program becomes correct. We start by formally defining a program in which the conditions at some of the condition locations are missing. Intuitively, such a location imposes no constraint on the continuation of the program execution at that location. Hence, the resulting program behaves non-deterministically.



(a) $\mathcal{G}_{\text{nd}}$

$$
\begin{gathered}
p_{init} \to \bot \\
p_0' \land p_1' \land (X' = X) \to p_{init} \\
p_2' \land p_3' \land (X' = X) \to p_1 \\
p_4' \land (z' = x) \land (x' = x) \land (y' = y) \to p_0 \\
p_4' \land (z' = -x) \land (x' = x) \land (y' = y) \to p_2 \\
p_4' \land (z' = 9) \land (x' = x) \land (y' = y) \to p_3 \\
p_5' \land (y \geq 3) \land (X' = X) \to p_4 \\
p_7' \land (y < 3) \land (X' = X) \to p_4 \\
p_6' \land (z' = z + 1) \land (x' = x) \land (y' = y) \to p_5 \\
p_4' \land (y' = y - 3) \land (z' = z) \land (x' = x) \to p_6 \\
p_{err}' \land (\neg(z >= 9 \land z >= x \land z >= -x)) \land (X' = X) \to p_7 \\
\top \to p_{err}
\end{gathered}
$$

(b) $\Pi_{\mathcal{S}}$

Fig. 2: The non-deterministic CFG $\mathcal{G}_{\text{nd}}$ with two non-deterministic nodes $\{l_{init}, l_1\}$ (left) and the set of CHCs $\Pi_{\mathcal{S}}$ (right).

Definition 1. *Let $\mathcal{G}$ be a CFG. A condition location $l_c \in \Lambda_{cond}$ is called* non-deterministic *if the two outgoing transitions from l_c have the following form: $\langle l_c, X' = X, l_c^t \rangle$ and $\langle l_c, X' = X, l_c^f \rangle$. If $\mathcal{G}$ has a non-deterministic condition location, we say that $\mathcal{G}$ is* non-deterministic.

Example 3. The left-hand-side of Figure 2 presents the non-deterministic CFG $\mathcal{G}_{\text{nd}}$, which is identical to the CFG $\mathcal{G}_{ex1}$ of Figure 1, except that locations $\{l_{init}, l_1\}$ are non-deterministic. The transitions leaving those locations are labeled with expressions of the form $?$ or $\neg ?$, to indicate that no condition is associated with these locations.

A non-deterministic CFG $\mathcal{G}$ can be transformed into a deterministic CFG by replacing every non-deterministic control location with a deterministic condition[2]. More formally,

Definition 2. *Let $\mathcal{G} = (\Lambda, \Delta, l_{init}, l_{err}, \Lambda_{cond})$ be a non-deterministic CFG and $\Lambda_{cond}^? \subseteq \Lambda_{cond}$ be the set of non-deterministic control locations. Let $\Psi : \Lambda_{cond}^? \to \Gamma$ be a function where for every $l_s \in \Lambda_{cond}^?$, $\Psi(l_s) \in \Gamma$ is a predicate over the set of program variables. Ψ is called a resolving function. The resolved CFG $\mathcal{G}_\Psi = (\Lambda, \Delta_\Psi, l_{init}, l_{err}, \Lambda_{cond})$, is defined as follows.*

– *For $l \in (\Lambda \setminus \Lambda_{cond}^?)$ and for a formula φ and $m \in \Lambda$. If $\langle l, \varphi, m \rangle \in \Delta$, then $\langle l, \varphi, m \rangle \in \Delta_\Psi$*

[2] We emphasize that a deterministic CFG can still contain non-deterministic assignments. In the context of CFG, non-determinisim only refers to the form/structure of the CFG.

– *For $l_s \in \Lambda^?_{cond}$, the only two transitions out of l_s in Δ_Ψ are* $\langle l_s, \Psi(l_s) \wedge (X = X'), l_s^t \rangle$ *and* $\langle l_s, \neg\Psi(l_s) \wedge (X = X'), l_s^f \rangle$

We define the *synthesis problem* as $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$, where $\mathcal{G}$ is a non-deterministic CFG and $\Lambda^?_{cond}$ is the set of non-deterministic condition locations. A *solution* to $\mathcal{S}$ is a resolving function $\Psi : \Lambda^?_{cond} \to \Gamma$ such that $\mathcal{G}_\Psi$ is deterministic and SAFE.

3.2 Reducing Condition Synthesis Realizability to CHC Satisfiability

A realizability problem is to determine whether a given synthesis problem $\mathcal{S}$ has a solution or not. In this section we show how the problem of condition synthesis is reducible to the CHC satisfiability problem. In what follows we refer to realizability w.r.t. the condition synthesis problem given by $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$, where $\mathcal{G}$ is non-deterministic and $\Lambda^?_{cond}$ is the set of non-deterministic control locations.

Doomed States To explain the reduction of realizability to CHC, we first introduce the notion of *doomed states*.

Definition 3. *A state σ is doomed at location l_i if every run from σ, starting at l_i, reaches the error location l_{err}.*

Note that, in particular, all runs from a state that is doomed at l_i are *finite*.

Intuitively, given a synthesis problem $\mathcal{S}$, if there exists a doomed state at location l_{init}, then $\mathcal{S}$ is *unrealizable*. Recall that $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$, and $\mathcal{G}$ is non-deterministic. Hence, if an initial state σ is doomed, then no matter which conditions are chosen for the non-deterministic control locations in $\mathcal{G}$, σ can reach the error location along every run. We exploit this observation to reduce the (un)realizability problem to identifying initial doomed states in a non-deterministic CFG, or proving their absence.

Example 4. Consider again the non-deterministic CFG $\mathcal{G}_{\mathrm{nd}}$, presented in Figure 2. The realizability problem in this case is to determine whether the synthesis problem $\mathcal{S} = (\mathcal{G}_{\mathrm{nd}}, \{l_{init}, l_1\})$ has a solution or not.

Note that a state σ in which $\sigma(x) = 10$ and $\sigma(y) = 0$ is doomed at location l_1: All runs from this state starting at l_1 end up in l_{err}. In contrast, no state is doomed at the initial location l_{init}. That is, from any such state it is possible to find a run that does not proceed to l_{err}. As we will see later, this implies that the synthesis problem has a solution – we can assign conditions to l_{init} and l_1 s.t. the resulting program is SAFE.

Realizability to CHC Finding the set of states that can reach l_{err} along some run from a given location $l \in \mathcal{G}$ can be achieved by iteratively computing the pre-image of bad states, starting from l_{err} up to the location l. Note that if there exists a condition location on paths from l to l_{err}, then the union of the pre-image along the "then" and "else" branches is computed.

In order to find doomed states, however, a non-deterministic condition location should be handled differently. Whenever the pre-image computation reaches a non-deterministic condition location l_s, the pre-image computed along the "then" and "else" branches need to be *intersected*. The result of this intersection is a set of states that reach l_{err} along *every* run that starts in l_s.

In what follows we describe how to construct a set of CHCs that captures doomed states. The construction is based on a transformation from the original set of CHCs $\Pi_{\mathcal{G}}$, and has two phases, described below. Due to lack of space, all proofs in the following sections are deferred to the full version.

Reversed CHC Assume that for $\mathcal{S} = (\mathcal{G}, \Lambda^{?}_{cond})$, $\Pi_{\mathcal{G}}$ is a set of CHCs originating from $\mathcal{G}$ using the procedure presented in section 2.3. As described above, computing doomed states requires computing the pre-image of states that can reach l_{err}. Hence, the first step of our reduction is to construct a new set of CHCs $\Pi^R_{\mathcal{G}}$, referred to as the *reverse* of $\Pi_{\mathcal{G}}$. As the name implies, $\Pi^R_{\mathcal{G}}$ is obtained by reversing the polarity of uninterpreted predicates in every clause. More precisely, a predicate that appears positively appears negatively in the reversed clause, and vice-versa. For example, if $p(X) \wedge \varphi(X, X') \to q(X')$ is a clause in $\Pi_{\mathcal{G}}$, then $q(X') \wedge \varphi(X, X') \to p(X)$, is a clause in $\Pi^R_{\mathcal{G}}$. Reversing a set of CHCs is performed using simple syntactic rules. We emphasize that this transformation is only applicable for linear CHCs. Reversing a non-linear CHC results in a clause that is not in Horn form.

Note that for a transition $\langle l, \varphi, m \rangle$, the clause $p_l(X) \wedge \varphi(X, X') \to p_m(X')$ captures the *image* operation. Namely, a given set of states in location l and their set of successors in location m satisfy the clause. Similarly, the reversed clause $p_m(X') \wedge \varphi(X, X') \to p_l(X)$ captures the *pre-image* operation. Meaning, a given set of states in location m and their predecessors at location l satisfy the reversed clause.

Theorem 1. *For every CFG $\mathcal{G}$, $\Pi_{\mathcal{G}}$ is satisfiable iff $\Pi^R_{\mathcal{G}}$ is satisfiable.*

Proof (sketch). Let $\mathcal{I}$ be an interpretation that satisfies $\Pi_{\mathcal{G}}$. Then, $\mathcal{I}^R[p_l] = \neg\mathcal{I}[p_l]$ for every location $l \in \Lambda$ is a satisfying interpretation for $\Pi^R_{\mathcal{G}}$. In the other direction, define $\mathcal{I}[p_l] = \neg\mathcal{I}^R[p_l]$, which satisfies $\Pi_{\mathcal{G}}$.

Lemma 2. *Let p_i be the predicate symbol associated with label l_i in the CFG $\mathcal{G}$. Assume that the reverse of $\Pi_{\mathcal{G}}$, $\Pi^R_{\mathcal{G}}$, is satisfiable by the interpretation $\mathcal{I}^R$. Then, for every state σ, if σ is a start of a run along a path from l_i to l_{err}, then $\sigma \models \mathcal{I}^R[p_i]$.*

Doomed States in Reversed CHCs Reversing the set of CHCs allows us to capture the pre-image of l_{err}. This, as noted, is only the first step. Recall that in order to identify doomed states, whenever a non-deterministic condition location is reached, the pre-image of the "then" branch must be intersected with the pre-image of the "else" branch.

For a given non-deterministic condition location $l_s \in \Lambda^{?}_{cond}$, the reversed set of CHCs, $\Pi^R_{\mathcal{G}}$, contains the following two clauses:

$$p^t_s(X') \wedge (X = X') \to p_s(X) \quad \text{and} \quad p^f_s(X') \wedge (X = X') \to p_s(X),$$

where p^t_s and p^f_s represent the pre-image of the "then" branch and "else" branch, respectively. As described above, the intersection of the pre-image along the "then" and "else" branches represents the doomed states. In order to represent this intersection, the second phase of the transformation replaces every two such clauses with the clause:

$$p_s^t(X') \wedge p_s^f(X') \wedge (X = X') \rightarrow p_s(X).$$

We denote the resulting set of CHCs as $\Pi_{\mathcal{S}}$.

Example 5. Consider again the non-deterministic CFG $\mathcal{G}_{\mathrm{nd}}$, presented in Figure 2. The right-hand-side of the figure presents the set $\Pi_{\mathcal{S}}$ of CHCs, which capture the doomed states in the control locations of $\mathcal{G}_{\mathrm{nd}}$. Assume $\mathcal{I}^{\mathcal{S}}$ is a satisfying interpretation for $\Pi_{\mathcal{S}}$. Since $\mathcal{I}^{\mathcal{S}}$ satisfies the clause $p_{init} \rightarrow \perp$ in $\Pi_{\mathcal{S}}$, then necessarily $\mathcal{I}^{\mathcal{S}}[p_{init}] = \perp$, which means that no initial state of $\mathcal{G}_{\mathrm{nd}}$ is doomed. As proved later, this guarantees that the synthesis problem $\mathcal{S} = (\mathcal{G}_{\mathrm{nd}}, \{l_{init}, l_1\})$ is realizable.

Lemma 3. *Let p_i be the predicate symbol associated with location l_i in the CFG $\mathcal{G}$. Let $\mathcal{I}^{\mathcal{S}}$ be an interpretation satisfying $\Pi_{\mathcal{S}}$ of $\mathcal{G}$. Then, for every state σ, if it is doomed at location l_i, then $\sigma \models \mathcal{I}^{\mathcal{S}}[p_i]$.*

The following theorem states that the satisfiability of $\Pi_{\mathcal{S}}$ determines the realizability of $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$. In fact, given a satisfying interpretation for $\Pi_{\mathcal{S}}$, it is possible to construct solutions to the synthesis problem $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$. Further, if $\Pi_{\mathcal{S}}$ is *not* satisfiable, then the synthesis problem is unrealizable.

Theorem 2. *$\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$ is realizable iff $\Pi_{\mathcal{S}}$ is satisfiable.*

We partition the proof of the theorem into two. Below we present the first direction. In Section 4 we prove the second direction of the theorem.

Lemma 4. *If $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$ is realizable then $\Pi_{\mathcal{S}}$ is satisfiable.*

4 Realizability and the Satisfying Interpretation of $\Pi_{\mathcal{S}}$

In this section we first show that if $\Pi_{\mathcal{S}}$ is satisfiable, then *there exists* a solution to the realizability problem. Later in Section 4.2, we show how such a solution, i.e. a resolving function, can be constructed. By that, we also prove the other direction of Theorem 2.

Lemma 5. *If $\Pi_{\mathcal{S}}$ is satisfiable, then there exists a resolving function Ψ that solves $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$. That is, $\mathcal{G}_\Psi$ is SAFE.*

The above lemma implies that in the case where $\Pi_{\mathcal{S}}$ is satisfiable, then $\mathcal{S}$ is realizable. Before describing how the resolving function is constructed, we develop both the intuition and the needed technical material in the following section.

4.1 The Role of The Resolving Function

Let $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$ be a synthesis problem such that $\Pi_\mathcal{S}$ is satisfiable, and $\mathcal{I}^\mathcal{S}$ is its satisfying interpretation. We wish to find a solution Ψ of $\mathcal{S}$.

Recall that for a location $l_i \in \Lambda$ and its associated predicate $p_i \in \Pi_\mathcal{S}$, $\mathcal{I}^\mathcal{S}[p_i]$ is an over-approximation of states that are doomed at l_i (Lemma 3). Clearly, if a synthesized program has a reachable state that is also doomed, then the program is not SAFE. Hence, the goal is to synthesize a program where for every location $l_i \in \Lambda$, the set of states $\mathcal{I}^\mathcal{S}[p_i]$ is not reachable at location l_i.

The synthesis procedure can only affect non-deterministic locations, we therefore consider $l \in \Lambda^?_{cond}$ with its "else" and "then" branches, represented by l^f and l^t, respectively, and their associated predicates p, p^f and p^t [3].

Let Ψ be a resolving function for $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$ (by Lemma 5, Ψ exists). We can view $\Psi(l)$ as a router, directing program states that reach l to either the "then" branch (i.e., l^t) or the "else" branch (i.e., l^f). Intuitively, this router must ensure doomed states are unreachable at the "then" and "else" branches. As an example, if a state is doomed at l^f, $\Psi(l)$ "routes" it to the "then" branch (namely, to l^t) and hence it never reaches l^f.

To generalize this example, let us denote by D, D^f and D^t, the exact sets of doomed states (non-approximated) at locations l, l^f and l^t, respectively. Since Ψ is a resolving function, D, D^f and D^t *must* be unreachable at locations l, l^f and l^t, respectively.

First, let us consider the set D. Note that, $D = D^f \wedge D^t$, since a state is doomed at l iff it is doomed at both l^f and l^t. Since Ψ is a resolving function, we conclude that states in D must be unreachable at location l (otherwise, the synthesized program cannot be SAFE). This implies that $\Psi(l)$ can direct states that are in D to either the "then" or "else" branch.

Next, consider the set D^f. To ensure that this set is unreachable at l^f, all states in D^f that are reachable at l must be directed to the "then" branch (i.e. to l^t) by $\Psi(l)$. We emphasize that given the fact that D is unreachable, only states in $D^f \backslash D$ can be reachable in l. Symmetrically, all states in the set D^t that are reachable at l (namely, states in $D^t \backslash D$) must be directed to the "else" branch by $\Psi(l)$.

To summarize the above intuition, Figure 3 presents guidelines for defining the function $\Psi(l)$. It illustrates the sets D^f and D^t inside the universe of all program states (i.e., all possible valuations of X) using a Venn diagram. There are four regions in the diagram, defining how $\Psi(l)$ behaves: states in $D^f \backslash D$ are directed to the "then" branch; states in $D^t \backslash D$ are directed to the "else" branch; and states in the Φ regions (states in D and in $(D^t \cup D^f)^c$) can be directed to either branch.

4.2 Defining a Resolving Function

As described in Section 3, for a location $l \in \Lambda$ with an associated predicate p, $\mathcal{I}^\mathcal{S}[p]$ is an over-approximation of states that are doomed at location l. We thus need to construct Ψ such that it directs states to the proper branch using the given over-approximations of doomed states, such that states in $\mathcal{I}^\mathcal{S}[p]$ are unreachable in $\mathcal{G}_\Psi$ at location l.

Based on the above we can use the satisfying interpretation $\mathcal{I}^\mathcal{S}$ in order to define the resolving function Ψ. We define two possible resolving functions: Ψ^f and Ψ^t. We

[3] For readability, in this section we omit s from l_s, l^f_s, l^t_s and their corresponding predicates.

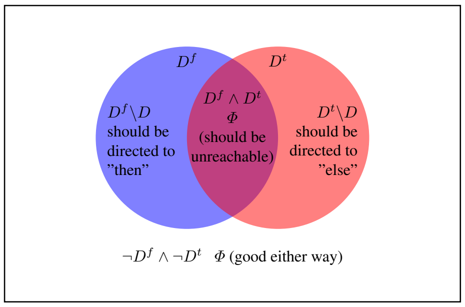

D^f
D^t
$D^f \setminus D$ should be directed to "then"
$D^f \wedge D^t$ Φ (should be unreachable)
$D^t \setminus D$ should be directed to "else"
$\neg D^f \wedge \neg D^t$ Φ (good either way)

Fig. 3: Venn diagrams of the precise sets of doomed states, D^f and D^t

prove that these two solutions are two extremes of a spectrum, hence defining a space of possible solutions. Recall that a resolving function Ψ always defines the predicate for sending states to the "then" branch (i.e., a state is directed to the "then" branch iff it satisfies Ψ). Therefore, the resolving functions Ψ^f and Ψ^t are defined as follows:

$$\forall l_s \in \Lambda^?_{cond} : \Psi^f(l_s) \triangleq \mathcal{I}^{\mathcal{S}}[p^f](X) \wedge \neg\mathcal{I}^{\mathcal{S}}[p](X)$$

$$\forall l_s \in \Lambda^?_{cond} : \Psi^t(l_s) \triangleq \neg(\mathcal{I}^{\mathcal{S}}[p^t](X) \wedge \neg\mathcal{I}^{\mathcal{S}}[p](X)) \equiv \neg\mathcal{I}^{\mathcal{S}}[p^t](X) \vee \mathcal{I}^{\mathcal{S}}[p](X)$$

The following two lemmas prove that solution Ψ^f behaves as desired. That is, if it directs a state σ to the "then" branch, then σ is not doomed at l^t. Moreover, if it directs it to the "else" branch, then σ is either not doomed at l^f or unreachable at l, and therefore also unreachable at l^f. Similar lemmas can be proved for solution Ψ^t.

Lemma 6. *Let σ be a state such that $\sigma \models \Psi^f$. Then $\sigma \not\models D^t$.*

Lemma 7. *Assume that $\mathcal{I}^{\mathcal{S}}[p]$ is unreachable at l. If σ is a state such that $\sigma \not\models \Psi^f$, then either $\sigma \not\models D^f$ or σ is unreachable.*

The Space of Possible Solutions The functions Ψ^f and Ψ^t defined above are two extremes of a spectrum defining a space of solutions. More precisely, every function Ψ that satisfies $\Psi^f \to \Psi \to \Psi^t$ is a resolving function.

Next, we prove that a function Ψ, such that $\Psi^f \to \Psi \to \Psi^t$, is a solution for $\mathcal{S} = (\mathcal{G}, \Lambda^?_{cond})$. Recall that the goal of our approach is to synthesize a program where for every location $l_i \in \Lambda$, the set of states $\mathcal{I}^{\mathcal{S}}[p_i]$ is not reachable at l_i. Moreover, the proof of the following lemma guarantees that $\mathcal{G}_\Psi$ is SAFE by showing that $\mathcal{I}^{\mathcal{S}}$ is a satisfying interpretation of $\Pi^R_{\mathcal{G}_\Psi}$ (Theorem 1). Hence, we conclude that in the synthesized program $\mathcal{G}_\Psi$, for every location $l_i \in \Lambda$, the set of states $\mathcal{I}^{\mathcal{S}}[p_i]$ (which is an over-approximation of states that reach l_{err} from l_i) is not reachable at l_i.

Lemma 8. *Let Ψ be a function s.t. for every $l \in \Lambda^?_{cond}$ the formula*

$$\Psi^f(l)(X) \to \Psi(l)(X) \to \Psi^t(l)(X)$$

is valid. Then, Ψ is a solution of $\mathcal{S}$.

Lemma 9. *There exists a function Ψ s.t. for every $l \in \Lambda^?_{cond}$ the following formula is valid: $\Psi^f(l)(X) \to \Psi(l)(X) \to \Psi^t(l)(X)$*

Summary Lemma 8 and Lemma 9 prove it is possible to synthesize a resolving function (in fact, a set of resolving functions) for $\mathcal{S}$ using the satisfying interpretation $\mathcal{I}^{\mathcal{S}}$ of $\Pi_{\mathcal{S}}$. This proves the correctness of the following Lemma:

Lemma 10. *If $\Pi_{\mathcal{S}}$ is satisfiable then $\mathcal{S} = (\mathcal{G}, \Lambda^{?}_{cond})$ is realizable.*

The correctness of Lemma 10 finalizes the proof of Theorem 2.

4.3 Synthesizing a Solution with a Grammar

While CoSyn does not require a grammar, in some cases where the problem is realizable, it may be desirable to synthesize a solution of a specific plausible shape. To achieve this, one can use CoSyn in conjunction with a synthesis framework such as the well-known Syntax Guided Synthesis (SyGuS) framework [1]. SyGuS is a prominent framework for program synthesis with respect to a formal specification. It limits the search-space to a user-defined grammar G. SyGuS algorithms have the advantage of ensuring that the solution found, if found, will be of a user-desired shape. However, they can only determine unrealizability w.r.t. to the given grammar.

In this setting, assume a grammar G is given, CoSyn is used in the following way:

(i) Execute CoSyn on the given condition synthesis problem. If the problem is unrealizable, stop and return “unrealizable”.
(ii) If the problem is realizable, use the realizability proof to define a specification: for every $l \in \Lambda^{?}_{cond}$ the implication $\Psi^f(l)(X) \to \Psi(l)(X) \to \Psi^t(l)(X)$ must hold.
(iii) Execute a SyGuS tool on the above specification with the given grammar G (on a conjunction of all implications, or one by one[4]).
(iv) Return the synthesized result.

The above shows how CoSyn can be used to complement existing synthesis algorithms. In fact, the generated specification for the synthesis tool is much simpler as it does not need to capture the behavior of the program, only the constraints for each of the locations. This is evident in our experimental evaluation presented in Section 5.

5 Experimental Results

We implemented a prototype of CoSyn on top of the software verification tool SeaHorn [10], which uses Spacer [17] as the CHC solver. To evaluate CoSyn and demonstrate its applicability, we compared it against the SyGuS framework.

In order to compare against SyGuS, we implemented a procedure that translates a condition synthesis problem to SyGuS. We emphasize that since the (partial) program is given and the specification is program correctness, the translation results in a SyGuS problem that requires the solver to only synthesize the missing conditions and loop

[4] We emphasize that the implications in the different locations are independent, thus allowing synthesis of the conditions separately, one by one. Separate synthesis of conditions cannot be done trivially in regular SyGuS, due to the dependency between conditions in the synthesized program.

invariants. To solve SyGuS problems, we use Cvc5 as it is known to be efficient, as demonstrated in the SyGuS competition[5].

The experiments were executed on an AMD EPYC 7742 64-Core Processor with 504GB of RAM, with a timeout of 60 minutes.

Benchmarks The benchmark suite consists of three collections of C language programs: TCAS [7], SV-COMP [2], and Introductory.

The TCAS collection is part of the Siemens suite [7], and consists of 41 faulty versions of a program implementing a traffic collision avoidance system for aircraft. To make the benchmarks suitable for condition synthesis we removed one or more conditions from each of the faulty versions and required equivalence to the correct version as a specification.

The SV-COMP benchmarks are taken from the ReachSafety-ControlFlow category of the SV-COMP competition[6], where they are described as "programs for which the correctness depends mostly on the control-flow structure and integer variables". This collection includes three sub-categories: `nt-drivers-simplified`, `openssl-simplified`, and `locks`. For all SV-COMP benchmarks we selected a condition to remove at random.

For the Introductory collection we implemented a variety of common introductory programming tasks including sort algorithms, string and int manipulations, etc. Then, we removed one or more conditions in different critical points of each algorithm.

Results Two different variants were tested and compared. In the first, no grammar is given to SyGuS, allowing it to synthesize any Boolean term as the solution[7]. This unrestricted mode is similar to how CoSyn is unrestricted by a grammar. The second variant executes SyGuS with a grammar $G1$. In this variant CoSyn executes as described in Section 4.3 using the same $G1$ grammar[8].

The table in Figure 4 summarizes the results. For each tool, we count the number of benchmarks it was able to solve in each category, separated based on the realizability result. As can be seen from the table, in both variants, with and without a grammar, CoSyn solves the most problems, both realizable and unrealizable. The advantage of CoSyn is most noticeable on unrealizable instances.

Note that, in the second variant CoSyn and Cvc5 join forces, with the goal of achieving more syntactically appealing conditions. Note, however, that this effort sometimes leads to a timeout, as demonstrated in the table in Figure 4, on lines 1 and 5, on the R (realizable) column. The "left" and "right" operands of the $+$ sign that appears in the T column differentiate timeout results which are due to CoSyn and Cvc5, respectively. As expected, combining CoSyn and Cvc5 does not influence the unrealizability

[5] Its predecessor, Cvc4, won the competition in most categories: `https://sygus.org/comp/2019/results-slides.pdf`

[6] `https://sv-comp.sosy-lab.org/2022/benchmarks.php`

[7] We used Cvc5's default configuration, except for the addition of `sygus-add-const-grammar` flag, following the advice of Cvc5's developers.

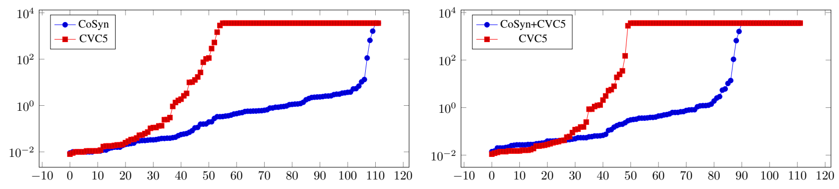
[8] $G1$ is a standard grammar allowing comparisons (e.g. $=$, $\leq$, etc.), using arrays, Integer and Boolean variables.

					Variant 1 No Grammar						Variant 2 Grammar G_1					
					Cosyn			CVC5			Cosyn+CVC5			CVC5		
category	p	c	l	LOC	R	U	T	R	U	T	R	U	T	R	U	T
introductory	35	59	54	1089	24	8	3	21	5	9	23	8	3+1	20	5	10
sv-comp/locks	13	15	13	909	11	2	0	11	2	0	11	2	0	11	2	0
sv-comp/ntdrivers-simplified	7	8	4	7831	5	2	0	4	2	1	5	2	0	4	1	2
sv-comp/openssl-simplified	23	51	23	12893	5	18	0	4	1	18	5	18	0	4	1	18
tcas	34	64	0	8059	21	13	0	5	0	29	2	13	0+19	2	0	32
total	112	197	94	30781	66	43	3	45	10	57	46	43	23	41	9	62

Fig. 4: Results summary. For each category, the columns p, c, l and LOC represent the total number of synthesis problems, conditions (after inlining), loops and lines-of-code, respectively. For each tool, columns R and U represent the number of realizable and unrealizable problems solved by the tool. T represents Timeout.

results (U column) when compared to CoSyn alone. That is, column U in CoSyn and CoSyn +Cvc5 are identical.

The graphs in Figure 5 summarize runtime results on all examples. As evident by these graphs, it is not only that CoSyn solves more instances, it also performs better w.r.t. runtime. Moreover, using CoSyn in conjunction with Cvc5 (Figure 5b), improves Cvc5's performance significantly, allowing it to solve more instances in less time. This shows that a SyGuS engine can greatly benefit from the addition of CoSyn.



(a) No grammar (b) Grammar $G1$

Fig. 5: Runtime (seconds) comparison. X/Y-axis represent the synthesis problems and runtime, respectively.

6 Related Work

As mentioned above, Syntax-guided synthesis (SyGuS) [1] is widely applicable and many state-of-the-art program synthesis algorithms use the SyGuS framework [13,23,24,14,21,3]. Another significant framework is semantics-guided synthesis (SemGuS) [16], which in addition to the specification and grammar, supplies a set of inference rules to define the semantics of constructs in the grammar. This is implemented in the tool Messy.

Similar to our work, the realizability of a SemGuS problem is reduced in Messy to a CHC satisfiability problem and a solution is extracted from a satisfying interpretation, if one exists. However, the CHC satisfiability problem solved by Messy and by CoSyn are completely different. The query in MESSY intuitively asks whether the initial symbol of the grammar can produce a term whose semantics coincide with the specification for a *particular, finite, set of inputs*. In contrast, our CHC satisfiability problem encodes the computation of doomed states for a given non-deterministic program. It does not encode any syntactic constraints, thus its unrealizability result is definite. Further, CoSyn ensures correctness for *all* inputs. However, CoSyn is specific to the problem of condition synthesis and cannot handle arbitrary synthesis problems.

Another synthesis approach is sketch-based synthesis [26], which allows to leave holes in place of code fragments, to be derived by a synthesizer. However, the code fragment that can be used to replace a hole in SKETCH is always limited in both structure and size. Therefore, if SKETCH finds the problem unrealizable, we can only conclude that there is no solution using the particular syntax. In contrast, our approach does not restrict the generated conditions syntactically at all. Further, [26] only performs bounded loop unwinding, while CoSyn guarantees correctness for unbounded computations. Another difference is that SKETCH interprets integer variables as fixed-width bit-vectors while CoSyn relies on SEAHORN, which treats integer variables using integer semantics.

Finally, many synthesis and repair tools, including some mentioned above, use the counterexample guided inductive synthesis (CEGIS) framework [26,20,5,16]. They initially find a solution for only a finite set of inputs I. If verification fails for input $i \notin I$, then i is added to I and the process is repeated. However, the CEGIS process may diverge and may become very costly. CoSyn does not require the CEGIS framework since it directly solves the synthesis problem for all inputs.

Recently, several works focus mainly on unrealizability [11,12,8], while applying SyGuS or CEGIS. In [15] a logic for proving unrealizability has been proposed. However, these works do not solve condition synthesis problems or take advantage of the power of CHC solvers.

7 Conclusion

This work presents a novel approach to *(un)realizability of condition synthesis*, based on a reduction to Constrained Horn Clauses (CHC). Our algorithm, CoSyn, relies on a central notion called *doomed states*. We encode into CHC the question of whether the program includes an initial doomed state and exploit the encoding to determine (un)realizability of the synthesis problem. A doomed initial state is returned as evidence, if the problem is unrealizable. Otherwise, conditions are provided as evidence – based on these conditions the program can be proved SAFE.

Our approach can handle any number of missing conditions in the program. Our experiments show that CoSyn can solve both realizable and unrealizable examples efficiently, and can complement SyGuS tools.

References

1. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.
2. D. Beyer. Competition on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012.
3. S. Bhatia, S. Padhi, N. Natarajan, R. Sharma, and P. Jain. Oasis: Ilp-guided synthesis of loop invariants. *CoRR*, 2019.
4. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.
5. R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC–An automatic debugging environment for C programs. In *Hardware and Software: Verification and Testing*, pages 260–265. Springer, 2012.
6. F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.
7. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.
8. A. Farzan, D. Lette, and V. Nicolet. Recursion synthesis with unrealizability witnesses. In R. Jhala and I. Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 244–259. ACM, 2022.
9. G. Fedyukovich and A. Gupta. Functional Synthesis with Examples. In *CP*, volume 11802 LNCS, pages 547–564, 2019.
10. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.
11. Q. Hu, J. Breck, J. Cyphert, L. D'Antoni, and T. W. Reps. Proving unrealizability for syntax-guided synthesis. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 335–352. Springer, 2019.
12. Q. Hu, L. D'Antoni, J. Cyphert, and T. Reps. Exact and Approximate Unrealizability of Syntax-Guided Synthesis Problems. In *PLDI*, 2020.
13. Q. Hu, I. Evavold, R. Samanta, R. Singh, and L. D'Antoni. Program Repair via Direct State Manipulation. 2018.
14. K. Huang, X. Qiu, P. Shen, and Y. Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174, 2020.
15. J. Kim, L. D'Antoni, and T. W. Reps. Unrealizability logic. In *POPL '23: Proceedings of the 50th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2023.
16. J. Kim, Q. Hu, L. D'Antoni, and T. W. Reps. Semantics guided synthesis. In *POPL*, 2020.

17. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.*, 48(3):175–205, 2016.
18. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete Functional Synthesis. In *PLDI*, 2010.
19. F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE*, pages 166–178. ACM, 2015.
20. T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin. Automatic Program Repair Using Formal Verification and Expression Templates. In *VMCAI*, 2019.
21. S. Padhi, R. Sharma, and T. Millstein. Loopinvgen: A Loop Invariant Generator based on Precondition Inference. *arXiv*, 2017.
22. O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *OOPSLA*, volume 25-30-Oct-, pages 107–126, 2015.
23. A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett, and C. Tinelli. CVC4SY: Smart and fast term enumeration for syntax-guided synthesis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11562 LNCS:74–83, 2019.
24. X. Si, W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik. Syntax-guided synthesis of datalog programs. In *ESEC/FSE*, pages 515–527, 2018.
25. S. So and H. Oh. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *SAS*, pages 364–381, 2017.
26. A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ACM Sigplan Notices*, volume 41, pages 404–415. ACM, 2006.
27. S. Srivastava, S. Gulwani, and J. S. Foster. From Program Verification to Program Synthesis. *POPL*, 2010.
28. X. WANG, I. DILLIG, and R. SINGH. Program Synthesis using Abstraction Refinement. *arXiv*, 2(January 2018), 2017.
29. Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise Condition Synthesis for Program Repair. In *ICSE*, 2017.