Proving Unrealizability for Syntax-Guided Synthesis

Qinheping Hu[1(✉)], Jason Breck[1], John Cyphert[1], Loris D'Antoni[1], and Thomas Reps[1,2]

[1] University of Wisconsin-Madison, Madison, USA
qhu28@wisc.edu
[2] GrammaTech, Inc., Ithaca, USA

Abstract. We consider the problem of automatically establishing that a given syntax-guided-synthesis (SyGuS) problem is *unrealizable* (i.e., has no solution). Existing techniques have quite limited ability to establish unrealizability for general SyGuS instances in which the grammar describing the search space contains infinitely many programs. By encoding the synthesis problem's grammar G as a nondeterministic program P_G, we reduce the unrealizability problem to a reachability problem such that, if a standard program-analysis tool can establish that a certain assertion in P_G always holds, then the synthesis problem is unrealizable.

Our method can be used to augment existing SyGuS tools so that they can establish that a successfully synthesized program q is *optimal* with respect to some syntactic cost—e.g., q has the fewest possible if-then-else operators. Using known techniques, grammar G can be transformed to generate the set of all programs with lower costs than q—e.g., fewer conditional expressions. Our algorithm can then be applied to show that the resulting synthesis problem is unrealizable. We implemented the proposed technique in a tool called nope. nope can prove unrealizability for 59/132 variants of existing linear-integer-arithmetic SyGuS benchmarks, whereas all existing SyGuS solvers lack the ability to prove that these benchmarks are unrealizable, and time out on them.

1 Introduction

The goal of program synthesis is to find a program in some search space that meets a specification—e.g., satisfies a set of examples or a logical formula. Recently, a large family of synthesis problems has been unified into a framework called *syntax-guided synthesis* (SyGuS). A SyGuS problem is specified by a regular-tree grammar that describes the search space of programs, and a logical formula that constitutes the behavioral specification. Many synthesizers now support a specific format for SyGuS problems [1], and compete in annual synthesis competitions [2]. Thanks to these competitions, these solvers are now quite mature and are finding a wealth of applications [9].

Consider the SyGuS problem to synthesize a function f that computes the maximum of two variables x and y, denoted by $(\psi_{\texttt{max2}}(f, x, y), G_1)$. The goal is to

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 335–352, 2019.
https://doi.org/10.1007/978-3-030-25540-4_18

create e_f—an expression-tree for f—where e_f is in the language of the following regular-tree grammar G_1:

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{GreaterThan}(\text{Start}, \text{Start}) \mid \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \end{aligned}$$

and $\forall x, y.\psi_{\texttt{max2}}([\![e_f]\!], x, y)$ is valid, where $[\![e_f]\!]$ denotes the meaning of e_f, and

$$\psi_{\texttt{max2}}(f, x, y) := f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y).$$

SyGuS solvers can easily find a solution, such as

$$e := \text{IfThenElse}(\text{GreaterThan}(x, y), x, y).$$

Although many solvers can now find solutions efficiently to many SyGuS problems, there has been effectively no work on the much harder task of proving that a given SyGuS problem is *unrealizable*—i.e., it does not admit a solution. For example, consider the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$, where G_2 is the more restricted grammar with if-then-else operators and conditions stripped out:

$$\text{Start} ::= \text{Plus}(\text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1$$

This SyGuS problem does *not* have a solution, because no expression generated by G_2 meets the specification.[1] However, to the best of our knowledge, current SyGuS solvers cannot prove that such a SyGuS problem is unrealizable.[2]

A key property of the previous example is that the grammar is infinite. When such a SyGuS problem is realizable, any search technique that systematically explores the infinite search space of possible programs will eventually identify a solution to the synthesis problem. In contrast, proving that a problem is unrealizable requires showing that *every* program in the *infinite* search space *fails to satisfy* the specification. This problem is in general undecidable [6]. Although we cannot hope to have an algorithm for establishing unrealizability, the challenge is to find a technique that succeeds for the kinds of problems encountered in practice. Existing synthesizers can detect the absence of a solution in certain cases (e.g., because the grammar is finite, or is infinite but only generate a finite number of functionally distinct programs). However, in practice, as our

[1] Grammar G_2 only generates terms that are equivalent to some linear function of x and y; however, the maximum function cannot be described by a linear function.

[2] The synthesis problem presented above is one that is generated by a recent tool called QSyGuS, which extends SyGuS with quantitative syntactic objectives [10]. The advantage of using quantitative objectives in synthesis is that they can be used to produce higher-quality solutions—e.g., smaller, more readable, more efficient, etc. The synthesis problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ arises from a QSyGuS problem in which the goal is to produce an expression that (i) satisfies the specification $\psi_{\texttt{max2}}(f, x, y)$, and (ii) uses the smallest possible number of if-then-else operators. Existing SyGuS solvers can easily produce a solution that uses one if-then-else operator, but cannot prove that no better solution exists—i.e., $(\psi_{\texttt{max2}}(f, x, y), G_2)$ is unrealizable.

experiments show, this ability is limited—no existing solver was able to show unrealizability for any of the examples considered in this paper.

In this paper, we present a technique for proving that a possibly infinite SyGuS problem is unrealizable. Our technique builds on two ideas.

1. We observe that unrealizability can often be proven using *finitely many input examples*. In Sect. 2, we show how the example discussed above can be proven to be unrealizable using four input examples—$(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.
2. We devise a way to encode a SyGuS problem $(\psi(f,\bar{x}),G)$ over a finite set of examples E as a *reachability problem in a recursive program* $P[G,E]$. In particular, the program that we construct has an assertion that holds if and only if the given SyGuS problem is unrealizable. Consequently, *unrealizability* can be proven by establishing that the assertion always holds. This property can often be established by a conventional program-analysis tool.

The encoding mentioned in item 2 is non-trivial for three reasons. The following list explains each issue, and sketches how they are addressed

(1) Infinitely many terms. We need to model the infinitely many terms generated by the grammar of a given synthesis problem $(\psi(f,\bar{x}),G)$.

To address this issue, we use non-determinism and recursion, and give an encoding $P[G,E]$ such that (i) each non-deterministic path p in the program $P[G,E]$ corresponds to a possible expression e_p that G can generate, and (ii) for each expression e that G can generate, there is a path p_e in $P[G,E]$. (There is an isomorphism between paths and the expression-trees of G)

(2) Nondeterminism. We need the computation performed along each path p in $P[G,E]$ to mimic the execution of expression e_p. Because the program uses non-determinism, we need to make sure that, for a given path p in the program $P[G,E]$, computational steps are carried out that mimic the evaluation of e_p for *each* of the finitely many example inputs in E.

We address this issue by threading the expression-evaluation computations associated with each example in E through the *same* non-deterministic choices.

(3) Complex Specifications. We need to handle specifications that allow for nested calls of the programs being synthesized.

For instance, consider the specification $f(f(x)) = x$. To handle this specification, we introduce a new variable y and rewrite the specification as $f(x) = y \wedge f(y) = x$. Because y is now also used as an input to f, we will thread both the computations of x and y through the non-deterministic recursive program.

Our work makes the following contributions:

- We reduce the SyGuS unrealizability problem to a reachability problem to which standard program-analysis tools can be applied (Sects. 2 and 4).
- We observe that, for many SyGuS problems, unrealizability can be proven using *finitely many input examples*, and use this idea to apply the Counter-Example-Guided Inductive Synthesis (CEGIS) algorithm to the problem of proving unrealizability (Sect. 3).

- We give an encoding of a SyGuS problem $(\psi(f, \bar{x}), G)$ over a finite set of examples E as a reachability problem in a nondeterministic recursive program $P[G, E]$, which has the following property: if a certain assertion in $P[G, E]$ always holds, then the synthesis problem is unrealizable (Sect. 4).
- We implement our technique in a tool NOPE using the ESolver synthesizer [2] as the SyGuS solver and the SeaHorn tool [8] for checking reachability. NOPE is able to establish unrealizability for 59 out of 132 variants of benchmarks taken from the SyGuS competition. In particular, NOPE solves all benchmarks with no more than 15 productions in the grammar and requiring no more than 9 input examples for proving unrealizability. Existing SyGuS solvers lack the ability to prove that these benchmarks are unrealizable, and time out on them.

Section 6 discusses related work. Some additional technical material, proofs, and full experimental results are given in [13].

2 Illustrative Example

In this section, we illustrate the main components of our framework for establishing the unrealizability of a SyGuS problem.

Consider the SyGuS problem to synthesize a function f that computes the maximum of two variables x and y, denoted by $(\psi_{\texttt{max2}}(f, x, y), G_1)$. The goal is to create e_f—an expression-tree for f—where e_f is in the language of the following regular-tree grammar G_1:

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{GreaterThan}(\text{Start}, \text{Start}) \mid \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \end{aligned}$$

and $\forall x, y.\psi_{\texttt{max2}}([\![e_f]\!], x, y)$ is valid, where $[\![e_f]\!]$ denotes the meaning of e_f, and

$$\psi_{\texttt{max2}}(f, x, y) := f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y).$$

SyGuS solvers can easily find a solution, such as

$$e := \text{IfThenElse}(\text{GreaterThan}(x, y), x, y).$$

Although many solvers can now find solutions efficiently to many SyGuS problems, there has been effectively no work on the much harder task of proving that a given SyGuS problem is *unrealizable*—i.e., it does not admit a solution. For example, consider the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$, where G_2 is the more restricted grammar with if-then-else operators and conditions stripped out:

$$\text{Start} ::= \text{Plus}(\text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1$$

This SyGuS problem does *not* have a solution, because no expression generated by G_2 meets the specification.[3] However, to the best of our knowledge, current

[3] Grammar G_2 generates all linear functions of x and y, and hence generates an infinite number of functionally distinct programs; however, the maximum function cannot be described by a linear function.

SyGuS solvers cannot prove that such a SyGuS problem is unrealizable. As an example, we use the problem $(\psi_{\texttt{max2}}(f,x,y),G_2)$ discussed in Sect. 1, and show how unrealizability can be proven using four input examples: $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.

```
1  int I_0;
2  void Start(int x_0,int y_0){
3    if(nd()){  // Encodes ``Start ::= Plus(Start, Start)''
4      Start(x_0, y_0);
5      int tempL_0 = I_0;
6      Start(x_0, y_0);
7      int tempR_0 = I_0;
8      I_0 = tempL_0 + tempR_0;
9    }
10   else if(nd()) I_0 = x_0;  // Encodes ``Start ::= x''
11   else if(nd()) I_0 = y_0;  // Encodes ``Start ::= y''
12   else if(nd()) I_0 = 1;    // Encodes ``Start ::= 1''
13   else          I_0 = 0;    // Encodes ``Start ::= 0''
14 }
15
16 bool spec(int x, int y, int f){
17   return (f>=x && f>=y && (f==x || f==y))
18 }
19
20 void main(){
21   int x_0 = 0; int y_0 = 1;   // Input example (0,1)
22   Start(x_0,y_0);
23   assert(!spec(x_0,y_0,I_0));
24 }
```

Fig. 1. Program $P[G_2,E_1]$ created during the course of proving the unrealizability of $(\psi_{\texttt{max2}}(f,x,y),G_2)$ using the set of input examples $E_1=\{(0,1)\}$.

Our method can be seen as a variant of Counter-Example-Guided Inductive Synthesis (CEGIS), in which the goal is to create a program P in which a certain assertion always holds. Until such a program is created, each round of the algorithm returns a counter-example, from which we extract an additional input example for the original SyGuS problem. On the i^{th} round, the current set of input examples E_i is used, together with the grammar—in this case G_2—and the specification of the desired behavior—$\psi_{\texttt{max2}}(f,x,y)$, to create a candidate program $P[G_2,E_i]$. The program $P[G_2,E_i]$ contains an assertion, and a standard program analyzer is used to check whether the assertion always holds.

Suppose that for the SyGuS problem $(\psi_{\texttt{max2}}(f,x,y),G_2)$ we start with just the one example input $(0,1)$—i.e., $E_1=\{(0,1)\}$. Figure 1 shows the initial program $P[G_2,E_1]$ that our method creates. The function `spec` implements the predicate $\psi_{\texttt{max2}}(f,x,y)$. (All of the programs $\{P[G_2,E_i]\}$ use the same function `spec`). The initialization statements "`int x_0 = 0; int y_0 = 1;`" at line (21) in procedure `main` correspond to the input example $(0,1)$. The recursive procedure `Start` encodes the productions of grammar G_2. `Start` is nondeterministic; it contains four calls to an external function `nd()`, which returns

a non-deterministically chosen Boolean value. The calls to `nd()` can be understood as controlling whether or not a production is selected from G_2 during a top-down, left-to-right generation of an expression-tree: lines (3)–(8) correspond to "Start ::= Plus(Start, Start)," and lines (10), (11), (12), and (13) correspond to "Start ::= x," "Start ::= y," "Start ::= 1," and "Start ::= 0," respectively. The code in the five cases in the body of `Start` encodes the semantics of the respective production of G_2; in particular, the statements that are executed along any execution path of $P[G_2, E_1]$ implement the *bottom-up evaluation of some expression-tree that can be generated by* G_2. For instance, consider the path that visits statements in the following order (for brevity, some statement numbers have been elided):

$$21\ \ 22\ \ (_{\texttt{Start}}\ \ 3\ \ 4\ \ (_{\texttt{Start}}\ \ 10\ \ )_{\texttt{Start}}\ \ 6\ \ (_{\texttt{Start}}\ \ 12\ \ )_{\texttt{Start}}\ \ 8\ \ )_{\texttt{Start}}\ \ 23, \quad (1)$$

where $(_{\texttt{Start}}$ and $)_{\texttt{Start}}$ indicate entry to, and return from, procedure `Start`, respectively. Path (1) corresponds to the top-down, left-to-right generation of the expression-tree `Plus(x,1)`, interleaved with the tree's bottom-up evaluation.

Note that with path (1), when control returns to `main`, variable `I_0` has the value 1, and thus the assertion at line (23) fails.

A sound program analyzer will discover that some such path exists in the program, and will return the sequence of non-deterministic choices required to follow one such path. Suppose that the analyzer chooses to report path (1); the sequence of choices would be t, f, t, f, f, f, t, which can be decoded to create the expression-tree `Plus(x,1)`. At this point, we have a candidate definition for f: $f = x + 1$. This formula can be checked using an SMT solver to see whether it satisfies the behavioral specification $\psi_{\texttt{max2}}(f, x, y)$. In this case, the SMT solver returns "false." One counter-example that it could return is $(0, 0)$.

At this point, program $P[G_2, E_2]$ would be constructed using both of the example inputs $(0, 1)$ and $(0, 0)$. Rather than describe $P[G_2, E_2]$, we will describe the final program constructed, $P[G_2, E_4]$ (see Fig. 2).

As can be seen from the comments in the two programs, program $P[G_2, E_4]$ has the same basic structure as $P[G_2, E_1]$.

- `main` begins with initialization statements for the four example inputs.
- `Start` has five cases that correspond to the five productions of G_2.

The main difference is that because the encoding of G_2 in `Start` uses non-determinism, we need to make sure that along *each* path p in $P[G_2, E_4]$, each of the example inputs is used to evaluate the *same* expression-tree. We address this issue by threading the expression-evaluation computations associated with each of the example inputs through the *same* non-deterministic choices. That is, each of the five "production cases" in `Start` has four encodings of the production's semantics—one for each of the four expression evaluations. By this means, the statements that are executed along path p perform *four simultaneous bottom-up evaluations* of the expression-tree from G_2 that corresponds to p.

Programs $P[G_2, E_2]$ and $P[G_2, E_3]$ are similar to $P[G_2, E_4]$, but their paths carry out two and three simultaneous bottom-up evaluations, respectively. The

```
1  int I_0, I_1, I_2, I_3;
2  void Start(int x_0,int y_0,...,int x_3,int y_3){
3    if(nd()){  // Encodes ``Start ::= Plus(Start, Start)''
4      Start(x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3);
5      int tempL_0 = I_0; int tempL_1 = I_1;
6      int tempL_2 = I_2; int tempL_3 = I_3;
7      Start(x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3);
8      int tempR_0 = I_0; int tempR_1 = I_1;
9      int tempR_2 = I_2; int tempR_3 = I_3;
10     I_0 = tempL_0 + tempR_0;
11     I_1 = tempL_1 + tempR_1;
12     I_2 = tempL_2 + tempR_2;
13     I_3 = tempL_3 + tempR_3;}
14   else if(nd()) {  // Encodes ``Start ::= x''
15     I_0 = x_0; I_1 = x_1; I_2 = x_2; I_3 = x_3;}
16   else if(nd()) {  // Encodes ``Start ::= y''
17     I_0 = y_0; I_1 = y_1; I_2 = y_2; I_3 = y_3;}
18   else if(nd()) {  // Encodes ``Start ::= 1''
19     I_0 = 1;   I_1 = 1;   I_2 = 1;   I_3 = 1;}
20   else {           // Encodes ``Start ::= 0''
21     I_0 = 0;   I_1 = 0;   I_2 = 0;   I_3 = 0;}
22 }
23
24 bool spec(int x, int y, int f){
25   return (f>=x && f>=y && (f==x || f==y))
26 }
27
28 void main(){
29   int x_0 = 0; int y_0 = 1;  // Input example (0,1)
30   int x_1 = 0; int y_1 = 0;  // Input example (0,0)
31   int x_2 = 1; int y_2 = 1;  // Input example (1,1)
32   int x_3 = 1; int y_3 = 0;  // Input example (1,0)
33   Start(x_0,y_0,x_1,y_1,x_2,y_2,x_3,y_3);
34   assert(   !spec(x_0,y_0,I_0) || !spec(x_1,y_1,I_1)
35          || !spec(x_2,y_2,I_2) || !spec(x_3,y_3,I_3));
36 }
```

Fig. 2. Program $P[G_2, E_4]$ created during the course of proving the unrealizability of $(\psi_{\texttt{max2}}(f, x, y), G_2)$ using the set of input examples $E_4 = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$.

actions taken during rounds 2 and 3 to generate a new counter-example—and hence a new example input—are similar to what was described for round 1. On round 4, however, the program analyzer will determine that the assertion on lines (34)–(35) always holds, which means that there is no path through $P[G_2, E_4]$ for which the behavioral specification holds for all of the input examples. This property means that there is no expression-tree that satisfies the specification—i.e., the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ is unrealizable.

Our implementation uses the program-analysis tool SeaHorn [8] as the assertion checker. In the case of $P[G_2, E_4]$, SeaHorn takes only 0.5 s to establish that the assertion in $P[G_2, E_4]$ always holds.

3 SyGuS, Realizability, and CEGIS

3.1 Background

Trees and Tree Grammars. A *ranked alphabet* is a tuple (Σ, rk_Σ) where Σ is a finite set of symbols and $rk_\Sigma : \Sigma \to \mathbb{N}$ associates a rank to each symbol. For every $m \geq 0$, the set of all symbols in Σ with rank m is denoted by $\Sigma^{(m)}$. In our examples, a ranked alphabet is specified by showing the set Σ and attaching the respective rank to every symbol as a superscript—e.g., $\Sigma = \{+^{(2)}, c^{(0)}\}$. (For brevity, the superscript is sometimes omitted). We use T_Σ to denote the set of all (ranked) trees over Σ—i.e., T_Σ is the smallest set such that (i) $\Sigma^{(0)} \subseteq T_\Sigma$, (ii) if $\sigma^{(k)} \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma^{(k)}(t_1, \cdots, t_k) \in T_\Sigma$. In what follows, we assume a fixed ranked alphabet (Σ, rk_Σ).

In this paper, we focus on *typed* regular tree grammars, in which each non-terminal and each symbol is associated with a type. There is a finite set of types $\{\tau_1, \ldots, \tau_k\}$. Associated with each symbol $\sigma^{(i)} \in \Sigma^{(i)}$, there is a type assignment $a_{\sigma^{(i)}} = (\tau_0, \tau_1, \ldots, \tau_i)$, where τ_0 is called the *left-hand-side type* and $\tau_1, \ldots, \tau_i$ are called the *right-hand-side types*. Tree grammars are similar to word grammars, but generate trees over a ranked alphabet instead of words.

Definition 1 (Regular Tree Grammar). *A* **typed regular tree grammar** *(RTG) is a tuple $G = (N, \Sigma, S, a, \delta)$, where N is a finite set of non-terminal symbols of arity 0; Σ is a ranked alphabet; $S \in N$ is an initial non-terminal; a is a type assignment that gives types for members of $\Sigma \cup N$; and δ is a finite set of productions of the form $A_0 \to \sigma^{(i)}(A_1, \ldots, A_i)$, where for $1 \leq j \leq i$, each $A_j \in N$ is a non-terminal such that if $a(\sigma^{(i)}) = (\tau_0, \tau_1, \ldots, \tau_i)$ then $a(A_j) = \tau_j$.*

In a SyGuS problem, each variable, such as x and y in the example RTGs in Sect. 1, is treated as an arity-0 symbol—i.e., $x^{(0)}$ and $y^{(0)}$.

Given a tree $t \in T_{\Sigma \cup N}$, applying a production $r = A \to \beta$ to t produces the tree t' resulting from replacing the left-most occurrence of A in t with the right-hand side β. A tree $t \in T_\Sigma$ is generated by the grammar G—denoted by $t \in L(G)$—iff it can be obtained by applying a sequence of productions $r_1 \cdots r_n$ to the tree whose root is the initial non-terminal S.

Syntax-Guided Synthesis. A SyGuS problem is specified with respect to a background theory T—e.g., linear arithmetic—and the goal is to synthesize a function f that satisfies two constraints provided by the user. The first constraint, $\psi(f, \bar{x})$, describes a *semantic property* that f should satisfy. The second constraint limits the *search space* S of f, and is given as a set of expressions specified by an RTG G that defines a subset of all terms in T.

Definition 2 (SyGuS). *A* SyGuS *problem over a background theory T is a pair $sy = (\psi(f, \bar{x}), G)$ where G is a regular tree grammar that only contains terms in T—i.e., $L(G) \subseteq T$—and $\psi(f, \bar{x})$ is a Boolean formula constraining the semantic behavior of the synthesized program f.*

A SyGuS *problem is* **realizable** *if there exists a expression $e \in L(G)$ such that $\forall \bar{x}.\psi([\![e]\!], \bar{x})$ is true. Otherwise we say that the problem is* **unrealizable**.

Theorem 1 (Undecidability [6]). *Given a* SyGuS *problem sy, it is undecidable to check whether sy is realizable.*

Counterexample-Guided Inductive Synthesis. The Counterexample-Guided Inductive Synthesis (CEGIS) algorithm is a popular approach to solving synthesis problems. Instead of directly looking for an expression that satisfies the specification φ on *all* possible inputs, the CEGIS algorithm uses a synthesizer S that can find expressions that are correct on a *finite* set of examples E. If S finds a solution that is correct on all elements of E, CEGIS uses a verifier V to check whether the discovered solution is also correct for all possible inputs to the problem. If not, a counterexample obtained from V is added to the set of examples, and the process repeats. More formally, CEGIS starts with an empty set of examples E and repeats the following steps:

1. Call the synthesizer S to find an expression e such that $\psi^E(\llbracket e \rrbracket, \bar{x}) \stackrel{\text{def}}{=} \forall \bar{x} \in E.\psi(\llbracket e \rrbracket, \bar{x})$ holds and go to step 2; return *unrealizable* if no expression exists.
2. Call the verifier V to find a model c for the formula $\neg\psi(\llbracket e \rrbracket, \bar{x})$, and add c to the counterexample set E; return e as a valid solution if no model is found.

Because SyGuS problems are only defined over first-order decidable theories, any SMT solver can be used as the verifier V to check whether the formula $\neg\psi(\llbracket e \rrbracket, \bar{x})$ is satisfiable. On the other hand, providing a synthesizer S to find solutions such that $\forall \bar{x} \in E.\psi(\llbracket e \rrbracket, \bar{x})$ holds is a much harder problem because e is a second-order term drawn from an infinite search space. In fact, checking whether such an e exists is an undecidable problem [6].

The main contribution of our paper is a reduction of the unrealizability problem—i.e., the problem of proving that there is no expression $e \in L(G)$ such that $\forall \bar{x} \in E.\psi(\llbracket e \rrbracket, \bar{x})$ holds—to an unreachability problem (Sect. 4). This reduction allows us to use existing (un)reachability verifiers to check whether a SyGuS instance is unrealizable.

3.2 CEGIS and Unrealizability

The CEGIS algorithm is sound but incomplete for proving unrealizability. Given a SyGuS problem $sy = (\psi(f, \bar{x}), G)$ and a finite set of inputs E, we denote with $sy^E := (\psi^E(f, \bar{x}), G)$ the corresponding SyGuS problem that only requires the function f to be correct on the examples in E.

Lemma 1 (Soundness). *If sy^E is unrealizable then sy is unrealizable.*

Even when given a perfect synthesizer S—i.e., one that can solve a problem sy^E for every possible set E—there are SyGuS problems for which the CEGIS algorithm is not powerful enough to prove unrealizability.

Lemma 2 (Incompleteness). *There exists an unrealizable* SyGuS *problem sy such that for every finite set of examples E the problem sy^E is realizable.*

Despite this negative result, we will show that a CEGIS algorithm can prove unrealizability for many SyGuS instances (Sect. 5).

4 From Unrealizability to Unreachability

In this section, we show how a SyGuS problem for finitely many examples can be reduced to a reachability problem in a non-deterministic, recursive program in an imperative programming language.

4.1 Reachability Problems

A *program* P takes an initial state I as input and outputs a final state O, i.e., $[\![P]\!](I) = O$ where $[\![\cdot]\!]$ denotes the semantic function of the programming language. As illustrated in Sect. 2, we allow a program to contain calls to an external function `nd()`, which returns a non-deterministically chosen Boolean value. When program P contains calls to `nd()`, we use $\hat{P}$ to denote the program that is the same as P except that $\hat{P}$ takes an additional integer input `n`, and each call `nd()` is replaced by a call to a local function `nextbit()` defined as follows:

```
bool nextbit(){bool b = n%2; n=n»1; return b;}.
```

In other words, the integer parameter `n` of $\hat{P}$[`n`] formalizes all of the non-deterministic choices made by P in calls to `nd()`.

For the programs $P[G, E]$ used in our unrealizability algorithm, the only calls to `nd()` are ones that control whether or not a production is selected from grammar G during a top-down, left-to-right generation of an expression-tree. Given `n`, we can decode it to identify which expression-tree `n` represents.

Example 1. Consider again the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ discussed in Sect. 2. In the discussion of the initial program $P[G_2, E_1]$ (Fig. 1), we hypothesized that the program analyzer chose to report path (1) in P, for which the sequence of non-deterministic choices is t, f, t, f, f, f, t. That sequence means that for $\hat{P}$[`n`], the value of `n` is 1000101 (base 2) (or 69 (base 10)). The 1s, from low-order to high-order position, represent choices of production instances in a top-down, left-to-right generation of an expression-tree. (The 0s represent rejected possible choices). The rightmost 1 in `n` corresponds to the choice in line (3) of "`Start ::= Plus(Start, Start)`"; the 1 in the third-from-rightmost position corresponds to the choice in line (10) of "`Start ::= x`" as the left child of the `Plus` node; and the 1 in the leftmost position corresponds to the choice in line (12) of "`Start ::= 1`" as the right child. By this means, we learn that the behavioral specification $\psi_{\texttt{max2}}(f, x, y)$ holds for the example set $E_1 = \{(0, 1)\}$ for $f \mapsto$ `Plus(x,1)`. □

Definition 3 (Reachability Problem). *Given a program $\hat{P}$[`n`], containing assertion statements and a non-deterministic integer input `n`, we use re_P to denote the corresponding reachability problem. The reachability problem re_P is **satisfiable** if there exists a value n that, when bound to `n`, falsifies any of the assertions in $\hat{P}$[`n`]. The problem is **unsatisfiable** otherwise.*

4.2 Reduction to Reachability

The main component of our framework is an encoding *enc* that given a SyGuS problem $sy^E = (\psi^E(f,x), G)$ over a set of examples $E = \{c_1, \ldots, c_k\}$, outputs a program $P[G,E]$ such that sy^E is **realizable** if and only if $re_{enc(sy,E)}$ is **satisfiable**. In this section, we define all the components of $P[G,E]$, and state the correctness properties of our reduction.

Remark: In this section, we assume that in the specification $\psi(f,x)$ every occurrence of f has x as input parameter. We show how to overcome this restriction in App. A [13]. In the following, we assume that the input x has type τ_I, where τ_I could be a complex type—e.g., a tuple type.

Program Construction. Recall that the grammar G is a tuple $(N, \Sigma, S, a, \delta)$. First, for each non-terminal $A \in N$, the program $P[G,E]$ contains k global variables $\{\texttt{g\_1\_A}, \ldots, \texttt{g\_k\_A}\}$ of type $a(A)$ that are used to express the values resulting from evaluating expressions generated from non-terminal A on the k examples. Second, for each non-terminal $A \in N$, the program $P[G,E]$ contains a function

$$\texttt{void funcA(}\tau_I\ \texttt{v1}, \ldots, \tau_I\ \texttt{vk)\{}\ \ \mathit{bodyA}\ \texttt{\}}$$

We denote by $\delta(A) = \{r_1, \ldots, r_m\}$ the set of production rules of the form $A \to \beta$ in δ. The body *bodyA* of `funcA` has the following structure:

$$\begin{array}{l}\texttt{if(nd())}\ \{\texttt{En}_\delta(r_1)\}\\ \texttt{else if(nd())}\ \{\texttt{En}_\delta(r_2)\}\\ \ldots\\ \texttt{else}\ \{\texttt{En}_\delta(r_m)\}\end{array}$$

The encoding $\texttt{En}_\delta(r)$ of a production $r = A_0 \to b^{(j)}(A_1, \cdots, A_j)$ is defined as follows (τ_i denotes the type of the term A_i):

$$\begin{array}{l}\texttt{funcA1(v1,...,vk);}\\ \tau_1\ \texttt{child\_1\_1} = \texttt{g\_1\_A1}; \ldots; \tau_1\ \texttt{child\_1\_k} = \texttt{g\_k\_Aj};\\ \ldots\\ \texttt{funcAj(v1,...,vk);}\\ \tau_j\ \texttt{child\_j\_1} = \texttt{g\_1\_A1}; \ldots; \tau_j\ \texttt{child\_j\_k} = \texttt{g\_k\_Aj};\\ \texttt{g\_1\_A0} = \mathit{enc}_b^1(\texttt{child\_1\_1}, \ldots, \texttt{child\_1\_k})\\ \ldots\\ \texttt{g\_k\_A0} = \mathit{enc}_b^k(\texttt{child\_j\_1}, \ldots, \texttt{child\_j\_k})\end{array}$$

Note that if $b^{(j)}$ is of arity 0—i.e., if $j = 0$—the construction yields k assignments of the form $\texttt{g\_m\_A0} = \mathit{enc}_b^m()$.

The function enc_b^m interprets the semantics of b on the m^{th} input example. We take Linear Integer Arithmetic as an example to illustrate how enc_b^m works.

$$\begin{array}{rlrl}\mathit{enc}^m_{0^{(0)}} &:= \texttt{0} & \mathit{enc}^m_{1^{(0)}} &:= \texttt{1}\\ \mathit{enc}^m_{\texttt{x}^{(0)}} &:= \texttt{vi} & \mathit{enc}^m_{\text{Equals}^{(2)}}(L,R) &:= (L\texttt{=}R)\\ \mathit{enc}^m_{\text{Plus}^{(2)}}(L,R) &:= L\texttt{+}R & \mathit{enc}^m_{\text{Minus}^{(2)}}(L,R) &:= L\texttt{-}R\\ \multicolumn{4}{c}{\mathit{enc}^m_{\text{IfThenElse}^{(3)}}(B,L,R) := \texttt{if}(B)\ L\ \texttt{else}\ R}\end{array}$$

We now turn to the correctness of the construction. First, we formalize the relationship between expression-trees in $L(G)$, the semantics of $P[G, E]$, and the number `n`. Given an expression-tree e, we assume that each node q in e is annotated with the production that has produced that node. Recall that $\delta(A) = \{r_1, \ldots, r_m\}$ is the set of productions with head A (where the subscripts are indexes in some arbitrary, but fixed order). Concretely, for every node q, we assume there is a function $pr(q) = (A, i)$, which associates q with a pair that indicates that non-terminal A produced n using the production r_i (i.e., r_i is the i^{th} production whose left-hand-side non-terminal is A).

We now define how we can extract a number $\#(e)$ for which the program $\hat{P}[\#(e)]$ will exhibit the same semantics as that of the expression-tree e. First, for every node q in e such that $pr(q) = (A, i)$, we define the following number:

$$\#_{nd}(q) = \begin{cases} 1\underbrace{0\cdots0}_{i-1} & \text{if } i < |\delta(A)| \\ \underbrace{0\cdots0}_{i-1} & \text{if } i = |\delta(A)|. \end{cases}$$

The number $\#_{nd}(q)$ indicates what suffix of the value of `n` will cause `funcA` to trigger the code corresponding to production r_i. Let $q_1 \cdots q_m$ be the sequence of nodes visited during a pre-order traversal of expression-tree e. The number corresponding to e, denoted by $\#(e)$, is defined as the bit-vector $\#_{nd}(q_m) \cdots \#_{nd}(q_1)$.

Finally, we add the entry-point of the program, which calls the function `funcS` corresponding to the initial non-terminal S, and contains the assertion that encodes our reachability problem on all the input examples $E = \{c_1, \ldots, c_k\}$.

```
void main(){
  τ_I x1 = c_1; ··· ;τ_I xk = c_k;
  funcS(x1, ..., xk);
  assert ⋁_{1≤i≤k} ¬ψ(f, c_i)[g_i_S/f(x)]; // At least one c_i fails }
```

Correctness. We first need to show that the function $\#(\cdot)$ captures the correct language of expression-trees. Given a non-terminal A, a value n, and input values $i_1, \ldots, i_k$, we use $[\![\texttt{funcA}[n]]\!](i_1, \ldots, i_k) = (o_1, \ldots o_k)$ to denote the values of the variables $\{\texttt{g\_1\_A}, \ldots, \texttt{g\_k\_A}\}$ at the end of the execution of `funcA[n]` with the initial value of `n` $= n$ and input values $x_1, \ldots, x_k$. Given a non-terminal A, we write $L(G, A)$ to denote the set of terms that can be derived starting with A.

Lemma 3. *Let A be a non-terminal, $e \in L(G, A)$ an expression, and $\{i_1, \ldots, i_k\}$ an input set. Then, $([\![e]\!](i_1), \ldots, [\![e]\!](i_k)) = [\![\texttt{funcA}[\#(e)]]\!](i_1, \ldots, i_k)$.*

Each procedure `funcA[n]`$(i_1, \ldots, i_k)$ that we construct has an explicit dependence on variable `n`, where `n` controls the non-deterministic choices made by the `funcA` and procedures called by `funcA`. As a consequence, when relating numbers and expression-trees, there are two additional issues to contend with:

Non-termination. Some numbers can cause `funcA`$[n]$ to fail to terminate. For instance, if the case for "`Start ::= Plus(Start, Start)`" in program

$P[G_2, E_1]$ from Fig. 1 were moved from the first branch (lines (3)–(8)) to the final else case (line (13)), the number $\mathtt{n} = 0 = \ldots 0000000$ (base 2) would cause `Start` to never terminate, due to repeated selections of `Plus` nodes. However, note that the only assert statement in the program is placed at the end of the main procedure. Now, consider a value of n such that $re_{enc(sy,E)}$ is satisfiable. Definition 3 implies that the flow of control will reach and falsify the assertion, which implies that $\mathtt{funcA}[n]$ terminates.[4]

Shared suffixes of sufficient length. In Example 1, we showed how for program $P[G_2, E_1]$ (Fig. 1) the number $\mathtt{n} = 1000101$ (base 2) corresponds to the top-down, left-to-right generation of `Plus(x,1)`. That derivation consumed exactly seven bits; thus, any number that, written in base 2, shares the suffix 1000101—e.g., 11010101000101—will also generate `Plus(x,1)`.

The issue of shared suffixes is addressed in the following lemma:

Lemma 4. *For every non-terminal A and number $\mathtt{n}$ such that $[\![\mathtt{funcA}[n]]\!](i_1, \ldots, i_k) \neq \perp$ (i.e., `funcA` terminates when the non-deterministic choices are controlled by $\mathtt{n}$), there exists a minimal $\mathtt{n}'$ that is a (base 2) suffix of $\mathtt{n}$ for which (i) there is an $e \in L(G)$ such that $\#(e) = n'$, and (ii) for every input $\{i_1, \ldots, i_k\}$, we have $[\![\mathtt{funcA}[n]]\!](i_1, \ldots, i_k) = [\![\mathtt{funcA}[n']]\!](i_1, \ldots, i_k)$.*

We are now ready to state the correctness property of our construction.

Theorem 2. *Given a* SyGuS *problem $sy^E = (\psi_E(f, x), G)$ over a finite set of examples E, the problem sy^E is **realizable** iff $re_{enc(sy,E)}$ is **satisfiable**.*

5 Implementation and Evaluation

NOPE is a tool that can return two-sided answers to unrealizability problems of the form $sy = (\psi, G)$. When it returns ***unrealizable***, no expression-tree in $L(G)$ satisfies ψ; when it returns ***realizable***, some $e \in L(G)$ satisfies ψ; NOPE can also time out. NOPE incorporates several existing pieces of software.

1. The (un)reachability verifier SeaHorn is applied to the reachability problems of the form $re_{enc(sy,E)}$ created during successive CEGIS rounds.
2. The SMT solver Z3 is used to check whether a generated expression-tree e satisfies ψ. If it does, NOPE returns ***realizable*** (along with e); if it does not, NOPE creates a new input example to add to E.

It is important to observe that SeaHorn, like most reachability verifiers, is only sound for ***un***satisfiability—i.e., if SeaHorn returns ***unsatisfiable***, the reachability problem is indeed unsatisfiable. Fortunately, SeaHorn's one-sided

[4] If the SyGuS problem deals with the synthesis of programs for a language that can express non-terminating programs, that would be an additional source of non-termination, different from that discussed in item **Non-termination**. That issue does not arise for LIA SyGuS problems. Dealing with the more general kind of non-termination is postponed for future work.

answers are in the correct direction for our application: to prove unrealizability, NOPE only requires the reachability verifier to be sound for unsatisfiability.

There is one aspect of NOPE that differs from the technique that has been presented earlier in the paper. While SEAHORN is sound for ***un***reachability, it is not sound for reachability—i.e., it cannot soundly prove whether a synthesis problem is realizable. To address this problem, to check whether a given SYGUS problem sy^E is realizable on the finite set of examples E, NOPE also calls the SYGUS solver ESolver [2] to synthesize an expression-tree e that satisfies sy^E.[5]

In practice, for every intermediate problem sy^E generated by the CEGIS algorithm, NOPE runs the ESolver on sy^E and SEAHORN on $re_{enc(sy,E)}$ in *parallel*. If ESolver returns a solution e, SEAHORN is interrupted, and Z3 is used to check whether e satisfies ψ. Depending on the outcome, NOPE either returns ***realizable*** or obtains an additional input example to add to E. If SEAHORN returns ***unsatisfiable***, NOPE returns ***unrealizable***.

Modulo bugs in its constituent components, NOPE is sound for both realizability and unrealizability, but because of Lemma 2 and the incompleteness of SEAHORN, NOPE is not complete for unrealizability.

Benchmarks. We perform our evaluation on 132 variants of the 60 LIA benchmarks from the LIA SYGUS competition track [2]. We do not consider the other SYGUS benchmark track, Bit-Vectors, because the SEAHORN verifier is unsound for most bit-vector operations–e.g., bit-shifting. We used three suites of benchmarks. LIMITEDIF (resp. LIMITEDPLUS) contains 57 (resp. 30) benchmarks in which the grammar bounds the number of times an IfThenElse (resp. Plus) operator can appear in an expression-tree to be 1 less than the number required to solve the original synthesis problem. We used the tool QUASI to automatically generate the restricted grammars. LIMITEDCONST contains 45 benchmarks in which the grammar allows the program to contain only constants that are coprime to any constants that may appear in a valid solution—e.g., the solution requires using odd numbers, but the grammar only contains the constant 2. The numbers of benchmarks in the three suites differ because for certain benchmarks it did not make sense to create a limited variant—e.g., if the smallest program consistent with the specification contains no IfThenElse operators, no variant is created for the LIMITEDIF benchmark. In all our benchmarks, the grammars describing the search space contain infinitely many terms.

Our experiments were performed on an Intel Core i7 4.00 GHz CPU, with 32 GB of RAM, running Lubuntu 18.10 via VirtualBox. We used version 4.8 of Z3, commit 97f2334 of SEAHORN, and commit d37c50e of ESolver. The timeout for each individual SEAHORN/ESolver call is set at 10 min.

Experimental Questions. Our experiments were designed to answer the questions posed below.

EQ 1. Can NOPE prove unrealizability for variants of real SYGUS benchmarks, and how long does it take to do so?

[5] We chose ESolver because on the benchmarks we considered, ESolver outperformed other SYGUS solvers (e.g., CVC4 [3]).

Finding: NOPE*can prove unrealizability for* 59/132 *benchmarks.* For the 59 benchmarks solved by NOPE, the average time taken is 15.59 s. The time taken to perform the last iteration of the algorithm—i.e., the time taken by SEAHORN to return **unsatisfiable**—accounts for 87% of the total running time.

NOPE can solve all of the LIMITEDIF benchmarks for which the grammar allows at most one IfThenElse operator. Allowing more IfThenElse operators in the grammar leads to larger programs and larger sets of examples, and consequently the resulting reachability problems are harder to solve for SEAHORN.

For a similar reason, NOPE can solve only one of the LIMITEDPLUS benchmarks. All other LIMITEDPLUS benchmarks allow 5 or more Plus statements, resulting in grammars that have at least 130 productions.

NOPE can solve all LIMITEDCONST benchmarks because these require few examples and result in small encoded programs.

EQ 2. How many examples does NOPE use to prove unrealizability and how does the number of examples affect the performance of NOPE?

Note that Z3 can produce different models for the same query, and thus different runs of NOPE can produce different sequences of examples. Hence, there is no guarantee that NOPE finds a good sequence of examples that prove unrealizability. One measure of success is whether NOPE is generally able to find a small number of examples, when it succeeds in proving unrealizability.

Finding: Nope used 1 to 9 examples to prove unrealizability for the benchmarks on which it terminated. Problems requiring large numbers of examples could not be solved because either ESolver or times out—e.g., on the problem max4, NOPE gets to the point where the CEGIS loop has generated 17 examples, at which point ESolver exceeds the timeout threshold.

Finding: The number of examples required to prove unrealizability depends mainly on the arity of the synthesized function and the complexity of the grammar. The number of examples seems to grow quadratically with the number of bounded operators allowed in the grammar. In particular, problems in which the grammar allows zero IfThenElse operators require 2–4 examples, while problems in which the grammar allows one IfThenElse operator require 7–9 examples.

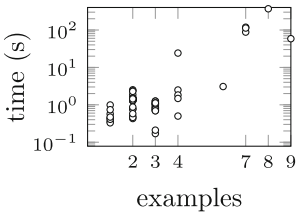

Fig. 3. Time vs examples.

Figure 3 plots the running time of NOPE against the number of examples generated by the CEGIS algorithm. *Finding: The solving time appears to grow exponentially with the number of examples* required to prove unrealizability.

6 Related Work

The SYGUS formalism was introduced as a unifying framework to express several synthesis problems [1]. Caulfield et al. [6] proved that it is undecidable to determine whether a given SYGUS problem is realizable. Despite this negative result,

there are several SyGuS solvers that compete in yearly SyGuS competitions [2] and can efficiently produce solutions to SyGuS problems when a solution exists. Existing SyGuS synthesizers fall into three categories: (*i*) Enumeration solvers enumerate programs with respect to a given total order [7]. If the given problem is unrealizable, these solvers typically only terminate if the language of the grammar is finite or contains finitely many functionally distinct programs. While in principle certain enumeration solvers can prune infinite portions of the search space, none of these solvers could prove unrealizability for any of the benchmarks considered in this paper. (*ii*) Symbolic solvers reduce the synthesis problem to a constraint-solving problem [3]. These solvers cannot reason about grammars that restrict allowed terms, and resort to enumeration whenever the candidate solution produced by the constraint solver is not in the restricted search space. Hence, they also cannot prove unrealizability. (*iii*) Probabilistic synthesizers randomly search the search space, and are typically unpredictable [14], providing no guarantees in terms of unrealizability.

Synthesis as Reachability. CETI [12] introduces a technique for encoding template-based synthesis problems as reachability problems. The CETI encoding only applies to the specific setting in which (*i*) the search space is described by an imperative program with a *finite number* of holes—i.e., the values that the synthesizer has to discover—and (*ii*) the specification is given as a finite number of input-output test cases with which the target program should agree. Because the number of holes is finite, and all holes correspond to values (and not terms), the reduction to a reachability problem only involves making the holes global variables in the program (and no more elaborate transformations).

In contrast, our reduction technique handles search spaces that are described by a grammar, which in general consist of an infinite set of terms (not just values). Due to this added complexity, our encoding has to account for (i) the semantics of the productions in the grammar, and (ii) the use of non-determinism to encode the choice of grammar productions. Our encoding creates one expression-evaluation computation for each of the example inputs, and threads these computations through the program so that each expression-evaluation computation makes use of the *same* set of non-deterministic choices.

Using the input-threading, our technique can handle specifications that contain nested calls of the synthesized program (e.g., $f(f(x)) = x$). (App. A [13]).

The input-threading technique builds a *product program* that performs multiple executions of the same function as done in relational program verification [4]. Alternatively, a different encoding could use multiple function invocations on individual inputs and require the verifier to thread the same bit-stream for all input evaluations. In general, verifiers perform much better on product programs [4], which motivates our choice of encoding.

Unrealizability in Program Synthesis. For certain synthesis problems—e.g., reactive synthesis [5]—the realizability problem is decidable. The framework tackled in this paper, SyGuS, is orthogonal to such problems, and it is undecidable to check whether a given SyGuS problem is realizable [6].

Mechtaev et al. [11] propose to use a variant of SyGuS to efficiently prune irrelevant paths in a symbolic-execution engine. In their approach, for each path π in the program, a synthesis problem p_π is generated so that if p_π is unrealizable, the path π is infeasible. The synthesis problems generated by Mechtaev et al. (which are not directly expressible in SyGuS) are decidable because the search space is defined by a finite set of templates, and the synthesis problem can be encoded by an SMT formula. To the best of our knowledge, our technique is the first one that can check unrealizability of general SyGuS problems in which the search space is an *infinite set of functionally distinct terms*.

Acknowledgment. This work was supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by ONR under grant N00014-17-1-2889; by NSF under grants CNS-1763871 and CCF-1704117; and by the UW-Madison OVRGE with funding from WARF.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 1–8. IEEE (2013)
2. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: SyGuS-Comp 2016: results and analysis. arXiv preprint arXiv:1611.07627 (2016)
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
5. Bloem, R.: Reactive synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), p. 3 (2015)
6. Caulfield, B., Rabe, M.N., Seshia, S.A., Tripakis, S.: What's decidable about syntax-guided synthesis? arXiv preprint arXiv:1510.08393 (2015)
7. ESolver. https://github.com/abhishekudupa/sygus-comp14
8. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: a framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 447–450. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_41
9. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 376–389 (2017)
10. Hu, Q., D'Antoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 386–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_21
11. Mechtaev, S., Griggio, A., Cimatti, A., Roychoudhury, A.: Symbolic execution with existential second-order constraints. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 389–399 (2018)

12. Nguyen, T.V., Weimer, W., Kapur, D., Forrest, S.: Connecting program synthesis and reachability: automatic program repair using test-input generation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 301–318. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_17
13. Qinheping, H., Jason, B., John, C., Loris, D., Reps, T.: Proving unrealizability for syntax-guided synthesis. arXiv preprint arXiv:1905.05800 (2019)
14. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. Commun. ACM **59**(2), 114–122 (2016)