Verifying Infinitely Many Programs at Once

Loris D'Antoni*

University of Wisconsin
Madison WI 53706-1685, USA
loris@cs.wisc.edu

Abstract. In traditional program verification, the goal is to automatically prove whether a program meets a given property. However, in some cases one might need to prove that a (potentially infinite) *set* of programs meets a given property. For example, to establish that no program in a set of possible programs (i.e., a search space) is a valid solution to a synthesis problem specification, e.g., a property φ, one needs to verify that all programs in the search space are incorrect, e.g., satisfy the property $\neg\varphi$. The need to verify multiple programs at once also arises in other domains such as reasoning about partially specified code (e.g., in the presence of library functions) and self-modifying code. This paper discusses our recent work in designing systems for verifying properties of infinitely many programs at once.

1 Introduction

In traditional program verification, the goal is to automatically prove whether **a program** meets a given property. However, in some cases we might need to prove that **a potentially infinite set of programs** meets a given property.

For example, consider the problem of establishing that a program-synthesis problem is *unrealizable* (i.e., has no solution in a given search space of programs) [5,4,9]. To establish unrealizability, i.e., that no program in a set of possible programs (i.e., a search space) is a valid solution to a synthesis problem specification, e.g., a property φ, one needs to verify that all programs in the search space are incorrect, e.g., satisfy the property $\neg\varphi$.

Example 1 (Proving Unrealizability). Consider the synthesis problem sy_{first} where the goal is to synthesize a function f that takes as input a state (x, y), and returns a state where $y = 10$. Assume, however, that the search space of possible programs in sy_{first} is defined using the following grammar G_{first}:

$$Start \to y := E \qquad E \to x \mid E + 1$$

* Work done in collaboration with Qinheping Hu, Jinwoo Kim, and Thomas W. Reps. Supported by NSF under grants CCF-{1750965, 1918211, 2023222}; by a Facebook Research Faculty Fellowship, by a Microsoft Research Faculty Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

Clearly $y := 10 \notin L(Start)$; moreover, all programs in $L(Start)$ are incorrect on at least one input. For example, on the input $x = 15$ every program in the grammar sets y to a value greater than 15. Consequently, sy_{first} is unrealizable.

While it is trivial for a human to establish that sy_{first} is indeed unrealizable, it is definitely not trivial to *automatically verify* that all the infinitely many programs accepted by the grammar G_{first} are incorrect on at least one input.

Another setting where one may want to prove a property about multiple programs is when verifying partial programs—i.e., programs where some components are unknown [10].

Example 2 (Symbolically Executing Partial Programs). Consider the following program `foo` that outputs the difference between the number of elements of an array for which applying a function `f` results in a positive number and the number of elements for which the result of applying `f` is a negative number.

```
1 def foo(f, array):
2     count = 0
3     for i in range(len(array)):
4         if f(array[i]) > 0:
5             count += 1
6         else:
7             count -= 1
8     return count
```

Now assume that we know this program will always receive a function `f` drawn from the following grammar:

$$
\begin{aligned}
F &\to \lambda \texttt{x}.G \\
G &\to \texttt{abs}(H) \mid \texttt{-abs}(H) \\
H &\to \texttt{x} \mid H\texttt{+}H \mid H\texttt{*}H \mid H\texttt{-}H \mid \texttt{abs}(H)
\end{aligned}
$$

We might be interested in symbolically executing the program `foo` to identify feasible paths and understand if the program can be pruned or whether perhaps it is equivalent to a simpler program. One can do so using an uninterpreted function to model the behavior of `f`, but this approach would be imprecise because the grammar under consideration is such that all the infinitely many programs in $L(F)$ either always return a positive number (i.e., `f` is of the form λ`x.abs(`H`)`) or always return a negative number (i.e., `f` is of the form λ`x.-abs(`H`)`). Using an uninterpreted function one would detect that the path that follows the line numbers $[1, 2, 3, 4, 5, 3, 6, 7]$—i.e., the path that reaches line 5 in the first iteration of the loop and line 7 in the second iteration—is feasible, which is a false positive since the witness for `f` for this path is a function that does not have a counterpart in $L(F)$. We would like to devise a verification technique that can symbolically execute `foo` and avoid this source of imprecision.

The examples we showed illustrate how verification sometimes requires one to reason about more than one program at once. In fact, our examples require one to reason about *infinitely many programs* at once! Our work introduced automated techniques to prove properties of infinite sets of programs. First, we designed sound but incomplete automated verification techniques specialized in proving unrealizability of synthesis problems [5,4,9] (Section 2). Second, we designed unrealizability logic [8] a sound and relatively complete Hoare-style logical framework for expressing proofs that infinitely many programs satisfy a pre-post condition pair (Section 3). We conclude this extended abstract with some reflections of the current limitations and directions of our work.

2 Proving Unrealizability of Synthesis Problems

Program synthesis refers to the task of discovering a program, within a given search space, that satisfies a behavioral specification (e.g., a logical formula, or a set of input-output examples) [2,11,1].

While tools are becoming better at synthesizing programs, one property that remains difficult to reason about is the *unrealizability* of a synthesis problem, i.e., the non-existence of a solution that satisfies the behavioral specification within the search space of possible programs. Unrealizability has many applications; for example, one can show that a certain synthesized solution is optimal with respect to some metric by proving that a better solution to the synthesis problem does not exist—i.e., by proving that the synthesis problem where the search space contains only programs of lower cost is unrealizable [6].

In our work, we built tools that can establish unrealizability for several types of synthesis problems. Our tools Nay [5] and Nope [4] can prove unrealizability for syntax-guided synthesis (SyGuS) problems where the input grammar only contains expressions, whereas our tool MESSY [9] can prove unrealizability for semantics-guided synthesis (SemGuS) problems.

Nay The key insight behind Nay is that one can, given a synthesis problem, build a nondeterministic recursive program that always satisfies an assertion if and only if the given problem is unrealizable. For example, for the problem in Example 1, one would build a program like the following to check that no program in the search space of the synthesis problem, when given an input that sets x to 15, can produce an output where y is set to 10.

```
def genE(x, y):
    if nondet(): # nondeterministic guard
        return x  # simulates E -> x
    return genE(x, y) + 1  # simulates E -> E+1
def genStart(x, y):
    return (x, genE(x, y)) # simulates Start -> y:=E
def main():
    genStart(15, nondet()) # if we set x to 15
    assert y != 10 # y is never 10
```

NOPE The key observation of NOPE is that for problems involving only expressions, unrealizability can be checked by determining what set of values $\eta(i)$ a certain set of programs can produce for a given input i, and making sure that the output value we would like our program to produce for input i does not lie in that set $\eta(i)$. For example, for the problem in Example 1, one can define the following equation that computes the possible values $\eta_{Start}(15)$ of x and y, when the input value of x is 15.

$$\begin{aligned}\eta_{Start}(15) &= \{(v_x, v_y) \mid v_y \in \eta_E(15)\}\\ \eta_S(15) &= \{v \mid v = 15 \vee (v' \in \eta_E(15) \wedge v = v' + 1)\}\end{aligned}$$

NOPE can solve this type of equations for a limited set of SYGUS problems using fixed-point algorithms and can then check if $\eta_{Start}(15)$ contains a pair where the second element is 10 (in this case it does not). NOPE can automatically prove unrealizability for many problems involving linear integer (LIA) arithmetic and is in fact sound and complete for conditional linear integer arithmetic (CLIA) when the specification is given as a set of examples—i.e., NOPE provides a decision procedure for this fragment of SYGUS (Theorem 6.2 in [4]). Some of the techniques presented in NOPE have been extended to design specialized unrealizability-checking algorithms for problems involving bit-vector arithmetic [7].

MESSY SEMGUS is a general framework for specifying synthesis problems, which also allows one to define synthesis problems involving, for example, imperative constructs. In SEMGUS, one can specify a synthesis problem by providing a grammar of programs and constrained Horn clauses (CHCs) that describe the semantics of programs in the grammar. For the problem in Example 1, the following CHC can capture the semantics of the assignment $y := e$ using two relations: (i) The relation $Sem_{Start}(p, (x, y), (x', y'))$ holds when evaluating the program p on state (x, y) results in the state (x', y'), and (ii) The relation $Sem_E(e, (x, y), v)$ holds when evaluating expression p on state (x, y) results in the value v.

$$\frac{Sem_E(e, (x, y), v) \quad x' = x \quad y' = v}{Sem_{Start}(y := e, (x, y), (x', y'))}\ y := E \tag{1}$$

Once a problem is modeled with CHCs (i.e., we have semantic rules for all the possible constructs in the language), proving unrealizability can be phrased as a proof search problem. In particular, if we add the following CHC to the set of CHCs defining the semantics, the relation $Solution(p)$ captures all programs that on input $x = 15$ set the value of y to 10.

$$\frac{Sem_{Start}(p, (x, y), (x', y') \quad x = 15 \quad y' = 10}{Solution(p)}\ y := E \tag{2}$$

MESSY then uses a CHC solver to find whether there exists a program p such that $Solution(p)$ is provable using the given set of CHCs. If the answer is no, the problem is unrealizable. MESSY is currently the only automated tool that can (sometimes) prove unrealizability for problems involving imperative programs and could, for example, prove that no imperative program that only uses bitwise *and* and bitwise *or* can implement a bitwise *xor*.

3 Unrealizability Logic

The works we discussed in Section 3 provide automatic techniques to establish that a problem is unrealizable; however, these techniques are all *closed-box*, meaning that they conceal the reasoning behind *why* a synthesis problem is unrealizable. In particular, these techniques typically do not produce a proof artifact that can be independently checked.

Our most recent work presents *unrealizability logic* [8], a Hoare-style proof system for reasoning about the unrealizability of synthesis problems (In this section, we include some excerpts from [8].). In addition to the main goal of reasoning about unrealizability, unrealizability logic is designed with the following goals in mind:

- to be a *general* logic, capable of dealing with various synthesis problems;
- to be amenable to *machine reasoning*, as to enable both automatic proof checking and to open future opportunities for automation;
- to *provide insight* into why certain synthesis problems are unrealizable through the process of completing a proof tree.

Via unrealizability logic, one is able to (i) reason about unrealizability in a principled, explicit fashion, and (ii) produce concrete proofs about unrealizability.

To prove whether the problem in Example 1 is unrealizble, one would use unrealizability logic to derive the following triple, which states that if one starts in a state where $x = 15$, executing any program in the set $L(Start)$ will result in a state where y is different than 10:

$$\{\!|x = 15|\!\}\ L(Start)\ \{\!|y \neq 10|\!\}$$

Unrealizability logic shares much of the intuition behind Hoare logic and its extension toward recursive programs. However, these concepts appearing in Hoare logic alone are insufficient to model unrealizability, which motivated us to develop the new ideas that form the basis of unrealizability logic.

Hoare logic is based on triples that overapproximate the set of states that can be reached by a program s; i.e., the Hoare triple

$$\{P\}\ \mathsf{s}\ \{Q\}$$

asserts that Q is an *overapproximation* of all states that may be reached by executing s, starting from a state in P. The intuition in Hoare logic is that one will often attempt to prove a triple like $\{P\}\ \mathsf{s}\ \{\neg X\}$ for a set of bad states X, which ensures that execution of s cannot reach X.

Unrealizability logic operates on the same overapproximation principle, but differs in two main ways from standard Hoare logic. The differences are motivated by how synthesis problems are typically defined, using two components: (i) a search space S (i.e., a set of programs), and (ii) a (possibly infinite) set of related input-output pairs $\{(i_1, o_1), (i_2, o_2), \cdots\}$.

To reason about *sets* of programs, in unrealizability logic, the central element (i.e., the program s) is changed to a *set* of programs S. The unrealizability-logic triple

$$\{\!|P|\!\}\ S\ \{\!|Q|\!\}$$

thus asserts that Q is an overapproximation of all states that are reachable by executing *any possible combination* of a pre-state $p \in P$ *and* a program $\mathsf{s} \in S$.

The second difference concerns *input-output pairs*: in unrealizability logic, we wish to place the input states in the precondition, and overapproximate the set of states reachable from the input states (through a set of programs) as the postcondition. Unfortunately, the input-output pairs of a synthesis problem cannot be tracked using standard pre- and postconditions; nor can they be tracked using auxiliary variables, because of a complication arising from the fact that unrealizability logic must reason about a *set* of programs—i.e., we want our possible output states to be the results of executing the *same* program on the given input (for all possible programs) and prevent output states where different inputs are processed by different programs in the search space.

To keep the input-output relations in check, the predicates of unrealizability logic talk instead about (potentially infinite) *vector-states*, which are sets of states in which each individual state is associated with a unique index—e.g., variable x of the state with index i is referred to as x_i. We defer the reader to the original unrealizability logic paper for these details [8].

The proof system for unrealizability logic has sound underpinnings, and provides a way to build proofs of unrealizability similar to the way Hoare logic [3] provides a way to build proofs that a given program cannot reach a set of bad states. Furtheremore the systems is relatively complete in the same sense as Hoare logic is.

4 Conclusions

This paper outlines recent advances in reasoning about infinite sets of programs at once. We presented techniques for proving unrealizability of synthesis problems that draw inspiration from traditional program verification. However, such techniques did not provide ways to produce proof artifact and to address this limitation, we discussed *unrealizability logic*, the first proof system for overapproximating the execution of an infinite set of programs. This logic is also the first approach that allows one to prove unrealizability for synthesis problems that require infinitely many inputs to be proved unrealizable.

The name "unrealizability logic" is perhaps misleading as the logic allows one to reason about many properties beyond unrealizability. The fact that unrealizability logic is both sound and relatively complete means that this proof system can prove (given powerful enough assertion languages) any property of a given set of programs expressed as a grammar. For example, the problem given in Example 2 of identifying whether a symbolic execution path is infeasible can be phrased as proving whether an unrealizability triple holds.

It is thus natural to conclude with two open questions: (*i*) What applications besides unrealizability can benefit from unrealizability logic as a proof system? (*ii*) Can unrealizability logic be automated in the same successful way Hoare logic has been automated for traditional program verification?

References

1. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. ACM SIGPLAN Notices **50**(6), 229–239 (2015)
2. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices **46**(1), 317–330 (2011)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
4. Hu, Q., Breck, J., Cyphert, J., D'Antoni, L., Reps, T.: Proving unrealizability for syntax-guided synthesis. In: International Conference on Computer Aided Verification. pp. 335–352. Springer (2019)
5. Hu, Q., Cyphert, J., D'Antoni, L., Reps, T.: Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1128–1142 (2020)
6. Hu, Q., D'Antoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: International Conference on Computer Aided Verification. pp. 386–403. Springer (2018)
7. Kamp, M., Philippsen, M.: Approximate bit dependency analysis to identify program synthesis problems as infeasible. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 353–375. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_16, `https://doi.org/10.1007/978-3-030-67067-2_16`
8. Kim, J., D'Antoni, L., Reps, T.: Unrealizability logic. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). https://doi.org/10.1145/3571216, `https://doi.org/10.1145/3571216`
9. Kim, J., Hu, Q., D'Antoni, L., Reps, T.: Semantics-guided synthesis. Proceedings of the ACM on Programming Languages **5**(POPL), 1–32 (2021)
10. 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. pp. 389–399 (2018)
11. Phothilimthana, P.M., Elliott, A.S., Wang, A., Jangda, A., Hagedorn, B., Barthels, H., Kaufman, S.J., Grover, V., Torlak, E., Bodik, R.: Swizzle inventor: data movement synthesis for gpu kernels. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 65–78 (2019)