Edinburgh Research Explorer

UCLID5: Multi-Modal Formal Modeling, Verification, and Synthesis

UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis

Elizabeth Polgreen[1,2]([✉]) , Kevin Cheang[1]([✉]), Pranav Gaddamadugu[1],
Adwait Godbole[1] , Kevin Laeufer[1] , Shaokai Lin[1] , Yatin A. Manerkar[3],
Federico Mora[1] , and Sanjit A. Seshia[1]

[1] UC Berkeley, Berkeley, USA
`elizabeth.polgreen@ed.ac.uk`
[2] University of Edinburgh, Edinburgh, UK
[3] University of Michigan, Ann Arbor, USA

Abstract. UCLID5 is a tool for the multi-modal formal modeling, verification, and synthesis of systems. It enables one to tackle verification problems for heterogeneous systems such as combinations of hardware and software, or those that have multiple, varied specifications, or systems that require hybrid modes of modeling. A novel aspect of UCLID5 is an emphasis on the use of syntax-guided and inductive synthesis to automate steps in modeling and verification. This tool paper presents new developments in the UCLID5 tool including new language features, integration with new techniques for syntax-guided synthesis and satisfiability solving, support for hyperproperties and combinations of axiomatic and operational modeling, demonstrations on new problem classes, and a robust implementation.

1   Overview

Tools for formal modeling and verification are typically specialized for particular domains and for particular methods. For instance, software verification tools like Boogie [4] focuses on modeling sequential software and Floyd-Hoare style reasoning, while hardware verifiers like ABC [5] are specialized for sequential circuits and SAT-based equivalence and model checking. Specialization makes sense when the problems fit well within a homogeneous problem domain with specific verification needs. However, there is an emerging class of problems, such as in security and cyber-physical systems (CPS), where the systems under verification are heterogeneous, or the types of specifications to be verified are varied, or there is not a single type of model that is effective for verification. An example of such a problem is the verification of trusted computing platforms [37] that involve hardware and software components working in tandem, and where the properties to be checked include invariants, refinement checks, and hyperproperties. There is a need for automated formal methods and tools to handle this class of problems.

UCLID5 is a system for *multi-modal* formal modeling, verification, and synthesis that addresses the above need. UCLID5 is multi-modal in three important ways. First, it permits different modes of modeling, using axiomatic and

operational semantics, or as combinations of concurrent transition systems and procedural code. This enables modeling systems with multiple characteristics. Second, it offers a varied suite of specification modes, including first-order formulas in a combination of logical theories, temporal logic, inline assertions, pre- and post-conditions, system invariants, and hyperproperties. Third, it supports the first two capabilities with a varied suite of verification techniques, including Floyd-Hoare style proofs, k-induction and bounded model checking (BMC), verifying hyperproperties, or using syntax-guided and inductive synthesis to provide more automation in tedious steps of verification, or to automate the modeling process (as proposed in [34]).

The UCLID5 framework was first proposed in 2018 [35], itself a major evolution of the much older UCLID system [6], one of the first satisfiability modulo theories (SMT) based modeling and verification tools. Since that publication [35], which laid out the vision for the tool and described a preliminary implementation, the utility of the tool has been demonstrated on several problem classes (e.g., [7,8,25]), such as for verifying security across the hardware-software interface. The syntax has been extended and state-of-the-art methods for syntax-guided synthesis (SyGuS) have also been integrated into the tool [28], including new capabilities for satisfiability and synthesis modulo oracles [32]. This tool paper presents an overview of the latest version of UCLID5, highlighting novel multi-modal aspects of the tool, as well as the new features supported since 2018 [35]. The paper is structured as follows: in Sect. 2 we give an overview of the UCLID5 tool; in Sect. 3 we detail different multi-modal aspects of the tool, as well as high-lighting new features; and in Sect. 4 we present a case study using UCLID5 to verify a Trusted Abstract Platform. We cover related work in Sect. 5. The new features we highlight are:

1. Fully integrated support for synthesis across all verification modes
2. Support for modeling with external oracles, via satisfiability and synthesis modulo oracles [32]
3. New language features to support combining axiomatic and operational modeling
4. Direct support for hyperproperties
5. Front-end translations from Chisel/FIRRTL to UCLID5, and from RISC-V binaries to UCLID5, referenced in Sect. 6.
6. New case studies: covering models for distributed CPS in Lingua Franca [23], and encodings of μhb specifications and verification of a Trusted Abstract Platform described in Sects. 3.2 and 4 and in the corresponding artifact [31].

2   Overview of UCLID5

In verification mode, UCLID5 reduces the question of whether a model satisfies a given specification to a set of constraints that can be solved by an off-the-shelf SMT solver. In synthesis mode, UCLID5 reduces the problem of finding an interpretation for an uninterpreted function such that the specification is satisfied into a SyGuS problem that can be solved by an off-the-shelf SyGuS solver. In order to do so, UCLID5 performs the following main tasks, as shown in Fig. 1:

Front End: UCLID5 takes models written in the UCLID5 language as input. The command-line front-end allows user configuration, including specifying the external SMT-solver/SyGuS-solver to be used, as well as enabling certain utilities such as automatically converting uninterpreted functions to arrays. The parser builds an abstract syntax tree from the model.

AST Passes: UCLID5 performs a number of transformations and checks on the abstract syntax tree, including type-checking and inlining of procedures. This intermediate representation supports limited control flow such as if-statements and switch-cases, but loops are not permitted in procedural code and are removed via unrolling (bounded for-loops) or replacement with user-provided invariants (while loops). However, unbounded control flow can be handled by representation as transition systems (where each module consists of a transition system with an initial and a next block, each represented as a separate AST).

Symbolic Simulator: The symbolic simulator performs a simulation of the transition system in the model, according to the verification command provided, and produces a set of assertions. For instance, if bounded model checking is used, UCLID5 will symbolically execute the main module a bounded number of times. UCLID5 encodes the violation of each independent verification condition as a separate assertion tree.

Synth-Lib Interface: UCLID5 supports both synthesis and verification. The Synth-Lib interface constructs either a verification or a synthesis problem from the assertions generated by the symbolic simulator. The verification problems are passed to the SMT-LIB interface, which converts each assertion in UCLID5's intermediate representation to an assertion in SMT-LIB. Similarly, the synthesis problems are passed to the SyGuS-IF interface, which converts each assertion to an assertion in SyGuS-IF. The verification and synthesis problems are then passed to the appropriate provided external solver and the result is reported back to the user.
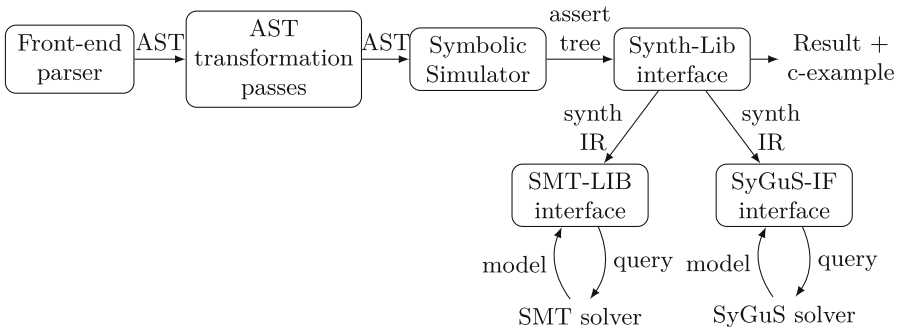

Fig. 1. Architecture of UCLID5

Basic UCLID5 Models. A simple UCLID5 model that computes the Fibonacci sequence is shown in Fig. 2. UCLID5 models are contained within modules which comprise of 3 parts: a system model represented using combinations of sequential, concurrent, operational and axiomatic modeling, as described in Sects. 3.2; a system specification described in Sect. 3.1; and a proof script that specifies the verification tasks UCLID5 should perform to prove that the system satisfies its specification, using a variety of supported verification and synthesis techniques described in Sect. 3.1.

3   Multi-modal Language Features

3.1   Multi-modal Verification and Synthesis

Specification. UCLID5 supports a variety of different types of specifications. The standard properties supported include inline assertions and assumptions in sequential code, pre-conditions and post-conditions for procedures, and global axioms and invariants (both as propositional predicates, and temporal invariants in Linear Temporal Logic (LTL)).

The latest version of UCLID5 further provides direct support for hyperinvariants and hyperaxioms (for k-safety). This new support for direct hyperproperties comprises of two new language constructs: hyperaxiom and hyperinvariant. The former places an assumption on the behavior of the module, if n instances of the module were instantiated, and the latter is an invariant over n instances of the module, which is verified via the usual verification methods. A variable x from the nth instance of the module is reasoned about in the predicate using $x.n$, and the number of modules instantiated is determined by the maximum n in both the invariant and the axiom. For example, `hyperinvariant[2] det_xy: y.1 == y.2` asserts that a 2-safety hyperproperty holds.

Verification. To verify these specifications, we implement multiple classic techniques. As a result, once a model is written in UCLID5, the user can deploy a combination of verification techniques, depending on the properties targeted. UCLID5 supports a range of verification techniques including: Bounded Model Checking (for LTL, hyperinvariants and assertion-based properties); induction and k-induction for assertion-based invariants and hyperinvariants; and verification of pre-and post-conditions on procedures and hyperinvariants.

As an exemplar of the utility of multi-modal verification, consider the hyperproperty based models verified by Sahai et al. [33]. These models use both procedure verification and induction to verify k-trace properties.

Synthesis. The latest version of UCLID5 integrates program synthesis fully across all the verification modes previously described. Specifically, users are able to declare and use *synthesis functions* anywhere in their models, and UCLID5 will seek to automatically synthesize function bodies for these functions such that the user-selected verification task will pass. In this section, we give an illustrative example of synthesis in UCLID5, we provide the necessary background on program synthesis, and then we formulate the existing verification techniques inside of UCLID5 for synthesis.

```
1  module main {
2
3    // Part 1: System Description.
4    var a, b : integer;
5    init {
6      a, b = 0, 1;
7    }
8    next {
9      a', b' = b, a + b;
10   }
11
12   // Part 2: System Specification.
13   invariant a_le_b: a <= b;
14
15   // Part 3: (NEW) Synthesis Integration
16   synthesis function
17     h(x : integer, y : integer): boolean;
18   invariant hole: h(a, b);
19
20   // Part 4: Proof Script.
21   control {
22     induction;
23     check;
24     print_results;
25   }
26 }
```

Fig. 2. UCLID5 Fibonacci model. Part 3 shows the new synthesis syntax, and how to find an auxiliary invariant.

Consider the UCLID5 model in Fig. 2. The 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. Figure 2 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. CVC5 [3] as a syntax-guided 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. Furthermore, the integration scheme allows us to enable synthesis for any verification procedure in UCLID5, by simply letting users declare and use functions to synthesize and relying on existing SyGuS-IF solvers to carry out the automated reasoning.

3.2   Multi-modal Modeling

Combining Concurrent and Sequential Modeling. A unique feature of the UCLID5 modeling language is the ability to easily combine sequential and concurrent modeling. This allows a user to easily express models representing

sequential programs, including standard control flow, procedure calls, sequential updates, etc., in a sequential model, and to combine these components within a system designed for concurrent modeling based on transition systems. The sequential program modeling is inspired by systems such as Boogie [4] and allows the user to port Boogie models to UCLID5. The concurrent modeling is done by defining transition systems with a set of initial states and a transition relation. Within UCLID5, each module is a transition system. A main module can be defined that triggers when each child module is stepped. For an example of this combination of sequential and concurrent modeling, we refer the reader to the CPU example presented in the original UCLID5 paper [35], which uses concurrent modules to instantiate multiple CPU modules, modeled as transition systems, with sequential code to model the code that executes instructions, and to the case study in Sect. 4.

Reasoning with External Oracles. New in the latest version, UCLID5 supports the modeling with *oracle function symbols* [32] in both verification and synthesis. Namely, a user can include "oracle functions" in any UCLID5 model, where an oracle function is a function without a provided implementation, but which is associated to a user-provided external binary that can be queried by the solver. We note that oracle functions (and functions in general) can only be first-order within the UCLID5 modeling language, i.e., functions cannot receive functions as arguments.

This support is useful in cases where some components of the system are difficult or impossible to model, but could be compiled into a binary that the solver can query; or where the model of the system would be challenging for an SMT solver to reason about (for instance, highly non-linear arithmetic), and it may be better to outsource that reasoning to an external binary.

UCLID5 supports oracle function symbols in verification by interfacing with a solver that supports Satisfiability Modulo Theories and Oracles (SMTO) [32], and in synthesis by interfacing with a solver that supports Synthesis Modulo Oracles (SyMO) [32].

Oracle function symbols are declared like functions, with the keyword `oracle`, and an annotation pointing to the binary implementation. For instance `oracle function [isprime] Prime (x: integer): boolean` would indicate to the solver that the binary `isprime` takes an integer as input and returns a boolean. This is translated into the corresponding syntax in SMTO or SyMO, as detailed in [30].

An exemplar of such reasoning in a synthesis file is available in the artifact [31], where we use UCLID5 to synthesize a safe and stabilizing controller for a Linear Time Invariant system, similar to Abate et al. [1].

Combining Operational and Axiomatic Modeling. UCLID5 can model a system being verified using an operational (transition system-based) approach, as Fig. 2 shows. However, UCLID5 also supports modeling a system in an *axiomatic* manner, whereby the system is specified as a set of properties over traces. Any execution satisfying the properties is allowed by the system, and

any execution violating the properties is disallowed. Axiomatic modeling can provide order-of-magnitude performance improvements over operational models in certain cases [2], and is often well suited to systems with large amounts of non-determinism. We provide an example of fully axiomatic modeling in the artifact [31].

However, uniquely, UCLID5 allows users to specify multi-modal systems using a combination of operational and axiomatic modeling. In such models, some constraints on the execution are enforced by the initial state and transition relation (operational modeling), while others are enforced through axiomatic invariants (axiomatic modeling). This allows the user to choose the mode of modeling most appropriate to each constraint. For example, the ILA-MCM work [39] combined operational ILA (Instruction Level Abstraction) models to describe the functional behavior of processing elements with memory consistency model (MCM) orderings that are more naturally specified axiomatically [2]. (MCM orderings constrain shared-memory communication and synchronization between multiple processing elements.) The combined model, used for System-on-Chip verification, worked by sharing variables (called "facets") between both the models. UCLID5 makes it much easier to perform such a combination.

Figure 3 depicts parts of a UCLID5 model of microarchitectural execution that uses both operational and axiomatic modeling (similar to that from the ILA-MCM work), based on the μspec specifications of COATCheck [24]. In this model, the steps of instruction execution are driven by the `init` and `next` blocks, i.e., the operational component of the model. Multiple instructions can step at any time (`curTime` denotes the current time in the execution), but they can only take one step per timestep. Meanwhile, axioms such as the `fifoFetch` axiom enforce ordering *between* the execution of multiple instructions. The `fifoFetch` axiom specifically enforces that instructions in program order on the same core must be fetched in program order. (Enforcing this order is tricky using operational modeling alone). The transition rules and axioms operate over the same data structures, ensuring that executions of the final model abide by both sets of constraints.

μspec models routinely function by grounding quantifiers over a finite set of instructions. Thus, to fully support μspec axiomatic modeling, we introduce two new language features —namely, *groups* and *finite quantifiers*. A group is a set of objects of a single type. A group can have any number of elements, but it must be finite, and the group is immutable once created. For instance, the group `testInstrs` in Fig. 3 consists of four instructions. Finite quantifiers, meanwhile, are used to quantify over group elements.

This example showcases UCLID5's highly flexible multi-modal modeling capability. Models can be purely operational, purely axiomatic, or a combination of the two. Note that axiomatic modeling relies on the new language features `finite_forall` and `groups`. For a further example of axiomatic and operational multi-modal modeling, we refer the reader to the case study checking reachability properties in reactive embedded systems described in the artifact [31].

```
1  module main {
2      <type declarations>
3      var i1, i2, i3, i4 : microop_t;
4      <set i1-i4 to be the instructions of a test, like mp>
5      group testInstrs : microop_t = {i1, i2, i3, i4};
6
7      //Vars to decide which instrs to step and when.
8      var next1, next2, next3, next4 : boolean;
9      var curTime : integer;
10
11     init {
12         i1.Fetch.nExists = false; i1.Execute.nExists = false;
13         <...>
14     }
15     //Axiom enforcing that instructions are fetched in order.
16     axiom fifoFetch :
17         finite_forall (i : microop_t) in testInstrs ::
18         finite_forall (j : microop_t) in testInstrs ::
19         (ProgramOrder(i, j) && NodeExists(j.Fetch)) ==>
20             EdgeExists(i.Fetch, j.Fetch);
21
22     procedure stepInst(index : integer)
23       returns (instr_next : microop_t)
24     {
25         //Steps instr@index, unless it has completed.
26         case
27             (index == 1) : {
28                 instr_next = i1;
29                 if (!instr_next.Fetch.nExists) {
30                     instr_next.Fetch.nExists = true;
31                     instr_next.Fetch.nTime = curTime;
32                 } else {
33                     <...>
34         esac
35     }
36     next {
37         //Increment the current timestamp and
38         //nondeterministically step instructions.
39         curTime' = curTime + 1;
40         havoc next1, next2, next3, next4;
41
42         if (next1) { call (i1') = stepInst(1); }
43         if (next2) { call (i2') = stepInst(2); }
44         if (next3) { call (i3') = stepInst(3); }
45         if (next4) { call (i4') = stepInst(4); }
46     }
47 }
```

Fig. 3. UCLID5 model that incorporates both operational modeling (through the `init` and `next` blocks) and axiomatic modeling (through the `axiom` keyword).

4   Case Study: TAP Model

The final case study we wish to describe verifies a model of a trusted execution environment. Trusted execution environments [10,11,17,20] often provide a software interface for users to execute enclaves, using hardware primitives to enforce memory isolation. In contrast to software which requires reasoning about sequential code, hardware modeling uses a paradigm that permits concurrent updates to a system. Moreover, verifying hyperproperties such as integrity requires reasoning about multiple instances of a system which most existing tools are not well suited for. In this section, we present the UCLID5 port[1] of the Trusted

---

[1] https://github.com/uclid-org/trusted-abstract-platform/.

Abstract Platform (TAP) which was originally[2] written in Boogie and introduced by Subramanyan et al. [37] to model an abstract idealized trusted enclave platform. We demonstrate how UCLID5's multi-model support alleviates the difficulties in modeling the TAP model in existing tools.

```
 1  module tap {
 2    // State variable declarations
 3    var tap_enclave_metadata_valid: tap_enclave_metadata_valid_t;
 4    var tap_enclave_metadata_addr_map: tap_enclave_metadata_addr_map_t;
 5    ...
 6    // Enclave operations
 7    procedure launch(eid: tap_enclave_id_t, ...) { ... }
 8    ...
 9    init { ... } // initialize TAP
10    next { // step the system
11      case
12        (tap_current_mode == mode_untrusted) : {
13          call (...) = AdversarialStep(...);
14        }
15        (tap_current_mode == mode_enclave) : {
16          call (...) = EnclaveStep(...);
17        }
18      esac
19    }
20  }
21
22  module integrity_proof {
23    // Create two instances of the TAP model
24    instance tap_1: tap(...);
25    instance tap_2: tap(...);
26    // Example invariant: Memory that is mapped are equal between the two traces
27    invariant equal_mem: (forall (pa : wap_addr_t) ::
28      e_excl_map[pa] ==> (tap_1.mem[pa] == tap_2.mem[pa]));
29    ...
30    init { ... } // initialize proof
31    next { // step the system
32      next(tap_1); next(tap_2);
33    }
34    control {
35      v = induction;
36      check;
37    }
38  }
```

Fig. 4. UCLID5 transition system-styled model of TAP and the integrity proof.

Modeling the TAP and Proving Integrity. The UCLID5 model of TAP in Fig. 4 demonstrates some of UCLID5's key features: the enclave operations of the TAP model (e.g. `launch`) are implemented as procedures, and a transition relation of the TAP is defined using a next block that either executes an untrusted adversary operation or the trusted enclave, which in turn executes one of the enclave operations atomically. Proving the integrity hyperproperty on the TAP thus only requires two instantiations of the TAP model, specifying the integrity invariants, and defining a next block which steps each of the TAP instances as shown in the `integrity_proof` module. The integrity proof in UCLID5 uses inductive model checking.

---
[2] https://github.com/0tcb/TAP.

Results and Statistics of the TAP Modules. Table 1 shows the approximate size of the TAP model in both Boogie and UCLID5. #pr, #fn, #an, and #ln refer to the number of procedures, functions, annotations, and lines of code respectively. Annotations are the number of loop invariants, assertions, assumptions, pre- and post-conditions

Table 1. Boogie vs UCLID5 Model Results

Model/Proof	Size				Verif.
	#pr	#fn	#an	#ln	Time (s)
Boogie					
TAP	22	25	254	1840	51
Integrity	14	11	71	835	346
UCLID5					
TAP	53	25	87	2765	49
Integrity	2	0	54	293	30

that were manually specified. The verification time includes compilation and solving.

While the #ln for the TAP model in UCLID5 is higher than that of the model in Boogie due to stylistic differences, the crucial difference is in the integrity proof. The original model in Boogie implements the TAP model and integrity proof as procedures, where the transition of the TAP model is implemented as a while loop. However, this lack of support for modeling transition systems introduces duplicate state variables in a hyperproperty such as integrity, requires context switching and additional procedures for the new variables, which makes the model difficult to maintain and self composition unwieldy. In UCLID5, the proof is no longer implemented as a procedure, but rather, we create instances of the TAP model. We also note that the number of annotations is less in UCLID5 compared to Boogie for the TAP model and proof. Additionally, this model lends itself for more direct verification of hyperproperties.

The verification results are run on a machine with 2.6GHz 6-Core Intel Core i7 and 16GB of RAM running OSX. As shown on the right of Table 1, the verification runtimes between the Boogie and UCLID5 models and proofs are comparable.

5   Related Work

There are a multitude of verification and synthesis tools related to UCLID5. In this brief review, we highlight prominent examples and contrast them with UCLID5 along the key language features described in Sect. 3.

UCLID5 allows users to combine sequential and concurrent modeling (see Sect. 3.2). Most existing tools primarily support either sequential, e.g. [4,21,38], or concurrent computation modeling, e.g. [5,9,14,26,27]. Although users of these systems can often overcome the tool's modeling focus by manually including support for different computation paradigms, for example, Dafny can be used to model concurrent systems [22], this is not always straightforward, and limited support for different paradigms can manifest as limitations in downstream applications. For example, the Serval [29] framework, based on Rosette, cannot reason about concurrent code. UCLID5, to the best of our knowledge, is the only verification tool natively supporting modeling with external oracles.

UCLID5 supports different kinds of specifications and verification procedures (see Sect. 3.1). Most existing tools [5,9,21] do not support multi-modal verification at all. Tools that do offer multi-modal verification do not offer the same range of options as UCLID5. For example, [26] does not support linear temporal logic, and [13,27] does not support hyperproperty verification.

Finally, UCLID5 supports a generic integration with program synthesis (see Sect. 3.1), and so related work includes a number of synthesis engines. The SKETCH system [36] synthesizes expressions to fill holes in programs, and has subsequently been applied to program repair [16,19]. UCLID5 is more flexible than this work, and allows users to declare unknown functions even in the verification annotations, as well as supporting multiple verification algorithms and types of properties. Rosette [38] provides support for synthesis and verification, but, unlike UCLID5, the synthesis is limited to bounded specifications of sequential programs and external synthesis engines are not supported. Synthesis algorithms have been used to assist in verification tasks, such as safety and termination of loops [12], and generating invariants [15,40], but none of this work to-date integrates program synthesis fully into an existing verification tool. Before the new synthesis integration, UCLID5 supported synthesis of inductive invariants. The key insight of this work is to generalize the synthesis support, and to unify all synthesis tasks by re-using the verification back-end.

6   Software Project

The source code for UCLID5 is made publicly available under a BSD-license[3]. UCLID5 is maintained by the UCLID5 team[4], and we welcome patches from the community. Additional front-ends are available for UCLID5, including translators from Firrtl [18][5], and RISC-V binaries[6] to UCLID5 models. An artifact incuding the code for the case studies in this paper is available [31].

References

1. Abate, A., et al.: Automated formal synthesis of provably safe digital controllers for continuous plants. Acta Informatica **57**(1-2), 223–244 (2020)
2. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data-mining for weak memory. ACM Trans. Programm. Lang. Syst. (TOPLAS) 36, July 2014

---

[3] https://github.com/uclid-org/uclid.
[4] https://github.com/uclid-org/uclid/blob/master/CONTRIBUTORS.md.
[5] https://github.com/uclid-org/chiselucl.
[6] https://github.com/uclid-org/riscverifier.

3. Barbosa, H., et al.: CVC5: a versatile and industrial-strength SMT solver. In: TACAS (1), vol. 13243, pp. 415–442. Springer (2022)
4. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
5. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
6. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_7
7. Cheang, K., Rasmussen, C., Lee, D., Kohlbrenner, D., Asanović, K., Seshia, S.A.: Verifying RISC-V physical memory protection (2020)
8. Cheang, K., Rasmussen, C., Seshia, S.A., Subramanyan, P.: A formal approach to secure speculation. In: Proceedings of the Computer Security Foundations Symposium (CSF), June 2019
9. Cimatti, A., Roveri, M., Sheridan, D.: Bounded verification of past LTL. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 245–259. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30494-4_18
10. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch. **2016**, 86 (2016)
11. Costan, V., Lebedev, I., Devadas, S.: Sanctum: minimal hardware extensions for strong software isolation. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 857–874. USENIX Association, Austin, TX (2016)
12. David, C., Kroening, D., Lewis, M.: Using program synthesis for program analysis. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 483–498. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_34
13. Dill, D.L.: The Murphi verification system. In: CAV (1996)
14. Dutertre, B., Jovanović, D., Navas, J.A.: Verification of fault-tolerant protocols with Sally. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NASA Formal Methods, pp. 113–120. Springer, Cham (2018)
15. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 251–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_14
16. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: ICSE, pp. 12–23. ACM (2018)
17. Intel: Intel trust domain extensions (2020). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html
18. Izraelevitz, A., et al.: Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 209–216, November 2017
19. Le, X.D., Chu, D., Lo, D., Goues, C.L., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: ESEC/SIGSOFT FSE, pp. 593–604. ACM (2017)
20. Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., Song, D.: Keystone: an open framework for architecting trusted execution environments. In: EuroSys, pp. 38:1–38:16. ACM (2020)

21. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20

22. Leino, K.R.M.: Modeling concurrency in Dafny. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2017. LNCS, vol. 11174, pp. 115–142. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02928-9_4

23. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Trans. Embed. Comput. Syst. **20**(4), 36:1–36:27 (2021)

24. Lustig, D., Sethi, G., Martonosi, M., Bhattacharjee, A.: Coatcheck: verifying memory ordering at the hardware-os interface. In: ASPLOS, pp. 233–247. ACM (2016)

25. Magyar, A., Biancolin, D., Koenig, J., Seshia, S.A., Bachrach, J., Asanovic, K.: Golden Gate: Bridging the resource-efficiency gap between ASICs and FPGA prototypes. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 1–8, November 2019

26. Mann, M., et al.: `Pono`: a flexible and extensible SMT-based model checker. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 461–474. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_22

27. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 190–202. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_12

28. Mora, F., Cheang, K., Polgreen, E., Seshia, S.A.: Synthesis in UCLID5. CoRR abs/2007.06760 (2020)

29. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with serval. In: SOSP, pp. 225–242. ACM (2019)

30. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.1 (2014). https://sygus.org/assets/pdf/SyGuS-IF.pdf

31. Polgreen, E., et al.: UCLID5 artifact. https://doi.org/10.5281/zenodo.6557711. https://doi.org/10.5281/zenodo.6557711

32. Polgreen, E., Reynolds, A., Seshia, S.A.: Satisfiability and synthesis modulo oracles. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 263–284. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_13

33. Sahai, S., Subramanyan, P., Sinha, R.: Verification of quantitative hyperproperties using trace enumeration relations. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 201–224. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_11

34. Seshia, S.A.: Combining induction, deduction, and structure for verification and synthesis. Proc. IEEE **103**(11), 2036–2051 (2015)

35. Seshia, S.A., Subramanyan, P.: UCLID5: integrating modeling, verification, synthesis and learning. In: MEMOCODE, pp. 1–10. IEEE (2018)

36. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_3

37. Subramanyan, P., Sinha, R., Lebedev, I.A., Devadas, S., Seshia, S.A.: A formal foundation for secure remote execution of enclaves. In: CCS, pp. 2435–2450. ACM (2017)

38. Torlak, E., Bodík, R.: Growing solver-aided languages with rosette. In: Onward!, pp. 135–152. ACM (2013)

39. Zhang, H., Trippel, C., Manerkar, Y.A., Gupta, A., Martonosi, M., Malik, S.: ILA-MCM: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In: FMCAD, pp. 1–10 (2018)
40. Zhang, H., Yang, W., Fedyukovich, G., Gupta, A., Malik, S.: Synthesizing environment invariants for modular hardware verification. In: Beyer, D., Zufferey, D. (eds.) VMCAI 2020. LNCS, vol. 11990, pp. 202–225. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_10