UCLID5: Integrating Modeling, Verification, Synthesis and Learning

Sanjit A. Seshia
EECS Department
University of California, Berkeley
sseshia@berkeley.edu

Pramod Subramanyan
CSE Department
Indian Institute of Technology, Kanpur
spramod@cse.iitk.ac.in

***Abstract*—Formal methods for system design are facing a confluence of transformative trends. First, systems are increasingly heterogeneous, comprising some combination of hardware, software, networking, and physical processes. Second, these systems are increasingly being designed with data-driven methods, in addition to traditional model-based design techniques. Third, traditional automated reasoning techniques based on deduction are being combined with new techniques for inductive inference and machine learning. In this paper, we present UCLID5, a new system for formal modeling, verification, and synthesis that addresses the challenges and opportunities arising from this confluence. UCLID5 can model heterogeneous computational systems, provides term-level abstraction supported by satisfiability modulo theories (SMT) solvers, enables compositional reasoning, and implements the paradigm of verification by reduction to synthesis, leveraging the advances in algorithmic synthesis and machine learning. We describe the key features of UCLID5 using illustrative examples.**

***Index Terms*—Formal methods, machine learning, specification, verification, synthesis, hardware, software, cyber-physical systems, security**

I. Introduction

Formal methods is a field of computer science and engineering concerned with the rigorous mathematical specification, design, and verification of systems [1], [2]. Progress in formal methods has led to a spectrum of effective techniques ranging from sequential program verification, model checking, simulation-based verification of temporal properties, abstract interpretation, interactive theorem proving, etc. However, in spite of this progress, several challenges remain for the wider adoption of formal methods, including in environment modeling, incompleteness in specifications, and the hardness of underlying decision problems (see [3] for further details).

A promising trend that addresses these challenges, as identified earlier by the first author [3], [4], is the *combination of traditional deductive methods with new techniques based on inductive inference and machine learning*. This trend combines traditional computational engines for formal methods, such as Boolean satisfiability solving (SAT) [5], Binary Decision Diagrams (BDDs) [6], and satisfiability modulo theories (SMT) solvers [7], with inductive learning, resulting in new class of solvers for syntax-guided and oracle-guided inductive synthesis (e.g., [8]–[11]). These solvers underpin an approach of performing verification by reduction to synthesis, to generate environment models, specifications, and other artifacts that ease the process of computational proving.

At the same time, we are facing a new trend that has to do with the nature of systems designed and the design process itself. Modeling languages for verification have traditionally been siloed by the class of system, with very different formalisms used for hardware, software, protocols, and cyber-physical systems. However, in recent times, *systems are becoming more heterogeneous*, and thus, verification problems often involve combinations of these domains. For instance, during a recent project on the verification of security properties of trusted platforms such as Intel SGX and RISC-V Sanctum [12], the authors realized that the task requires modeling both hardware (e.g., microarchitectural details) as well as software (e.g., OS, libraries, applications). Existing languages for constructing verification models, especially for highly-automated verification, are not adequate. Moreover, the types of properties to be verified include not only traditional assertions and temporal properties, but also hyperproperties [13]. Additionally, with the increasing use of machine learning in systems, *traditional model-based design methods are being combined with new data-driven approaches*. This is both a challenge and an opportunity. On the one hand, models need to be able to capture learning-based systems. On the other hand, there is a growing amount of data that can be used more effectively in modeling and verification.

The confluence of these trends indicates there is a need and an opportunity for a new class of formal modeling and verification systems. In this paper, we present UCLID5*, a new software toolkit for the formal modeling, specification, verification, and synthesis of computational systems*. The key novelty in UCLID5 is the integration of ideas in formal modeling, verification, synthesis, and learning to address the needs identified above by

1. Enabling compositional (modular) modeling of finite and infinite state transition systems across a range of concurrency models and background logical theories;
2. Performing highly-automated, compositional verification of a diverse class of specifications, including pre/post-conditions, assertions, invariants, temporal logic, refinement, and hyperproperties, and
3. Integrating modeling and verification with algorithmic synthesis and learning.

Importantly, we intend UCLID5 to be a laboratory for experimenting with new ideas in formal methods for the design of computational systems.

UCLID5 draws inspiration from the earlier UCLID system for modeling and verification of systems [14]–[16], in particular the idea of modeling concurrent systems at the "term level" — in first-order logic with a range of background theories —

978-1-5386-6195-6/18/$31.00©2018 IEEE.

and the use of proof scripts within the model. However, as we will describe in subsequent sections, the UCLID5 modeling language and verification capabilities go well beyond the original modeling language, and the integration with synthesis and learning is entirely novel.

In the rest of this paper, we describe the major needs of modeling and verification tools (Sec. II), an illustrative example (Sec. III), the key modeling features of UCLID5 (Sec. IV), and the key verification features (Sec. V), concluding with a summary of the main novel aspects of UCLID5 and an outlook towards the future (Sec. VI).

II. DESIDERATA

Selecting the right modeling formalism and language can be crucial for verification. As explained in a recent chapter in the handbook of model checking [17], there are several factors one must consider while selecting a modeling language for verification, including (1) type of system; (2) type of properties; (3) type of environment; (4) level of abstraction; (5) level of modularity; (6) form of composition; (7) computational engines available, and (8) practical ease of modeling and (9) expressiveness. In this section, we review the key desired features for formal modeling, verification, and synthesis that led us to design UCLID5. We also survey related work and place UCLID5 in the context of other systems for formal verification and automated reasoning.

A. Key Desired Features

The rationale for designing UCLID5 includes not only certain desiderata for formal modeling, but also desired properties for formal verification and synthesis. Our desired features are as follows:

- *Concurrent System Modeling:* The modeling language must easily express concurrent transition systems and concurrent updates to state variables. Many systems of interest, including hardware designs, operating systems, protocols, distributed systems, and cyber-physical systems (CPS) exhibit concurrency, and can be captured by a suitable class of transition systems.
- *Sequential Program Modeling:* The modeling language must provide constructs to easily express basic constructs of sequential programs including sequential updates, conditionals, iteration, procedure calls, etc. Additionally, it is desirable to be able to combine such constructs with concurrent system modeling, as we observed while modeling trusted platforms that include both hardware and software components [12].
- *Expressive Abstract Datatypes:* The modeling language should support not only low-level primitive datatypes such as Booleans and bit-vectors, but also mathematical and abstract datatypes, including integers, arrays, memories, uninterpreted functions, etc.
- *Diverse Specifications:* In our experience, there is usually not just one type of property that needs to be verified. We wish to have a system that can support specifications for sequential software (e.g., pre/post-conditions, assertions, assumptions, loop invariants), temporal specifications for concurrent systems (invariants, temporal logic, etc.), specifying refinement/simulation relations between systems, and hyperproperties to capture security policies and other richer classes of properties.
- *High Degree of Automation:* It is desirable to have a system that provides a high degree of automation in verification, especially with respect to the tedious aspects or the steps in the proof that require analyzing large state spaces. At the same time, we recognize that there is a balance to be achieved between high expressiveness of models and the degree of automation.
- *Diverse Verification Methods:* There are typically multiple algorithmic techniques for verifying the same class of properties on the same class of models, with often complementary performance. We wish to develop a system that supports this range of techniques.
- *Modular Specification and Verification:* Modularity, also termed compositionality, is key to scaling modeling and verification to large, real-world systems. We therefore desire a modeling language that supports modular system design, modular specification, and modular verification.
- *Leverage Emerging Computational Engines for Formal Reasoning:* SAT solvers, Binary Decision Diagrams (BDDs), and SMT solvers are the traditional computational engines used for automated reasoning and formal verification. However, in recent years, a new class of solvers for synthesis and learning have emerged (e.g., syntax-guided synthesis, SyGuS solvers [9]). It is desirable for new verification tools to be able to leverage these emerging computational engines for more efficient and effective automated reasoning.
- *Meaningful Counterexample Generation and Feedback:* One of the most important uses of formal methods is to find corner-case counterexamples, i.e. hard-to-find behaviors that result in a violation of desired properties. Any verification tool must support high-quality counterexample generation and other forms of feedback. In particular, the tool must be able to provide detailed counterexamples (showing values of all pertinent variables at all pertinent time points) while also permitting the user to customize counterexamples (e.g., giving the user the ability to restrict attention to a specific subset of state or times).

B. Related Work

There is a galaxy of software systems and tools for automated reasoning, formal verification, and formal synthesis, more than enough to fill a book on the topic. Many systems are designed for specific needs. Our goal in this section is not to provide a survey of all these tools; on the contrary, it is much more modest and focused entirely on the desiderata listed in Sec. II-A above. We will discuss a selection of the most closely related tools with respect to how well they provide those desired features, and argue for the need for a new system such as UCLID5.

A fundamental trade-off in automated formal reasoning is between expressiveness and automation. On one end of this spectrum are interactive theorem provers for higher-order logics such as PVS [22], HOL [23], Isabelle [24], ACL2 [25], and Coq [21], just to provide a few examples. These systems are very expressive, and so can capture both concurrent transition system models and sequential programs. However, they require

TABLE I
FEATURES WE DESIRE IN A FORMAL MODELING AND VERIFICATION SYSTEM.

Feature / Tool	ABC [18]	NuXMV [19]	Boogie [20]	Coq [21]	UCLID [14]
Concurrent systems/updates	H	H	L	M	H
Sequential programs/updates	L	L	H	H	L
Abstract Datatypes	L	M	H	H	H
Diverse Specifications	H	M	L	M	L
Diverse Verification Methods	H	M	L	M	M
High Degree of Automation	H	H	H	L	H
Modular Specification/Verification	M	M	H	H	L
Detailed, Customizable Counterexamples	H	M	L	L	M

We compare a selection of verifiers on the degree of support for these features: H – high, M – medium, L – low.

significant manual effort (even though many of them integrate other automated reasoning engines for specific tasks). We desire a much greater level of automation than these systems can provide.

On the other end of the spectrum are tools for bit-level (finite-state) modeling and model checking, including SPIN [26], SMV and NuSMV [27], and ABC [18]. These tools are highly automated, leveraging bit-level reasoning engines such as SAT solvers and BDD packages. They are effective for modeling finite-state concurrent systems, and can specify a range of temporal property languages. However, the representation is too low-level to perform effective system-level verification efforts, especially those that also require reasoning about software and richer datatypes.

In between lie several systems that have an intermediate level of expressiveness and automation. These systems typically rely on computational engines for reasoning in fragments of first-order logics with specialized background theories, including SMT solvers and custom solvers. The pioneering tools of this kind include Alloy [28], SAL [29], and UCLID [14]. Alloy is very effective for reasoning about software with relational logic, but not a good fit for low-level reasoning or modeling concurrent hardware. UCLID is a good fit for bounded verification of safety properties of concurrent transition systems, but does not provide a natural way to model sequential software or reason about more complex temporal logic properties. SAL supports a range of concurrent transition systems with different underlying logics, spanning applications in hardware, software, and CPS, but, like UCLID, does not naturally model sequential software. Support for compositional reasoning is also limited in these tools. More recent tools such as nuXmv [19] and KIND [30] also provide SMT-based verification methods for synchronous concurrent systems, and share the limitations of the afore-mentioned SMT-based tools with respect to sequential software. There is a plethora of program verification systems, such as Boogie [20], that provide excellent features for modular specification and verification of sequential software. Certain concurrent systems can be modeled using the non-deterministic constructs provided by these systems. However, modeling concurrent software and hardware using these systems is not straightforward, since one has to manually model the details of scheduling concurrent processes, performing synchronous composition, etc., which is tedious and error-prone.

One of the most exciting developments in formal methods over the last decade is the advance in algorithmic synthesis, applied to specifications, programs, controllers, etc. The *syntax-guided synthesis* (SyGuS) problem [9], the development of SyGuS solvers, and advances in inductive inference (machine learning) have opened up new automatic ways to synthesize artifacts arising in verification, including inductive invariants, assume-guarantee contracts, etc. (see [3] for further details). While a few systems for program synthesis (e.g., [31], [32]) include support for performing such syntax-guided synthesis, their objectives are different from those of formal verification systems, and they do not provide native support for modeling concurrent transition systems or specifying and verifying temporal properties. Moreover, few systems today make effective use of the advances in machine learning.

Thus, UCLID5 was created to meet a combination of needs expressed in Sec. II-A that is not adequately met by any of the existing tools. Table I summarizes the desired features and provides a high-level comparison with a small selection of the tools discussed. The table is intended not so much to point out limitations of these tools, but instead to highlight the unique combination of desired features we seek that is not adequately addressed by existing tools. UCLID5 seeks to supply a high level of support for all these features. It seeks to provide a natural way to model both concurrent transition systems and sequential software, using expressive abstract datatypes, specify a range of properties, provide a diverse palette of verification methods supported by state-of-the-art computational reasoning engines including those for synthesis and learning, support compositional reasoning, and give high-quality counterexamples and feedback to users. In the following sections, we describe how UCLID5 provides these desired features.

III. ILLUSTRATIVE EXAMPLE

This section provides a brief overview of UCLID5's modeling and verification features with an illustrative example: verifying a hyperproperty of a simple CPU model. Code for the complete model is split over Examples 1, 2, 3, 4 and 5.

A. The Structure of a UCLID5 Model

The atomic unit of modeling and verification in UCLID5 is a `module`. From the modeling perspective, each `module` can describe the functionality of a transition system. Multiple modules can be composed for modular construction of complex transition systems from a simpler ones. Modules also serve as the unit of re-use when declaring commonly used types, symbolic constants and uninterpreted functions.

```
module common {
// address type: an uninterpreted type.
type addr_t;
// word type: a bitvector type.
type word_t = bv32;
// type of operations supported by the CPU.
type op_t = enum {
  op_alu, op_load, op_store,
  op_imode_enter, op_nmode_exit
};
// CPU mode.
type mode_t = enum { normal_mode, isolated_mode };
// CPU memory type: an array type.
type mem_t = [addr_t]word_t;

// define zero constant of the word_t type.
const k0_word_t : word_t;
axiom k0_word_t == 0bv32;

// the entry address for isolated mode.
const imode_enter_addr : addr_t;
// the exit address for isolated mode.
const nmode_exit_addr : addr_t;
// the above two constants MUST be different.
axiom imode_enter_addr != nmode_exit_addr;
}
```

Example 1. Module `common` of the CPU model

```
module cpu {

// import all types from module common
type * = common.*;

type regindex_t;
type regs_t = [regindex_t]word_t;

input imem         : mem_t;  // program memory.
var dmem           : mem_t;  // data memory.
var regs           : regs_t; // registers.
var pc             : addr_t; // program counter.
var inst, result   : word_t; // inst reg, result
var mode           : mode_t; // normal/isolated?

// range of isolated memory.
const isolated_rng_lo, isolated_rng_hi : addr_t;
// uninterpreted predicate for the above.
function in_rng  (a : addr_t, b1 : addr_t, b2 :
    addr_t) : boolean;
// uninterpreted functions for decoding insns.
function inst2op
  (i : word_t) : op_t;
function inst2reg0
  (i : word_t) : regindex_t;
function inst2reg1
  (i : word_t) : regindex_t;
function inst2addr
  (i : word_t, r0 : word_t, r1 : word_t) : addr_t;
// uninterpreted functions for insn. execution
function aluOp
  (i : word_t, r0 : word_t, r1 : word_t) : word_t;
function nextPC
  (i : word_t, pc : addr_t, r0 : word_t) : addr_t;

// macro: is an addr in isolated memory?
define in_isolated_memory (a : addr_t) : boolean
  = in_rng(a, isolated_rng_lo, isolated_rng_hi);

// assumptions on imode entry and exit addresses
axiom (forall (a : addr_t) ::
        (a == common.imode_enter_addr)
            ==> in_isolated_memory(a));
axiom (forall (a : addr_t) ::
        (a == common.nmode_exit_addr)
            ==> !in_isolated_memory(a));

// code removed for formatting reasons:
// 1. code for procedure exec_inst (see Example 3)
// 2. init and next blocks (see Example 4)
}
```

Example 2. Type and variable declarations of the `cpu` module

```
procedure exec_inst(instr : word_t, pc : addr_t)
  returns (pc_next : addr_t)
  modifies regs, result, dmem, mode;
{
  var op    : op_t;
  var r0ind, r1ind : regindex_t;
  var r0, r1    : word_t;
  var addr  : addr_t;
  // get opcode.
  op = inst2op(instr);
  // get operands.
  r0ind,r1ind = inst2reg0(instr),inst2reg1(instr);
  r0,r1 = regs[r0ind],regs[r1ind];
  // get next pc (overwritten by enter/exit).
  pc_next = nextPC(instr, pc, r0);
  // get memory address
  addr = inst2addr(instr, r0, r1);
  // If we are in isolated mode, we only
  // set pc_next to isolated addresses.
  assume (mode == isolated_mode) ==>
     in_isolated_memory(pc_next);
  // If we are in isolated mode, we only
  // read from isolated memory.
  assume (mode == isolated_mode && op == op_load)
     ==> in_isolated_memory(addr);
  // If we are already in isolated mode,
  // we don't execute enters.
  assume (mode == isolated_mode)
             ==> (op != op_imode_enter);
  case
    // alu operation.
    (op == op_alu) : {
      result = aluOp(instr, r0, r1);
      regs[r0ind] = result;
    }
    // load instruction.
    (op == op_load) : {
      // check permissions.
      if (mode == isolated_mode ||
          !in_isolated_memory(addr))
      {
        // perform load
        result = dmem[addr];
      } else {
        // load failed, return 0.
        result = common.k0_word_t;
      }
      regs[r0ind] = result;
    }
    // store instruction.
    (op == op_store) : {
      result = common.k0_word_t;
      // check permissions.
      if (mode == isolated_mode ||
          !in_isolated_memory(addr))
      {
        // perform store.
        dmem[addr] = r0;
      }
    }
    // enter isolated mode.
    (op == op_imode_enter) : {
      assert (mode == normal_mode);
      result = common.k0_word_t;
      // zero out registers.
      havoc regs;
      assume (forall (r : regindex_t)
                :: regs[r] == common.k0_word_t);
      // set pc.
      pc_next = common.imode_enter_addr;
      mode = isolated_mode;
    }
    // exit to normal mode.
    (op == op_nmode_exit) : {
      result = common.k0_word_t;
      // zero out registers.
      havoc regs;
      assume (forall (r : regindex_t)
                :: regs[r] == common.k0_word_t);
      // set pc.
      pc_next = common.nmode_exit_addr;
      mode = normal_mode;
    }
  esac
}
```

Example 3. Procedure `exec_inst` in the CPU model

```
init {
  // reset registers.
  assume (forall (r : regindex_t)
              :: regs[r] == common.k0_word_t);
  // set instruction to some deterministic value.
  inst = common.k0_word_t;
  // start execution at deterministic address.
  pc   = common.nmode_exit_addr;
  // in normal mode.
  mode = normal_mode;
}

next {
  inst' = imem[pc];
  call (pc') = exec_inst(inst', pc);
}
```

Example 4. The `init` and `next` blocks of the `cpu` module

Structure of the CPU Model: The CPU model is split into three UCLID5 modules.

1. Module `common`, shown in Example 1 provides the definitions for the datatypes (`addr_t`, `word_t`, etc.) and symbolic constants (e.g., `imode_enter_addr`) used in the rest of the model. Note this module does not define a transition system, it only provides type and variable definitions used elsewhere in the model.
2. Module `cpu`, shown in Examples 2, 3 and 4 models the functionality of a CPU with separate instruction and data memories and an isolated mode of execution. We describe the functionality of the `cpu` module in more detail in Section III-C.
3. Module `main`, shown in Example 5 is the verification driver. It models the environmental assumptions for the verification and contains a proof script that proves, via induction, a 2-safety integrity property for the CPU module.

B. Modeling Objectives

The `cpu` model presented in Examples 2, 3 and 4 models the functionality of a CPU with the following features: (i) a read-only instruction memory (`imem`), (ii) a read-write data memory (`dmem`), (iii) a program counter (`pc`), (iv) an unbounded number of registers (`regs`), (v) two modes of operation (`mode`): normal mode and isolated mode and (vi) a range of addresses between `isolated_rng_lo` and `isolated_rng_hi` which defines a memory region exclusively accessible to isolated mode.

The module uses uninterpreted functions to model instruction decoding (`inst2op`, `inst2reg0` etc.) and instruction execution (`aluOp` and `nextPC`). Further, the register file index type (`regindex_t`) and the address type (`addr_t`) are both uninterpreted types. This means the number of registers is unbounded, as is the size of the data and instruction memories. Abstracting away the specific details of each these features lets this UCLID5 model capture a wide variety of CPU implementations.

C. CPU Module Overview

The CPU module is shown in Examples 2, 3 and 4. Example 2 shows the declarations of the types, symbolic constants, uninterpreted functions, input variables, state variables and assumptions used by the module. The initial state and the transition relation of the transition system are shown in

```
module main {
// Import types.
type * = common.*;
type regindex_t = cpu.regindex_t;

// Instruction memory for the CPUs.
var imem1, imem2 : mem_t;

// Create two instances of the CPU module.
instance cpu1 : cpu(imem : (imem1));
instance cpu2 : cpu(imem : (imem2));

init {
  // Assume same isolated rngs.
  assume (cpu1.isolated_rng_lo == cpu2.
     isolated_rng_lo);
  assume (cpu1.isolated_rng_hi == cpu2.
     isolated_rng_hi);
  // Supervisor memory starts off identical.
  assume (forall (a : addr_t) :: (cpu.in_rng(a,
     cpu1.isolated_rng_lo, cpu2.isolated_rng_hi))
     ==> (cpu1.dmem[a] == cpu2.dmem[a]));
  assume (forall (a : addr_t) :: (cpu.in_rng(a,
     cpu1.isolated_rng_lo, cpu1.isolated_rng_hi))
     ==> (imem1[a] == imem2[a]));
}

next {
  next (cpu1);  next (cpu2);
  // imodes are taken in sync
  assume (cpu.inst2op(cpu1.inst) == op_imode_enter
      || cpu.inst2op(cpu2.inst) == op_imode_enter)
      ==> (cpu1.inst == cpu2.inst);
}

// PROPERTY: isolated mode memory is identical.
property eq_dmem : (forall (a : addr_t) :: (cpu.
     in_rng(a, cpu1.isolated_rng_lo, cpu2.
     isolated_rng_hi)) ==> (cpu1.dmem[a] == cpu2.
     dmem[a]));
// The two CPUs change mode in sync.
invariant eq_mode : (cpu1.mode == cpu2.mode);
// CPUs have the same PC when in isolated mode.
invariant eq_pc   : (cpu1.mode == isolated_mode)
     ==> (cpu1.pc == cpu2.pc);
// In isolated mode, CPUs execute the same code.
invariant eq_inst : (cpu1.mode == isolated_mode)
     ==> (cpu1.inst == cpu2.inst);
// In isolated mode, CPUs have same reg values.
invariant eq_regs : (forall (ri : regindex_t) :: (
     cpu1.mode == isolated_mode) ==> (cpu1.regs[ri
     ] == cpu2.regs[ri]));
// Supervisor rng of memory is the same.
invariant eq_sup_rng : (cpu1.isolated_rng_lo ==
     cpu2.isolated_rng_lo) && (cpu1.
     isolated_rng_hi == cpu2.isolated_rng_hi);
// In isolated mode, PC is in isolated rng.
invariant in_pc_rng1 : (cpu1.mode == isolated_mode
     ) ==> cpu.in_rng(cpu1.pc, cpu1.
     isolated_rng_lo, cpu1.isolated_rng_hi);
invariant in_pc_rng2 : (cpu2.mode == isolated_mode
     ) ==> cpu.in_rng(cpu2.pc, cpu2.
     isolated_rng_lo, cpu2.isolated_rng_hi);
// Same for instruction memory.
invariant eq_imem : (forall (a : addr_t) :: (cpu.
     in_rng(a, cpu1.isolated_rng_lo, cpu1.
     isolated_rng_hi)) ==> (imem1[a] == imem2[a]))
     ;

control {
  v = induction;
  check;
  print_results;
  v.print_cex(
      cpu.inst2op(cpu1.inst), cpu.inst2op(cpu2.
     inst), cpu1.result, cpu2.result, cpu1.mode,
     cpu2.mode, cpu1.pc, cpu2.pc, cpu1.
     isolated_rng_lo, cpu2.isolated_rng_lo, cpu1.
     isolated_rng_hi, cpu2.isolated_rng_hi, cpu.
     in_rng(cpu1.pc, cpu1.isolated_rng_lo, cpu1.
     isolated_rng_hi), cpu.in_rng(cpu2.pc, cpu2.
     isolated_rng_lo, cpu2.isolated_rng_hi));
}
}
```

Example 5. Module `main` in the CPU model

Example 4. Example 3 lists the procedure `exec_inst` which implements the execution of a single instruction.

1) Module Declarations: Datatypes used in the `cpu` model are defined in the module `common` shown in Example 1. These include uninterpreted types: `addr_t`, bit vectors: `word_t` which is defined as a synonym for 32 bit vectors (`bv32`), enumerated types: `op_t` and `mode_t` and array types: `mem_t` which is defined as a synonym for arrays with index type `addr_t` and range type `word_t`. Line 4 of Example 2 shows how all of the datatypes declared in the module `common` can be "imported" into the module `cpu`.

Line 9 declares the input variable `imem`, which is the instruction memory for the CPU. Lines 10–14 declare the state variables of the CPU: `dmem` is the data memory, `regs` is the register file, `pc` is the program counter, and `mode` tracks whether the CPU is in normal or isolated mode.

The uninterpreted functions that model instruction decoding and execution are declared between lines 21–33. Line 36 defines a macro that determines whether an address is in isolated memory.

Finally, lines 40–45 encode two important assumptions on the behavior of the CPU. Entering isolated mode always jumps to an address in isolated memory and exits to normal mode transfer control to an address not in isolated memory.

2) Defining the Transition System: The initial state of the transition system is defined by the `init` block (lines 1–11) of Example 4. This block ensures that all registers are initialized to 0, sets the initial value of the program counter to a deterministically chosen address that is guaranteed to not be in isolated memory, and sets the CPU to normal mode.

The transition relation is defined by the `next` block shown on lines 13–16 of Example 4. The instruction is fetched from instruction memory and executed by invoking the procedure `exec_inst`. This procedure is defined in Example 3 and models the execution of an instruction in the CPU using straightforward imperative code.

D. Verification Objectives

The verification objective for this UCLID5 model is to show the following: if two CPUs start off with identical values in the isolated memory (but possibly different normal-accessible memories), and further, these CPUs enter isolated mode identically and in lockstep, then the two CPUs' isolated memory ranges will remain identical. This is an integrity property showing that normal mode cannot affect isolated memory. This property is stated on line 29 of the `main` module in Example 5.

This specification, which is a 2-safety property, is verified in UCLID5 by creating two instances of the `cpu` module (lines 10 and 11 of Example 5). The verification environment is set up to satisfy the antecedents of the property in order to ensure that the two CPUs start with identical values in isolated memory and that they enter isolated mode in lockstep; see the `init` (lines 13–20) and `next` blocks (lines 22–26) of the `main` module.

E. Verification Strategy

We prove the 2-safety property by induction; this requires the specification of a number of "strengthening" inductive invariants. These are specified on lines 31–44 of Example 5. The proof script itself is in the `control` block of the `main` module (lines 46–52). Line 47 specifies that an inductive proof is to be attempted, invocation of the proof engines is done on line 48 and results of the verification and any potential counterexamples are printed on lines 49, 50 and 51.

IV. Modeling Features

UCLID5 integrates multiple features for modeling to address the desiderata discussed in Sec. II-A. In this section, we give an overview of these modeling features through illustrative examples. The full language reference and additional information may be found in the UCLID5 tutorial [33].

A. Term-Level Abstraction

UCLID5 borrows from the original UCLID [14] system the idea of *term-level abstraction*. In term-level abstraction, concrete functions or functional blocks are replaced by uninterpreted functions or partially-interpreted functions. Concrete low-level datatypes such as Booleans, bit-vectors, and finite-precision integers are abstracted away by uninterpreted symbols or more abstract datatypes (such as the unbounded integers). UCLID5 is designed to use all the major background logical theories supported by SMT solvers.

Term-level abstraction is useful in hiding details that are unnecessary for verification. Consider the CPU model given as Example 2. In this example, uninterpreted functions, defined using the `function` declaration, are used to abstract away the details of instruction decoding (`inst2op`, `inst2addr`, etc. on lines 21–28). These functions model the mapping from instructions to the operations performed by them, the source and destination registers of an instruction, and any memory addresses referenced by the instruction. The uninterpreted function `aluOp` (line 30) models the execution of non-memory instructions while the function `nextPC` (line 32) models instructions that affect control flow. The specific details of the implementation of these functions are not required for reasoning about the properties specified in the `main` module given as Example 5.

Interpreted or partially-interpreted functions may be defined using a combination of the `function` and `define` constructs. Further details on performing term-level abstraction may be found in [14], [15], [34], [35].

B. Blending Sequential and Concurrent System Modeling

UCLID5 combines constructs for modeling sequential programs with those for modeling concurrent transition systems.

1) Sequential Program Modeling: Inspired by systems such as ESC/Java [36] and Boogie [20], UCLID5 supports constructs to perform modular program verification. A `procedure` is the unit of sequential programming in UCLID5. Within a procedure, one can use most standard constructs of imperative, sequential programming, including variable declarations, sequencing, assignments, conditionals, and iteration (including bounded for-loops and unbounded while loops). No recursion is currently permitted. In addition, similar to verification languages such as Boogie, nondeterministic constructs can be modeled. An arbitrary assignment to a variable may be performed using the `havoc` construct: the statement "`havoc v;`" assigns an arbitrary value to `v` from the domain associated with its type. See

the procedure `exec_inst` in Example 3 for examples of the various code constructs. Simultaneous assignment is also possible. Specifications can also be associated with procedures in the usual way; we defer discussion of these to Sec. IV-C.

2) Concurrent System Modeling: Many automated verification tools, including model checkers, model concurrent systems as transition systems. A transition system comprises a set of variables modeling input, output, and state of the system along with definitions of the set of initial states and the transition relation. UCLID5 also provides this ability. In UCLID5, a `module` is the unit of concurrent modeling.

Consider the code in Examples 2, 4 and 5. The `cpu` module includes variable declarations, an `init` block that defines the set of initial states, and a `next` block that defines the transition relation. The `init` block is treated similar to sequential code within a procedure; thus, sequential updates are possible. However, a `next` block only permits synchronous (concurrent) updates to next-state variables, where the next-state variable for `v` is denoted by priming it, as `v'`. Examples 2, 4 and 5 give illustrations of the code in modules. We also note that the `next` block of a module can be stepped selectively, allowing one to select when a module executes relative to other modules.

While previous and existing modeling languages have either supported modeling sequential programs or concurrent systems, UCLID5 is the first, to our knowledge, that supports the ability to model both for the purpose of formal verification. This gives one the ability to combine both formalisms to deal with the modeling challenges of combined hardware-software systems such as enclave platforms [12].

C. Diversity in Specification

UCLID5 supports a variety of ways to formally specify properties of the system being modeled. Currently, the following kinds of properties can be directly specified with associated keywords.

Procedure specifications: In UCLID5, similar to program verifiers such as Boogie, one can accompany the definition of a procedure with pre-conditions (specified using `requires`) and post-conditions (specified using `ensures`), as well as specify variables that are modified by the procedure using `modifies`. An example of a procedure with pre-/post-conditions is shown in Example 7. The procedure `searchQ` searches through the queue data structured modelled by the array `data`. The procedure pre-condition (line 5) states that `count` — the number of items currently in the queue — should be a non-negative value less than the maximum size of the queue. The post-condition (line 6) states that the return value `found` is true if and only if `data` exists in the queue.

Assertions and Assumptions: An assertion specifies that a Boolean condition over program variables must always hold at a point inside a procedure; it forms a proof obligation for UCLID5. An assumption imposes a Boolean condition over program variables at a particular point in a procedure; it is added to the assumptions UCLID5 uses to discharge proof obligations from that point onwards. Assertions and assumptions are specified per usual with the `assert` and `assume` keywords respectively. Use of these statements is demonstrated on lines 61 and 65 of Example 3.

```
1  module queue {
2  // types and constants.
3  type op_t = enum { push, pop, search };
4  type queue_t = [integer]integer;
5  const SIZE : integer = 4;
6  // inputs, outputs and state variables
7  input op                          : op_t;
8  input data                        : integer;
9  output head                       : integer;
10 output valid, full, empty, found : boolean;
11 var contents                      : queue_t;
12 var head, tail, count             : integer;
13 var inited                        : boolean;
14 // macros
15 define itemIndex(i : integer) : integer =
16   if ((i + head) >= SIZE)
17     then (i + head) - SIZE
18     else (i + head);
19 define in_queue(v : integer) : boolean =
20   (exists (i : integer) ::
21     (i >= 0 && i < count) &&
22      contents[itemIndex(i)] == v);
23 // Procedure searchQ shown in Example 7.
24 init {
25   head, tail, count = 0, 0, 0;
26   valid, full, empty = false, false, false;
27   inited, found = false, false;
28 }
29 next {
30   inited' = true;
31   head'   = contents[head];
32   full'   = (count' == SIZE);
33   empty'  = (count' == 0);
34
35   case
36     (op == push && !full) : {
37       call (contents', tail', count') = pushQ();
38     }
39     (op == pop && !empty) : {
40       call (head', count') = popQ();
41     }
42     (op == search) : {
43       call (found') = searchQ();
44     }
45   esac;
46 }
47 // Symbolic constant to "capture" pushed value
48 const pushed_data : integer;
49 property[LTL] pushed_value_becomes_head :
50   // pops occur infinitely often
51   G(F(op == pop)) ==>
52   // every value that is pushed
53   G((inited && op == push &&
54      data == pushed_data && !full) ==>
55     // eventually becomes the head of the queue
56     F(head == pushed_data));
57 control {
58   vQ      = bmc(5);
59   vPush   = verify(pushQ);
60   vPop    = verify(popQ);
61   vSearch = verify(searchQ);
62   check;
63   print_results;
64   vQ.print_cex(op, data, pushed_data,
65     head, tail, head, count, contents[0],
66     contents[1], contents[2], contents[3]);
67 } }
```

Example 6. Queue model in UCLID5

Invariants: Sequential programs may also include unbounded `while` loops. Partial correctness of programs using while loops can be checked by specifying loop invariants using the `invariant` keyword. Example 7 shows two invariants for the while loop (lines 13 and 14) which respectively state that: (i) the loop index is always within the bounds of array `data`, and (ii) and that found is set to true iff one of the array elements accessed by the loop is equal to the value `data`.

Temporal invariants (i.e., inductive invariants of the transition system corresponding to a module) can also specified

```
1  // Procedures pushQ and popQ not shown for brevity
2
3  procedure searchQ()
4    returns (found : boolean)
5    requires (count >= 0 && count <= SIZE);
6    ensures (in_queue(data) <==> found);
7  {
8    var i : integer;
9
10   i = 0;
11   found = false;
12   while (i < count)
13     invariant (i >= 0 && i <= count);
14     invariant
15       (exists (j : integer) ::
16         j >= 0 && j < i &&
17         contents[itemIndex(j)] == data)
18       <==> found;
19   {
20     if (contents[itemIndex(i)] == data) {
21       found = true;
22     }
23     i = i + 1;
24   }
25 }
```

Example 7. Procedure searchQ in the Queue model

using the `invariant` keyword. Several examples of such inductive invariants are shown in Example 5.

Linear Temporal logic (LTL): LTL specifications (see [37]) can be provided by using the `[LTL]` decorator along with the `property` keyword. Temporal operators currently supported in UCLID5 are: `G` (globally), `F` (eventually), `U` (strong until), `W` (weak until) and `X` (next). An example of an LTL specification is shown on line 49 of Example 6. This property states that every value pushed will eventually reach the head of the queue assuming the queue is popped infinitely often.

In addition, UCLID5 currently supports indirect specification of other kinds of properties as listed below.

Hyperproperties (k-safety): An important application of UCLID5 is to reason about security properties, many of which are hyperproperties [13]. While support for directly specifying hyperproperties in UCLID5 is currently being investigated (e.g., using logics such as HyperLTL [38]), one can specify a sub-class of hyperproperties indirectly by encoding to safety properties (temporal invariants). Specifically, k-safety properties [13] can be specified by the standard approach of composing k copies of the system together and specifying a safety property on the resulting composition. Example 5 gives an example of a 2-safety property on the CPU model that is encoded as a safety property on a self-composition of 2 instantiations of the CPU module.

Simulation/Correspondence between Modules: UCLID5 can also be used to check if one system simulates another, i.e., that steps of one module can be mimicked by the other. (See Chapter 14 of [39] for background material on simulation.) Typically, each system is modeled a UCLID5 module, and the simulation check can be set up in the `main` module. A prototypical example of such a check is the Burch-Dill correspondence checking for processor verification [40], where one checks whether the instruction set architecture (ISA) specification model of a processor simulates its implementation. An example of correspondence checking may be found in the UCLID5 tutorial [33].

D. Modularity in Modeling and Specification

Modular (compositional) reasoning is essential for scalable verification and synthesis. UCLID5 provides multiple features for modular reasoning, including:

- *Modules:* UCLID5 models are composed of modules and these provide a mechanism for both compositional and hierarchical modeling. Modules are instantiated using the `instance` declaration as shown in Example 5. By default, modules are composed together synchronously, but asynchronous composition and partially synchronous composition can be performed using the appropriate scheduling logic. State variables are private to modules, but they can share variables declared in a parent module using a `sharedvar` declaration. The `main` module is always the top-level module. Overall, these features are similar to other verification tools for reactive and concurrent systems.
- *Procedures:* Procedures provide a mechanism to modularize the sequential program logic of a UCLID5 model. Procedures can be decorated with `[inline]` and `[noinline]` decorators that direct UCLID5 to inline them during verification, or instead use their specifications instead of inlining them, respectively. These features are similar to those provided by other tools for sequential program verification.
- *Modular Specification and Verification:* UCLID5 extends modularity to specifications in the natural way. Procedures can be specified with pre/post-conditions and modifies clauses. Properties defined within a module are local to it and can include both assumptions as well as proof obligations to be verified (including invariants, LTL properties, etc.). Modular verification of concurrent systems can be performed via assume-guarantee reasoning. The `control` block of each module can be used to specify a proof script that is local to that module. By default, only the proof script of the `main` module is executed. In this manner, the user can control the granularity of verification within UCLID5.

V. Verification Features

UCLID5 support a variety of verification methods to go with the diverse modeling and specification formalisms it provides. In this section, we give a brief overview of the major kinds of verification currently supported by UCLID5.

A. Sequential Program Verification

Sequential program verification is supported in UCLID5 using the `verify` command. This command checks partial correctness of procedures. It translates procedure pre-conditions, post-conditions and loop invariants into a set of verification conditions (VCs) using an algorithm similar to ESC/Java [41]. VCs are discharged using an SMT solver. Like most other program verifiers including Boogie, UCLID5 requires loop invariants to be manually specified. However, as we describe in Sec. V-C below, one can leverage underlying synthesis solvers to generate such invariants in certain settings.

B. Induction, Bounded Model Checking, and Symbolic Simulation

Temporal invariants of transition systems can be verified using the `induction(k)` command. The argument `k` is the 'k' in k-induction and defaults to one. Example 5 uses the `induction` command on line 47.

LTL specifications can verified using the `bmc(n)` command which performs bounded model checking up to a bound of 'n' transitions. Verification is based on the construction of monitors à la Claessen et al. [42] Liveness properties are verified by reduction to safety through "lasso" detection. Example 6 uses the `bmc` command on Line 58 to perform bounded model checking of the LTL property on Lines 49-56 in the queue model.

Similar to UCLID, UCLID5 can also be used to perform symbolic simulation (execution) of the transition system in a configurable manner, which allows one to set up simulation/refinement checks (see [33] for an example).

C. Synthesis-Driven Verification

UCLID5 seeks to implement the paradigm of *verification by reduction to synthesis* [3]. In particular, it seeks to leverage the advances in algorithmic synthesis, particularly counterexample-guided inductive synthesis [8], syntax-guided synthesis [9] and formal inductive synthesis [11], to automate tricky or tedious sub-tasks of verification.

One of these sub-tasks is that of finding invariants or strengthenings of invariants to perform proofs by (k-)induction. For this, UCLID5 currently provides a command, `synthesize_invariant`, that calls an underlying *synthesis solver* to find a (strengthened) inductive invariant that completes a proof by induction. Currently, this command is discharged by syntax-guided synthesis (SyGuS) [9] solvers.

More generally, UCLID5 is designed to provide for syntax-guided synthesis of other model and proof artifacts. The language has constructs to specify functions to be synthesized as well as syntactic restrictions via grammars. These "synthesis functions" are then replaced with implementations generated by the back-end solvers whenever proof obligations involving those synthesis functions are discharged. As discussed in [3], such an approach can be used to synthesize not just inductive invariants, but also abstractions, pre/post-conditions, assume-guarantee contracts, and many more proof and model artifacts that are essential for formal verification.

D. Other Features

UCLID5 supplements its suite of verification methods with constructs that allow users to perform modular verification and to control the level of information provided to them via counterexamples and other forms of feedback. Two sample features are described below.

1) Modular Proof Scripting: UCLID5 borrows from UCLID [14], [16] the notion of a *control block*, which is the part of a model that specifies a proof script, a list of commands to UCLID5 specifying the sequence of proof steps and other auxiliary steps. It is important to note that each of these proof steps is typically a fully automated verification command, not the detailed guidance one has to give to an interactive theorem prover or proof assistant such as Coq or PVS. Two examples of a control block are given in Lines 46–52 of Example 5 and Lines 57–67 of Example 6.

UCLID5 goes beyond UCLID in having control blocks that are local to each module. Thus, one can use it to perform local reasoning. For example, the control block in Example 6 is local to the `queue` module. This module can be instantiated within another module, but the verification steps specified in the `queue` module do not need to be repeated when performing verification in the module that instantiates it. However, any properties proved about the queue model can be used in the instantiating module.

2) Counterexample Generation: When a property is violated, UCLID5 can generate a detailed counterexample showing the values of all variables of the corresponding module. For a proof by induction, this is a counterexample-to-induction (CTI), a 2-state counterexample. For a program verification problem involving proving a post-condition of a procedure, it will similarly print out the pre-state and the post-state of that procedure. For temporal properties, including temporal invariants and LTL properties, it will print out a sequence of states showing how the property is violated (including lasso-like counterexamples for liveness properties).

Counterexamples are associated with verification objects — objects that store the results of a verification command. For example, `vQ` and `vPush` are the verification objects associated with the `bmc` and `verify` commands used on Lines 68 and 69 of Example 6 respectively. Each has an associated counterexample when it is violated, which can be displayed by the user. By default, UCLID5 prints out all variables, but the user can restrict the counterexample to a subset for readability and ease of understanding. UCLID5 counterexamples can also be translated into output for third-party viewers; e.g., currently, there is a translator to the value change dump (VCD) format used commonly in circuit simulation and verification.

VI. Conclusion

UCLID5 is a new system for formal modeling, verification, and synthesis whose key novel contribution is the integration of several features in a single language and toolkit: (1) blending concurrent system and sequential program modeling; (2) combining term-level abstraction with the ability to perform low-level reasoning; (3) integrating algorithmic synthesis and machine learning with verification; (4) combining the ability to specify a diverse range of properties with diverse highly-automated verification capabilities, and (5) enabling modular reasoning and verification. Initial experience with UCLID5 has shown that this combination of features can be effective at tackling the challenges identified at the beginning of this paper. Several case studies are underway.

There are several exciting ongoing projects through which we are actively improving and extending the capabilities of UCLID5. We are working on improving the support for synthesis through more efficient and expressive SyGuS solvers. Additionally, we are adding support for a broader class of oracle-guided inductive synthesis (OGIS) [11] solvers, so as to enable the generation of models from implementations [43]. Support for direct specification and verification of hyperproperties is being added. We are also integrating UCLID5 more closely with machine learning for various tasks including importing data, inferring specifications, and auto-tuning the parameters of the underlying reasoning engines. Finally, we are exploring the extension of UCLID5 to a broader class of systems, including cyber-physical systems.

Acknowledgments

We gratefully acknowledge several people who have contributed to the development of UCLID5, including Kevin

Cheang, Albert Magyar, Cameron Rasmussen, and Rohit Sinha. We thank Randy Bryant for being an early user and providing encouragement, crucial feedback, and critical insights. UC Berkeley students in EECS 219C, Spring 2018, also provided valuable feedback. The second author did much of the work reported in this paper while at UC Berkeley. This work was funded in part by the National Science Foundation under grants CNS-1528108 and CNS-1545126, by SRC contract 2638.001, by the Intel ADEPT Center, by the iCyPhy center, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, no. 9, pp. 8–24, September 1990.

[2] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[3] S. A. Seshia, "Combining induction, deduction, and structure for verification and synthesis," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2036–2051, 2015.

[4] ——, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," in *Proceedings of the Design Automation Conference (DAC)*, June 2012, pp. 356–365.

[5] S. Malik and L. Zhang, "Boolean satisfiability: From theoretical hardness to practical success," *Communications of the ACM (CACM)*, vol. 52, no. 8, pp. 76–82, 2009.

[6] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[7] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[8] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proc. Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006, pp. 404–415.

[9] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, October 2013, pp. 1–17.

[10] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 215–224.

[11] S. Jha and S. A. Seshia, "A Theory of Formal Synthesis via Inductive Learning," *Acta Informatica*, vol. 54, no. 7, pp. 693–726, 2017.

[12] P. Subramanyan, R. Sinha, I. A. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2435–2450.

[13] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[14] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proc. Computer-Aided Verification (CAV'02)*, ser. LNCS 2404, E. Brinksma and K. G. Larsen, Eds., July 2002, pp. 78–92.

[15] S. A. Seshia, "Adaptive eager boolean encoding for arithmetic reasoning in verification," Ph.D. dissertation, Carnegie Mellon University, May 2005.

[16] "UCLID Verification System, version 3.1," Available at http://uclid.eecs.berkeley.edu.

[17] S. A. Seshia, N. Sharygina, and S. Tripakis, "Modeling for verification," in *Handbook of Model Checking*, E. M. Clarke, T. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, ch. 3, pp. 75–105.

[18] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[19] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 334–342.

[20] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: a modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 364–387.

[21] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy *et al.*, "The Coq proof assistant reference manual: Version 6.1," INRIA, https://coq.inria.fr/, 1997.

[22] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, vol. 607, Jun. 1992, pp. 748–752.

[23] M. J. Gordon, "HOL: A proof generating system for higher-order logic," in *VLSI specification, Verification and Synthesis*. Springer, 1988, pp. 73–128.

[24] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[25] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on Common Lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, 1997.

[26] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[27] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[28] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[29] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 496–500.

[30] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 model checker," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 510–517.

[31] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, EECS Department, UC Berkeley, 2008.

[32] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013, pp. 135–152.

[33] P. Subramanyan and S. A. Seshia, "Getting started with Uclid5," Available at https://github.com/uclid-org/uclid.

[34] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary, "ATLAS: automatic term-level abstraction of RTL designs," in *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2010, pp. 31–40.

[35] B. Brady, R. E. Bryant, and S. A. Seshia, "Learning conditional abstractions," in *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, October 2011, pp. 116–124.

[36] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002, pp. 234–245.

[37] N. Piterman and A. Pnueli, "Temporal logic and fair discrete systems," in *Handbook of Model Checking*. Springer, 2018, pp. 27–73.

[38] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," in *International Conference on Principles of Security and Trust*. Springer, 2014, pp. 265–284.

[39] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*, 2nd ed. MIT Press, 2016. [Online]. Available: http://leeseshia.org

[40] J. R. Burch and D. L. Dill, "Automated verification of pipelined microprocessor control," in *Computer-Aided Verification (CAV '94)*, ser. LNCS 818, D. L. Dill, Ed. Springer-Verlag, June 1994, pp. 68–80.

[41] C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: Generating compact verification conditions," in *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL)*, 2001, pp. 193–205.

[42] K. Claessen, N. Een, and B. Sterin, "A circuit approach to LTL model checking," in *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, October 2013, pp. 53–60.

[43] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for soc verification," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, 2015, pp. 160–167.