Program Synthesis For Verification in UCLID5

FEDERICO MORA, University of California, Berkeley
ELIZABETH POLGREEN, University of California, Berkeley
SANJIT A. SESHIA, University of California, Berkeley

This document describes synthesis in UCLID5, UCLID5's intermediate synthesis language, and UCLID5's interface to external synthesis engines.

1 VERIFICATION WITH HOLES.

Formal verification is a time consuming and tedious task. Verification users need to model their system, annotate it thoroughly, and formally specify its behavior. UCLID5 alleviates this burden on verification users through program synthesis. In UCLID5, users can declare unknown functions and use these functions in models, specifications, or annotations. At verification time, UCLID5 will automatically find implementations for these functions that achieve the desired verification goal.

Take for example the UCLID5 model in Fig. 1. The (hypothetical) user wants to prove the invariant `a_le_b` using induction, but finds that the proof fails. At this point the user has two options: they can either manually strengthen the inductive argument, or they can ask UCLID5 to automatically do this for them. Fig. 2 demonstrates the latter. In this example, the user specifies a function to synthesize called `h` and conjuncts the existing invariant with a call to `h`. Given this model, UCLID5 will automatically generate the function `h(x, y) = x >= 0`, which completes the induction proof.

2 INTERMEDIATE SYNTHESIS LANGUAGE.

We propose a new intermediate synthesis language, SYNTH-LIB, tailored to verification with holes. The language is SMT-LIB, but with one extra command taken from SYGUS-IF: `synth-blocking-fun`. The syntax for the command is

(`synth-blocking-fun` ⟨fname⟩ ((⟨argname⟩ ⟨argsort⟩)*) ⟨rsort⟩ ⟨grammar⟩?)

where ⟨fname⟩ is the name of the function, ⟨argname⟩ is the name of an argument, ⟨argsort⟩ is the sort of the corresponding argument, there are zero or more arguments, ⟨rsort⟩ is the sort returned by the function, and ⟨grammar⟩ is an optional syntactic specification for the function body.

The semantics of SYNTH-LIB is exactly that of SMT-LIB when no function to synthesize is on the assertion stack. When the assertion stack does contain a function to synthesize, the meaning of the command `check-sat` differs. In particular, `check-sat` will return `sat` if for each function to synthesize on the assertion stack the solver finds a function that adheres to the grammar, and the corresponding SMT-LIB query with these function plugged-in returns `unsat`. These functions are the satisfying model: the `get-model` command will return them. If no such functions exist, then `check-sat` will return `unsat`. If the solver is unable to conclude `sat` or `unsat`, it will return `unknown`. Intuitively, a SYNTH-LIB query with a single `synth-blocking-fun` declaration asks "is there a function that makes this underlying SMT-LIB query unsatisfiable?"

Authors' addresses: Federico Mora, University of California, Berkeley, fmora@berkeley.edu; Elizabeth Polgreen, University of California, Berkeley, epolgreen@berkeley.edu; Sanjit A. Seshia, University of California, Berkeley, sseshia@eecs.berkeley.edu.

2019. 2475-1421/2019/10-ART1 $15.00
https://doi.org/

3 FROM UCLID5 TO SYNTH-LIB

Uclid5, without synthesis, encodes each property violation as a separate smt-lib query. Let's call the i^{th} property violation $P_i(x_0 \dots x_m)$, and take the i^{th} smt-lib query to be checking the validity of $\exists x_0 \dots x_m \ P_i(x_0 \dots x_m)$, where P_i contains no free variables. If any $\exists x_0 \dots x_m \ P_i(x_0 \dots x_m)$ is valid, then Uclid5 determines that the verification failed. In other words, verification succeeds iff $\neg\exists x_0 \dots x_m \ \bigvee_{i=0}^{i=n} P_i(x_0 \dots x_m)$ is valid (we can do this step because existential quantifiers distribute over disjunction). This is the encoding into smt-lib that most verification tools use.

With synthesis, Uclid5 checks a version of this expression directly rather than checking each property individually. More specifically, Uclid5 constructs a synth-lib query representing the validity of $\exists f \ \neg\exists x_0 \dots x_m \ \bigvee_{i=0}^{i=n} P_i(f, x_0 \dots x_m)$, where each P_i encodes a property violation that may refer to a function to synthesize, f. That is, all we have to do to enable synthesis is declare and use an unknown function.

Fig. 3 and Fig. 4 show two examples. Fig. 3 shows the synth-lib query for the verification by induction of the Uclid5 program in Fig. 1—it is the same as an smt-lib query because there is no function to synthesize. Fig. 4 shows the synth-lib query for verification by induction of the Uclid5 program in Fig. 2. A synthesis engine might solve the query in Fig. 4 by finding the function `(define-fun h ((x Int) (y Int)) Bool (>= x 0))`. We know this is a correct solution because the corresponding smt-lib query, displayed in Fig. 5, is unsatisfiable.

4 INTERFACE TO SYNTHESIS ENGINES.

Unfortunately, existing synthesis engines do not support synth-lib, so we have to encode the query in sygus-if. We do this with these four rewrite rules

$$\text{(assert } a\text{)} \rightarrow \text{(constraint (not } a\text{))} \tag{1}$$

$$\text{(declare-fun } a \text{ (}s_0 \dots s_{n-1}\text{) } s_n\text{)} \rightarrow \text{(declare-var } a \ s_0 \text{ -> ... -> } s_n\text{)} \tag{2}$$

$$\text{synth-blocking-fun} \rightarrow \text{synth-fun} \tag{3}$$

$$\text{check-sat} \rightarrow \text{check-synth.} \tag{4}$$

The first rewrite rule is the most important. It implements the following equivalence

$$\exists f \ \neg\exists x_0 \dots x_m \ \bigvee_{i=0}^{i=n} P_i(f, x_0 \dots x_m) \equiv \exists f \ \forall x_0 \dots x_m \ \bigwedge_{i=0}^{i=n} \neg P_i(f, x_0 \dots x_m).$$

The left hand side of the equivalence corresponds to the kind of queries that synth-lib solves, whereas the right hand side corresponds to the kind of queries that sygus-if solves.

5 FIGURES

```
module main {
  // Part 1: System description.
  var a, b : integer;

  init {
    a = 0;
    b = 1;
  }
  next {
    a', b' = b, a + b;
  }

  // Part 2: System specification.
  invariant a_le_b: a <= b;

  // Part 3: Proof script.
  control {
    induction;
    check;
    print_results;
  }
}
```

Fig. 1. Uclid5 Fibonacci model

```
module main {
  synthesis function h(x : integer, y : integer) : boolean;
  var a, b : integer;

  init {
    a = 0;
    b = 1;
  }
  next {
    a', b' = b, a + b;
  }

  // Part 2: System specification.
  invariant a_le_b: a <= b && h(a, b);

  // Part 3: Proof script.
  control {
    induction;
    check;
    print_results;
  }
}
```

Fig. 2. Uclid5 Fibonacci model with a hole

```
(declare-fun initial_b () Int)
(declare-fun initial_a () Int)
(declare-fun new_a () Int)
(declare-fun new_b () Int)

(assert
(or
    (<= 0 1)
    (and
        (<= initial_a initial_b)
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b))
        (not (<= new_a new_b)))))

(check-sat)
```

Fig. 3. Induction query for Fig. 1

```
(synth-blocking-fun h ((x Int) (y Int)) Bool)

(declare-fun initial_b () Int)
(declare-fun initial_a () Int)
(declare-fun new_a () Int)
(declare-fun new_b () Int)

(assert
(or
    (not (h 0 1))
    (and
        (and (<= initial_a initial_b) (h initial_a initial_b))
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b))
        (not (and (<= new_a new_b) (h new_a new_b))))))

(check-sat)
```

Fig. 4. Induction query of Fig. 2

```
(define-fun h ((x Int) (y Int)) Bool (>= x 0))

(declare-fun initial_b () Int)
(declare-fun initial_a () Int)
(declare-fun new_a () Int)
(declare-fun new_b () Int)

(assert
(or
    (not (h 0 1))
    (and
        (and (<= initial_a initial_b) (h initial_a initial_b))
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b))
        (not (and (<= new_a new_b) (h new_a new_b))))))

(check-sat)
```

Fig. 5. Solution to Fig. 4

```
(synth-fun h ((x Int) (y Int)) Bool)

(declare-var initial_b Int)
(declare-var initial_a Int)
(declare-var new_a Int)
(declare-var new_b Int)

(constraint (not
(or
    (not (h 0 1))
    (and
        (and (<= initial_a initial_b) (h initial_a initial_b))
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b))
        (not (and (<= new_a new_b) (h new_a new_b)))))))

(check-synth)
```

Fig. 6. SYGUS-IF for Fig. 4