Synthesis in UCLID5

Federico Mora
University of California, Berkeley

Kevin Cheang
University of California, Berkeley

Elizabeth Polgreen
University of California, Berkeley

Sanjit A. Seshia
University of California, Berkeley

Abstract

We describe an integration of program synthesis into UCLID5, a formal modelling and verification tool. To the best of our knowledge, the new version of UCLID5 is the only tool that supports program synthesis with bounded model checking, k-induction, sequential program verification, and hyperproperty verification. We use the integration to generate 25 program synthesis benchmarks with simple, known solutions that are out of reach of current synthesis engines, and we release the benchmarks to the community.

1 Introduction

Formal verification can be a time-consuming task that requires significant manual effort. Especially for complex systems, users often need to manually provide, for example, loop invariants, function summaries, or environment models. Synthesis has the potential to alleviate some of this manual burden [16]. For example, prior work has used synthesis to reason about program loops [5], and to automate program repair [9]. We believe this is a promising direction, but, for it to make a real impact, verification tools need to offer flexible synthesis integration, generic support for proof procedures, and a capable synthesis engine back-end.

In this work, we primarily address the first two requirements. Specifically, we integrate program synthesis into the UCLID5 [14] formal modelling and verification tool by allowing users to declare functions to synthesize and to use these functions freely. While UCLID5 has previously supported use of synthesis, it only supported invariant synthesis through a special command that was independent of verification. We use the new synthesis integration to generate 25 benchmarks from existing verification tasks. These benchmarks have small solutions, but are out of reach for current synthesis engines. We hope that they will help the synthesis engine development effort, particularly for syntax-guided synthesis [1].

Illustrative Example. Consider the UCLID5 model in Fig. 1, which represents a Fibonacci sequence. The (hypothetical) user wants to prove by induction that the invariant `a_le_b` at line 13 always holds. Unfortunately, the proof fails because the invariant is not inductive. Without synthesis, the user would need to manually strengthen the invariant until it became inductive. However, the user can ask UCLID5 to automatically do this for them. Fig. 1 demonstrates this on lines 16, 17 and 18. Specifically, the user specifies a function to synthesize called `h` at lines 16 and 17, and then uses `h` at line 18 to strengthen the existing set of invariants. Given this input, UCLID5, using e.g. CVC4 [2] as a synthesis engine, will automatically generate the function `h(x, y) = x >= 0`, which completes the inductive proof.

In this example, the function to synthesize represents an inductive invariant. However, functions to synthesize are treated exactly like any interpreted function in UCLID5: the user could have called `h` anywhere in the code. Furthermore, this example uses induction and a global invariant, however, the user could also have used a linear temporal logic (LTL) specification and bounded model checking (BMC). In this sense, our integration is fully flexible and generic.

Contributions. We present an integration of synthesis into the verification tool UCLID5, allowing users to generate program synthesis queries for unknown parts of a system they wish to verify. The integration is a natural extension of the existing UCLID5 language, and to the best of our knowledge, is the first to support program synthesis with bounded model checking, k-induction, sequential program verification, and hyperproperty verification. The synthesis queries UCLID5 generates are in the standard SYGUS-IF [12] specification language. We use this tool to generate a 25 SYGUS-IF synthesis benchmarks from existing verification queries and release these benchmarks to the community.

2 Related work

Program sketching [17] synthesizes expressions to fill holes in programs, and has subsequently been applied to program repair [8, 9]. UCLID5 aims to be more flexible than this work, allowing users to declare unknown functions even in the verification annotations, as well as supporting multiple verification algorithms and types of properties. Rosette [18] provides support for synthesis and verification, but the synthesis is limited to bounded specifications of sequential programs, whereas UCLID5 can also synthesize programs that satisfy unbounded specifications, by using proof procedures like induction. Formal synthesis algorithms have been used to assist in verification tasks, such as safety and termination of loops [5], and generating invariants [7, 19], but none of this work to-date integrates program synthesis fully into an existing verification tool. Before this new synthesis integration, UCLID5 supported synthesis of inductive invariants. The key insight of this work is to generalize the synthesis

```
module main {
  // Part 1: System Description.
  var a, b : integer;
  init {
    a, b = 0, 1;
  }
  next {
    a', b' = b, a + b;
  }

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

  // Part 3: (NEW) Synthesis Integration
  synthesis function
    h(x : integer, y : integer): boolean;
  invariant hole: h(a, b);

  // Part 4: Proof Script.
  control {
    induction;
    check;
    print_results;
  }
}
```

Figure 1. UCLID5 Fibonacci model. Part 3 shows the new synthesis syntax, and how to find an auxiliary invariant.

support, and to unify all synthesis tasks in UCLID5 by reusing the verification back-end.

3 From Verification to Program Synthesis

In this section, we give the necessary background on program synthesis, and the existing verification techniques inside of UCLID5. We then describe how we combine the two to realize synthesis in UCLID5.

3.1 Program Synthesis

The program synthesis problem corresponds to the second-order query

$$\exists f\ \forall \vec{x} \sigma(f, \vec{x}),$$

where f is the function to synthesize, $\vec{x}$ is the set of all possible inputs, and σ is the specification to be satisfied.

3.2 Verification in UCLID5

At at high level, UCLID5 takes in a model, generates a set of verification conditions, asks a satisfiability modulo theory (SMT) solver [4] to check the verification conditions, and then returns the results to the user. This process is the same regardless of the proof procedure used. The important point, is that UCLID5 encodes the violation of each independent verification condition as a separate SMT-LIB query.

Let $P_i(\vec{x})$ encode the i^{th} verification condition, and take the i^{th} SMT-LIB query to be checking the validity of $\exists \vec{x}\ \neg P_i(\vec{x})$, where P_i contains no free variables. We say that there is

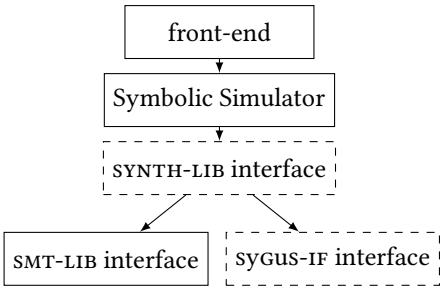

Figure 2. Overview of synthesis in UCLID5. Dashed blocks indicate blocks introduced for the new synthesis integration.

a counterexample to the i^{th} verification query if the query $\exists \vec{x}\ \neg P_i(\vec{x})$ is valid. Verification of a model with n verification conditions succeeds *iff* there are no counter-examples:

$$\forall \vec{x} \bigwedge_{i=0}^{i=n} P_i(\vec{x}).$$

3.3 Synthesis Encoding in UCLID5

Given a UCLID5 model in which the user has declared a function to synthesize, f, we wish to construct a synthesis query that is satisfied *iff* there is an f for which all the verification conditions pass for all possible inputs. We build this synthesis query by taking the conjunction of the negation of all the verification queries. Specifically, we check the validity of

$$\exists f\ \forall \vec{x} \bigwedge_{i=0}^{i=n} P_i(f, \vec{x})$$

where each P_i encodes a verification condition that may refer to the function to synthesize, f. Note the similarity between this query and the standard program synthesis formulation: the specification for synthesis, σ, from the equation in Sec. 3.1, is now replaced with the conjunction of all the verification conditions. With this observation, to enable synthesis for any verification procedure in UCLID5, all we do is let users declare and use functions to synthesize.

4 Implementation

The UCLID5 verification tool is constructed as shown in Figure 2. An input UCLID5 model is parsed by the front-end into an abstract syntax tree. From this abstract syntax tree, a symbolic simulator generates an assertion stack that contains an assertion for each verification condition. Prior to our work, assertions were then passed to an SMT-LIB interface which converted the assertions to SMT-LIB and called a solver. The new UCLID5 instead uses a new intermediate representation, SYNTH-LIB, that is easily passed to either an SMT solver or a synthesis engine. This architecture allows us to use the same code that generates verification queries for synthesis.

```
1  (synth-blocking-fun h ((x Int) (y Int)) Bool)
2  ;(define-fun h ((x Int) (y Int)) Bool (>= x 0))
3  (declare-fun initial_b () Int)
4  (declare-fun initial_a () Int)
5  (declare-fun new_a () Int)
6  (declare-fun new_b () Int)
7  (assert (or
8   (not (and (<= initial_a initial_b) (h 0 1)))
9   (and
10   (and (<= initial_a initial_b) (h initial_a
       initial_b))
11   (= new_a initial_b)
12   (= new_b (+ initial_a initial_b))
13   (not (and (<= new_a new_b) (h new_a new_b))))))
14 (check-sat)
```

Figure 3. SYNTH-LIB induction query of Fig. 1

The SYNTH-LIB representation is SMT-LIB [3], but with one extra command borrowed from SYGUS-IF [12]. The syntax for the new 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. 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?"

Fig. 3 shows the SYNTH-LIB query corresponding to the Fibonacci model in Fig. 1. A synthesis engine might solve the query in Fig. 3 by finding the function `h(x, y) = x >= 0`. This is a correct solution because the corresponding SMT-LIB query—which we can get by commenting out line 1 of Fig. 3 and uncommenting line 2—is unsatisfiable.

The semantics of SYNTH-LIB is exactly that of SMT-LIB when no function to synthesize is on the assertion stack, and assertions are passed directly to the SMT solver. When the assertion stack contains a function to synthesize, UCLID5 applies the following four rewrite rules to convert SYNTH-LIB into SYGUS-IF:

1. (assert a) ⟶ (constraint (not a))
2. (declare-fun a ($s_0 \ldots s_{n-1}$) s_n ⟶ (declare-var a s_0 ->...-> s_n)
3. synth-blocking-fun ⟶ synth-fun
4. check-sat ⟶ check-synth

The first rewrite rule is the most important: it implements the following equivalence

$$\exists f\ \neg\exists \vec{x}\ \bigvee_{i=0}^{i=n} \neg P_i(f,\vec{x}) \equiv \exists f\ \forall \vec{x}\ \bigwedge_{i=0}^{i=n} P_i(f,\vec{x}),$$

where the left hand side is the form of queries in SYNTH-LIB, and the right hand side is the corresponding query in SYGUS-IF. The source code for UCLID5 is available online [15].

5 Benchmark Suite

The integration of synthesis into UCLID5 allows us to generate synthesis benchmarks from any UCLID5 verification task. We thus present a set of 25 benchmarks with known, small solutions that are out of reach of existing synthesis solvers. These benchmarks use induction, BMC, LTL specifications, and sequential code. To conform to the SYGUS-IF language, we limited ourselves to bit-vector, integer, array, and boolean data-types, and did not use verification tasks that required quantifiers. All benchmarks are available online [11].

The benchmarks come from four different sources. Four benchmarks come from a simplified model of the Two Phase Commit protocol, written in P [6]; three benchmarks come from Sahai at al's [13] work on hyperproperty verification; six benchmarks come from UCLID5's documentation; and the remaining 12 benchmarks come from models used in UC Berkeley's EECS 219C course. In all cases, we constructed the benchmarks by replacing small parts of either auxiliary invariants or parts of existing code with functions to synthesize. 12 benchmarks come from models that use induction, and 13 from models that use LTL specifications and BMC. All 25 benchmarks are difficult for existing state-of-the-art engines, but are a reasonable target for synthesis engines.

6 Conclusions and Future Work

We have presented an integration of synthesis into the UCLID5 verification tool, allowing users to generate synthesis queries for unknown parts of a system they wish to verify. This integration is compatible with all verification algorithms currently supported by UCLID5, and generates synthesis queries in the standard SYGUS-IF format.

In the future, we intend to apply synthesis in UCLID5 to the verification of distributed systems written in P. Prior work has been successfully in finding invariants for bounded distributed systems, and then generalizing the invariants to the unbounded setting [10]. We plan to explore these approaches with UCLID5 now that we can easily switch between synthesis using e.g. BMC and k-induction.

Acknowledgments

This work was supported in part by NSF grants 1739816 and 1837132, a gift from Intel under the SCAP program, SRC Task 2867.001, and the iCyPhy center.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *FMCAD*. 1–17.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV*. 171–177.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[4] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, Armin Biere, Hans van Maaren, and Toby Walsh (Eds.). Vol. 4. IOS Press, Chapter 8.

[5] Cristina David, Daniel Kroening, and Matt Lewis. 2015. Using Program Synthesis for Program Analysis. In *LPAR*. Springer, 483–498.

[6] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *PLDI* (2013), 321–332.

[7] Grigory Fedyukovich and Rastislav Bodík. 2018. Accelerating Syntax-Guided Invariant Synthesis. In *TACAS (1)*. Springer, 251–269.

[8] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *ICSE*. ACM, 12–23.

[9] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *ESEC/SIGSOFT FSE*. ACM, 593–604.

[10] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *OSDI*. ACM, 370–384.

[11] Federico Mora. 2020. UCLID5 Synthesis Benchmarks. https://github.com/FedericoAureliano/synthesis-benchmarks

[12] Mukund Raghothaman and Abhishek Udupa. 2014. Language to Specify Syntax-Guided Synthesis Problems. `https://sygus.org/assets/pdf/SyGuS-IF.pdf`.

[13] Shubham Sahai, Rohit Sinha, and Pramod Subramanyan. 2020. Verification of Quantitative Hyperproperties Using Trace Enumeration Relations. In *CAV*.

[14] Sanjit Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating modeling, verification, synthesis, and learning. In *MEMOCODE*.

[15] Sanjit Seshia and Pramod Subramanyan. 2020. UCLID5: A system for modeling, verification, and synthesis of computational systems. https://github.com/uclid-org/uclid

[16] Sanjit A. Seshia. [n.d.]. Combining Induction, Deduction, and Structure for Verification and Synthesis. *IEEE*.

[17] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*. Springer, 4–13.

[18] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *Onward!* ACM, 135–152.

[19] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing Environment Invariants for Modular Hardware Verification. In *VMCAI*. Springer, 202–225.