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

Permalink
https://escholarship.org/uc/item/0c79w0z8

ISBN
978-3-031-13184-4

Authors
Polgreen, Elizabeth
Cheang, Kevin
Gaddamadugu, Pranav
et al.

Publication Date
2022

DOI
10.1007/978-3-031-13185-1_27

Peer reviewed

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[1,3], Federico Mora[1], and
Sanjit A. Seshia[1]

[1] UC Berkeley
[2] University of Edinburgh
[3] University of Michigan

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 [6] 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 [43] 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 use synthesis to automate the modeling process.

The UCLID5 framework was first proposed in 2018 [41], itself a major evolution of the much older UCLID system [7], one of the first satisfiability modulo theories (SMT) based modeling and verification tools. Since that publication [41], 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., [9, 33, 18, 8]), 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 [36], including new capabilities for satisfiability and synthesis modulo oracles [39]. 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 [41]. The paper is structured as follows: in Section 2 we give an overview of the UCLID5 tool; in Section 3 we detail different multi-modal aspects of the tool, as well as high-lighting new features; and in Section 4 we present a case study using UCLID5 to verify a Trusted Abstract Platform. We cover related work in Section 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 [39]
3. New language features to support combining axiomatic and operational modeling
4. Direct support for hyper-properties
5. Front-end translations from Chisel/FirRTL to UCLID5, and from RISC-V binaries to UCLID5, referenced in Section 6.
6. New case studies: covering models for distributed CPS in Lingua Franca [29], and encodings of μhb specifications and verification of a Trusted Abstract Platform described in Sections 3.2 and 4 and Appendix A.1.

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 Figure 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 simulation of 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 reported back to the user.

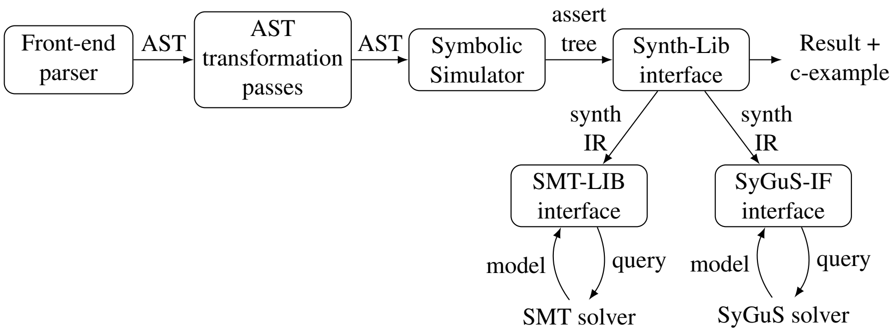


Fig. 1: Architecture of UCLID5

Basic UCLID5 Models A simple UCLID5 model that computes the Fibonacci sequence is shown in Figure 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 Sections 3.2; a system specification described in Section 3.1; and a proof script which instructs UCLID5 how to go about proving that the system satisfies its specification, using a variety of supported verification and synthesis techniques described in Section 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 n^{th} 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 hyper-property based models verified by Sahai et al. [40]. 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 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.

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

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

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

the invariant until it became inductive. However, the user can ask UCLID5 to automatically do this for them. Fig. 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 [5] 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 allows scheme lets us 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 via defining transition systems, similar to NuSMV [10], defining 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 [41], 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 Section 4.

Reasoning with External Oracles New in the latest version, UCLID5 supports the modeling with *oracle function symbols* [39] 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 (SMT0) [39], and in synthesis by interface with a solver that supports Synthesis Modulo Oracles (SyMO) [39].

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 [38].

As an exemplar of such reasoning in a synthesis file is available in Appendix A.3, 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 Figure 2 shows. However, UCLID5 also supports modeling a system in an *axiomatic* manner, whereby the system is specified as a set of invariants. Any execution satisfying the invariants is allowed by the system, and any execution violating the invariants is disallowed. Axiomatic modeling can provide order-of-magnitude performance improvements over operational models in certain cases [3], and is often well suited to systems with large amounts of non-determinism. We provide an example of fully axiomatic modeling in Appendix A.2.

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 [46] 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 [3]. (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.

Fig 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 [32]. 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. Thus, if executing `mp` (Figure 6a), this axiom will prevent executions where e.g. the `Fetch` stage of `i2` happens before the `Fetch` stage of `i1`. (Such an execution would be allowed by the operational component 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. It 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 Figure 7 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 in Appendix A.1.

4 Case Study: TAP model

The final case study we wish to describe verifies a model of a trusted execution environment. Trusted execution environments [24, 13, 12, 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 mod-

```
module main {
    <type declarations>
    var i1, i2, i3, i4 : microop_t;
    <set i1-i4 to be the instructions of a test, like mp>
    group testInstrs : microop_t = {i1, i2, i3, i4};

    //Vars to decide which instrs to step and when.
    var next1, next2, next3, next4 : boolean;
    var curTime : integer;

    init {
        i1.Fetch.nExists = false; i1.Execute.nExists = false;
        <...>
    }
    //Axiom enforcing that instructions are fetched in order.
    axiom fifoFetch :
        finite_forall (i : microop_t) in testInstrs ::
        finite_forall (j : microop_t) in testInstrs ::
        (ProgramOrder(i, j) && (NodeExists(i.Fetch) || NodeExists(j.Fetch))) ==>
            EdgeExists(i.Fetch, j.Fetch);

    procedure stepInst(index : integer)
      returns (instr_next : microop_t)
    {
        //Steps instr@index, unless it has completed.
        case
            (index == 1) : {
                instr_next = i1;
                if(!instr_next.Fetch.nExists) {
                    instr_next.Fetch.nExists = true;
                    instr_next.Fetch.nTime = curTime;
                } else {
                <...>
        esac
    }
    next {
        //Increment the current timestamp and
        //nondeterministically step instructions.
        curTime’ = curTime + 1;
        havoc next1, next2, next3, next4;

        if (next1) { call (i1’) = stepInst(1); }
        if (next2) { call (i2’) = stepInst(2); }
        if (next3) { call (i3’) = stepInst(3); }
        if (next4) { call (i4’) = stepInst(4); }
    }
}
```

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

eling uses a paradigm that permits concurrent updates to a system. Moreover, verifying hyper properties 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 [4] of the Trusted Abstract Platform (TAP) which was originally[5] written in Boogie and introduced by Subramanyan et. al. [43] to model an abstract idealized

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

[5] https://github.com/0tcb/TAP

trusted enclave platform. We demonstrate how UCLID5's multi-model support alleviates the difficulties in modeling the TAP model in existing tools.

Modeling the TAP and Proving Integrity Figure 9 shows a model of TAP (with details omitted for brevity) that 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.

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

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

Table 1: Boogie vs UCLID5 Model Results

While the #ln for the TAP model in UCLID5 is higher than that of the model in Boogie due to stylistic changes, 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 hyper property 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. Additionally, this model lends itself for more direct verification of hyper-properties.

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 Section 3.

UCLID5 allows users to combine sequential and concurrent modeling (see Section 3.2). Most existing tools primarily support either sequential, e.g. [25, 4, 44], or concurrent computation modeling, e.g. [6, 11, 35, 16, 34]. 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 [26], this is not always straightforward, and limited support for different paradigms can manifest as limitations in downstream applications. For example, the Serval [37] 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 Section 3.1). Most existing tools[25, 6, 11] 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, [34] does not support linear temporal logic, and [35, 15] does not support hyper property verification.

Finally, UCLID5 supports a generic integration with program synthesis (see Section 3.1), and so relate work includes a number of synthesis engines. The SKETCH system [42] synthesizes expressions to fill holes in programs, and has subsequently been applied to program repair [23, 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 [44] 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 [14], and generating invariants [17, 47], 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[6]. UCLID5 is maintained by the UCLID5 team [7], and we welcome patches from the community. Additional front-ends are available for UCLID5, including translators from Firrtl [21, 27][8], and RISC-V binaries [9] to UCLID5 models.

Acknowledgments: The UCLID5 project is grateful for the significant contributions by the late Pramod Subramanyan, one of the original creators of the tool.

[6] `https://github.com/uclid-org/uclid`

[7] `https://github.com/uclid-org/uclid/blob/master/CONTRIBUTORS.md`

[8] `https://github.com/uclid-org/chiselucl`

[9] `https://github.com/uclid-org/riscverifier`

References

1. Alessandro Abate, Iury Bessa, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica*, 57(1-2):223–244, 2020.
2. Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
3. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36, July 2014.
4. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.
5. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.
6. Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
7. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. LNCS 2404, pages 78–92, July 2002.
8. Kevin Cheang, Cameron Rasmussen, Dayeol Lee, David Kohlbrenner, Krste Asanović, and Sanjit A. Seshia. Verifying RISC-V physical memory protection. 2020.
9. Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, June 2019.
10. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer, 1999.
11. Alessandro Cimatti, Marco Roveri, and Daniel Sheridan. Bounded verification of past ltl. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pages 245–259. Springer, 2004.
12. Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.
13. Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.
14. Cristina David, Daniel Kroening, and Matt Lewis. Using program synthesis for program analysis. In *LPAR*, pages 483–498. Springer, 2015.
15. David L. Dill. The murphi verification system. In *CAV*, 1996.
16. Bruno Dutertre, Dejan Jovanović, and Jorge A. Navas. Verification of fault-tolerant protocols with sally. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods*, pages 113–120, Cham, 2018. Springer International Publishing.
17. Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In *TACAS (1)*, pages 251–269. Springer, 2018.
18. Pranav Gaddamadugu. Formally verifying trusted execution environments with uclid5. Master's thesis, EECS Department, University of California, Berkeley, Aug 2021.
19. Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *ICSE*, pages 12–23. ACM, 2018.

20. Intel. Intel trust domain extensions, 2020.
21. A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, Nov 2017.
22. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 28(9):690–691, 1979.
23. Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *ESEC/SIGSOFT FSE*, pages 593–604. ACM, 2017.
24. Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.
25. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
26. K. Rustan M. Leino. Modeling concurrency in dafny. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems*, pages 115–142, Cham, 2018. Springer International Publishing.
27. Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.
28. Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Reactors: A deterministic model for composable reactive systems. In *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*, volume LNCS 11971, page 27. Springer-Verlag, 2019.
29. Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):36:1–36:27, 2021.
30. Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(4):Article 36, May 2021.
31. Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *47th International Symposium on Microarchitecture (MICRO)*, 2014.
32. Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. Coatcheck: Verifying memory ordering at the hardware-os interface. In *ASPLOS*, pages 233–247. ACM, 2016.
33. Albert Magyar, David Biancolin, John Koenig, Sanjit A. Seshia, Jonathan Bachrach, and Krste Asanovic. Golden Gate: Bridging the resource-efficiency gap between ASICs and FPGA prototypes. In *In Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, November 2019.
34. Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. Pono: A flexible and extensible smt-based model checker. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 461–474, Cham, 2021. Springer International Publishing.
35. Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 190–202, Cham, 2020. Springer International Publishing.

36. Federico Mora, Kevin Cheang, Elizabeth Polgreen, and Sanjit A. Seshia. Synthesis in uclid5. *CoRR*, abs/2007.06760, 2020.

37. Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

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

39. Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. Satisfiability and synthesis modulo oracles. *CoRR*, abs/2107.13477, 2021. To appear in VMCAI 2022.

40. Shubham Sahai, Pramod Subramanyan, and Rohit Sinha. Verification of quantitative hyperproperties using trace enumeration relations. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2020.

41. Sanjit A. Seshia and Pramod Subramanyan. UCLID5: integrating modeling, verification, synthesis and learning. In *MEMOCODE*, pages 1–10. IEEE, 2018.

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

43. Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *CCS*, pages 2435–2450. ACM, 2017.

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

45. Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960, 2018.

46. Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. ILA-MCM: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.

47. Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. Synthesizing environment invariants for modular hardware verification. In *VMCAI*, pages 202–225. Springer, 2020.

A Further examples

A.1 Case Study: Checking reachability properties in reactive embedded systems

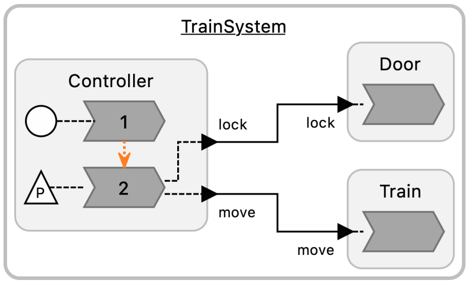


Fig. 4: A simple train system.

We further demonstrate the modeling flexibility of UCLID5 via a case study of checking reachability properties in reactive embedded systems written in a coordination language called Lingua Franca (LF), which allows users to compose reactive components called *reactors* [28, 30]. LF adopts discrete event semantics in which events are processed in timestamp order. Blocks of application code, named *reactions* (denoted in chevrons), can be activated by *triggers*, which include startup (circle), physical action (triangle labeled by "P"), and ports (dark solid triangle). Figure 4 shows the diagram of a train door system with three reactors (`Controller`, `Door`, and `Train`). The driver pressing a button provides the physical action, which triggers reaction 2 in `Controller`. The reaction then outputs signals to close the door (by triggering the reaction in `Door`) and to move the train (by triggering the reaction in `Train`). Using a combination of axiomatic and operational modeling enabled by UCLID5, we can check whether the system permits an unsafe behavior where the train moves before the door closes.

The UCLID5 snippet in Figure 5 illustrates this hybrid modeling approach. The axiomatic segment sits above the `next` block (line 1-7), specifying the semantics of reactors that should hold throughout the execution. Inside the `next` block, the `havoc` statement (line 9) sets the `state` variable in the next transition to a nondeterministic value. The `case` block (line 10-23) stores the new states in the appropriate state variables and sets boolean flags `doorCloses` and `trainMoves`. The `assume` statements (line 24-39) constrain the next nondeterministic value of `state` to one that complies with the language semantics. The operational modeling using the `next` block simplifies the specification of constraints including non-decreasing time tags (line 25), unique event per tag (line 27), reaction priority (line 29), connection delay (line 31-36), and trigger mechanism (line 38-40). The reachability property, "the train does not move when the door is open," can then be checked using proof by induction (line 44).

```
// All timestamps and microsteps are nonnegative.
axiom(pi1(g(state)) >= 0 && pi2(g(state)) >= 0);
// Each state can either be NULL or a specific id.
axiom(id(ctrl1State) == NULL || id(ctrl1State) == controller_1);
axiom(id(ctrl2State) == NULL || id(ctrl2State) == controller_2);
axiom(id(trainState) == NULL || id(trainState) == train_1);
axiom(id(doorState)  == NULL || id(doorState)  == door_1);
next {
    havoc state; // Update state
    case
        (id(state) == door_1) : { doorState’ = state; doorCloses’ = true; }
        (id(state) == train_1) : { trainState’ = state; trainMoves’ = true; }
        (id(state) == controller_1) : { ctrl1State’ = state; }
        (id(state) == controller_2) : { ctrl2State’ = state; }
    esac
    // Time tags do not decrement.
    assume(tag_same(g(state’), g(state)) || tag_later(g(state’), g(state)));
    // An event triggers only once in an instant.
    assume(g(state) == g(state’) ==> id(state) != id(state’));
    // Reaction with higher priority triggers first.
    assume((g(state) == g(state’) && same_reactor(id(state), id(state’))) ==>
     priority(id(state)) < priority(id(state’)));
    // Connection delay
    assume((id(ctrl1State) != NULL && id(ctrl2State’) != NULL)
        ==> tag_diff(g(ctrl2State’), g(ctrl1State)) == zero());
    assume((id(ctrl2State) != NULL && id(trainState’) != NULL)
        ==> tag_diff(g(trainState’), g(ctrl2State)) == zero());
    assume((id(ctrl2State) != NULL && id(doorState’) != NULL)
        ==> tag_diff(g(doorState’), g(ctrl2State)) == zero());
    // Trigger mechanism
    assume(id(ctrl1State) == NULL ==> id(ctrl2State’) == NULL);
    assume(id(ctrl2State) == NULL ==> id(trainState’) == NULL);
    assume(id(ctrl2State) == NULL ==> id(doorState’) == NULL);
}
property p: !(trainMoves && !doorCloses);
control {
    v = induction;
    check;
    print_results;
    v.print_cex;
}
```

Fig. 5: The UCLID5 model for the train system in Figure 4

A.2 Case Study: Fully axiomatic modeling

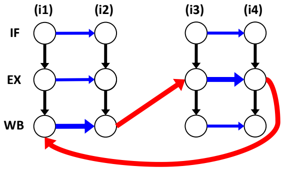
In the rest of this section, we provide an example of how to encode fully axiomatic models in UCLID5, specifically the μspec specifications of COATCheck [32].

Program executions on microarchitectures (component-level models of hardware) can be represented as microarchitectural happens-before (μhb) graphs [31]. Figure 6b depicts an example μhb graph for the execution of the `mp` litmus test[10] on a pedagogical microarchitecture with three-stage in-order pipelines of Fetch (`IF`), Execute (`EX`), and Writeback (`WB`) stages. Nodes in these graphs represent sub-events in instruction execution. For instance, the first node in the second row represents the event when instruction

[10] Litmus tests are small 4-8 instruction programs used in the verification of memory consistency [2]. μhb graphs and μspec specifications are typically used for memory consistency verification, but they can also be used for hardware security verification [45].

Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
SC forbids: r1=1, r2=0	

(a) Code for litmus test `mp`.



(b) μhb graph for the execution of `mp`

Fig. 6: Left: Code for litmus test `mp`. The outcome `r1=1,r2=0` is forbidden under sequential consistency (SC) [22] (i.e., interleaving semantics). Right: An example μhb graph for the execution of the `mp` litmus test where `r1=1,r2=0` on a microarchitecture with three-stage in-order pipelines. The graph is cyclic (as highlighted by the bolded edges), implying that this execution is unobservable on the microarchitecture.

```
type uhbNode_t = record { nExists : boolean, nTime : integer };

type microop_t = record {
    globalID : integer, coreID : integer,
    <...>,
    Fetch : uhbNode_t, Execute : uhbNode_t,
    Writeback : uhbNode_t
};

define EdgeExists (src, dest : uhbNode_t) : boolean =
 (src.nExists == true && dest.nExists == true && src.nTime < dest.nTime);
define NodeExists (n : uhbNode_t) : boolean = (n.nExists == true);
define ProgramOrder (i, j : microop_t) : boolean =
 (i.globalID < j.globalID && i.coreID == j.coreID);

var i1, i2, i3, i4 : microop_t;
group testInstrs : microop_t = {i1, i2, i3, i4};

axiom ex_in_order :
 finite_forall (a : microop_t) in testInstrs ::
 finite_forall (b : microop_t) in testInstrs ::
  EdgeExists(a.Fetch, b.Fetch) ==> EdgeExists(a.Execute, b.Execute);
```

Fig. 7: Part of the embedding of μspec in UCLID5 and an example μspec axiom written in this embedding, illustrating UCLID5's capability for axiomatic modeling.

`i1` from `mp` performs its Execute stage. Meanwhile, edges in μhb graphs represent happens-before relationships. For instance, the blue edge between the first two nodes in the second row enforces that instruction `i1`'s Execute stage must occur before the Execute stage of instruction `i2`, reflecting the in-order nature of this processor's pipelines.

The presence or absence of nodes and edges in μhb graphs for a given microarchitecture are enforced by *axioms* in the domain-specific language μspec [32]. μspec supports propositional logic over its built-in predicates. It also supports quantifiers over instructions and enforces that the set of edges in μhb graphs is closed under transitivity.

We embedded μspec into UCLID5 to showcase UCLID5's capability for axiomatic modeling, as well as to enable μspec models to benefit from UCLID5's built-in capabilities for modularity and synthesis (Section 3.1). Figure 7 shows part of this embedding as well as an example μspec axiom written using the embedding.

We represent a μhb node in UCLID5 using two variables: a Boolean variable to represent whether or not the node exists and an integer recording the execution timestamp at which it occurred. An instruction (microop_t) consists of the nodes representing its sub-events as well as metadata such as its global ID (a unique identifier), core ID, address, and data value (some fields are not shown for brevity).

The existence of an edge (EdgeExists) between two nodes can be modeled as enforcing that both the source and destination nodes exist, and constraining the source node's timestamp to be less than that of the destination node. Node existence (NodeExists) merely checks the value of the nExists variable, while ProgramOrder is determined by ascending order of globalID on the same core.

The axiom ex_in_order states that for every pair of instructions a and b in the group testInstrs, if an edge exists between their IF stages, then an edge must also exist between their EX stages. Thus, this axiom enforces the existence of the blue edges between the EX stages of i1 and i2 and between those of i3 and i4 in Figure 6b.

Since edges in μhb graphs represent happens-before relationships, a cyclic μhb graph implies that an event must happen before itself. Thus, a cyclic μhb graph represents an execution that is unobservable (i.e., impossible) on the microarchitecture being modeled. Likewise, an acyclic μhb graph represents an execution that is observable on the microarchitecture. A given litmus test outcome can be verified on a microarchitecture by grounding the axioms over the instructions and outcome of that litmus test and asking a SMT solver to search for an acyclic μhb graph satisfying the axioms. If the solver returns a satisfying assignment, the test outcome is observable on the microarchitecture. If the solver returns UNSAT, the test outcome is guaranteed to be unobservable on the microarchitecture.

While litmus test verification of μspec specifications using SMT-based approaches has been conducted by prior work [32], encoding such modeling in UCLID5 has the benefit of harnessing UCLID5's built-in capabilities for modularity and synthesis. In fact, we are currently using UCLID5's synthesis capability in ongoing work to synthesise μspec axioms that match a set of examples.

A.3 Example: Control Synthesis with External Oracles

Consider the task of synthesising a controller for a Linear Time Invariant similar to Abate et al. [1]. We use a state-space representation, which is discretized in time. $\dot{x}_{t+1} = A\vec{x}_t + B\vec{u}_t$, where $\vec{x} \in \mathbb{R}^n$, $\vec{u} \in \mathbb{R}^p$ is the input to the system, calculated as $K\vec{x}$ where K is the controller to be synthesized, $A \in \mathbb{R}^{n\times n}$ is the system matrix, $B \in \mathbb{R}^{n\times p}$ is the input matrix, and subscript t indicates the discrete time step.

We aim to find a stabilizing controller K, such that absolute values of the (potentially complex) eigenvalues of the closed-loop matrix $A - BK$ are less than one, checked by the oracle function isStable. We further require that the controller guarantees the states remain within a safe region of the state space up to a given number of time steps, using the bounded model checking verification command in UCLID5, as shown in Figure 8.

```
module main {
  var x0, x1: float;
  group stateVars : float = {x0, x1};
  const a00, a01, a10, a11 : float;
  const b0, b1 : float;
  const AminusBK00, AminusBK01, AminusBK10, AminusBK11 : float;
  oracle function [isstable] isStable(s00:float, s01:float, s10:float, s11:float)
     : boolean;
  synthesis function k0 (): float;
  synthesis function k1 (): float;

  // LTI system spec
  axiom A: (a00==0.901224922471 && a01==0.000000013429 && a10==0.000000007451 &&
     a11==0.000000000000);
  axiom B: (b0==128.000000000000 && b1==0.000000000000);
  axiom ax1: AminusBK00 == a00 - b0*k0() && !isNaN(AminusBK00);
  axiom ax2: AminusBK01 == a01 - b0*k1() && !isNaN(AminusBK01);
  axiom ax3: AminusBK10 == a10 - b1*k0() && !isNaN(AminusBK10);
  axiom ax4: AminusBK11 == a11 - b1*k1() && !isNaN(AminusBK11);

  init { // bound initial states
    assume (finite_forall (state: float) in stateVars :: state<0.1 && state >
     -0.1);
  }
  next { // step the system
    x0' = AminusBK00*x0 + AminusBK01*x1;
    x1' = AminusBK10*x0 + AminusBK11*x1;
  }
  // the safety condition
  invariant stability: isStable(AminusBK00, AminusBK01, AminusBK10, AminusBK11);
  invariant safety: finite_forall (state: float) in stateVars :: state < 1.0 &&
     state > -1.0;
  invariant isNotNaN: finite_forall (state: float) in stateVars :: state < 1.0 &&
     state > -1.0;

  control {
    unroll(10); // fix safety bound
    check;
  }
}
```

Fig. 8: UCLID5 control synthesis example. The next block assigns to the state variables according to the standard definition of Linear Time Invariant systems. Note this model uses finite quantifiers, as described in Section 3.2.

```
module tap {
  // State variable declarations
  var tap_enclave_metadata_valid: tap_enclave_metadata_valid_t;
  var tap_enclave_metadata_addr_map: tap_enclave_metadata_addr_map_t;
  ...

  // Enclave operations
  procedure launch(eid: tap_enclave_id_t, ...) { ... }
  ...

  init { ... } // initialize TAP
  next { // step the system
    case
      (tap_current_mode == mode_untrusted) : {
        call (...) = AdversarialStep(...);
      }
      (tap_current_mode == mode_enclave) : {
        call (...) = EnclaveStep(...);
      }
    esac
  }
}

module integrity\_proof {
  // Create two instances of the TAP model
  instance tap_1: tap(...);
  instance tap_2: tap(...);

  // Example invariant: Memory that is mapped are equal between the two traces
  invariant equal_mem: (forall (pa : wap_addr_t) ::
    e_excl_map[pa] ==> (tap_1.mem[pa] == tap_2.mem[pa]));
  ...

  init { ... } // initialize proof
  next { // step the system
    next(tap_1); next(tap_2);
  }

  control {
    v = induction;
    check;
  }
}
```

Fig. 9: UCLID5 transition system-styled model of TAP and the integrity proof.