Getting Started with Uclid5
Alpha release, version 0.9.1

Pramod Subramanyan and Sanjit A. Seshia
{*spramod, sseshia*}*@eecs.berkeley.edu*

May 2018

Contents

List of Uclid5 Examples

1. Introduction

Uclid5 is a software toolkit for the formal modeling, specification, verification, and synthesis of computational systems. The Uclid5 toolchain aims to:

1. Enable compositional (modular) modeling of finite and infinite state transition systems across a range of concurrency models and background logical theories;

2. Verification of a range of properties, including assertions, invariants, and temporal properties, and

3. Integrate modeling and verification with algorithmic and inductive synthesis.

Uclid5 draws inspiration from the earlier UCLID system for modeling and verification of systems [2, 1], in particular the idea of modeling concurrent systems in first-order logic with a range of background theories, and the use of proof scripts within the model. However, the Uclid5 modeling language and verification capabilities go beyond the original modeling language, and the planned integration with synthesis is novel.

This document serves as introduction to the Uclid5 modeling language and toolchain. With the Uclid5 system under active development, we expect this document to undergo several changes as the system and its applications evolve.

1.1. Getting Started: A Simple Uclid5 Model

A simple Uclid5 module that computes the Fibonacci sequence is shown in Example 1.1. We will now walk through each line in this model to understand the basics of Uclid5.

The top-level syntactic structure in Uclid5 is a `module`. All modeling, verification and synthesis code in Uclid5 is contained within modules. In Example 1.1, we have defined one `module` named `main`. This module starts on line 1 and ends on line 18. The module can be conceptually split into three parts: a system model, a specification and proof script.

In the example, these three conceptual parts are also kept separate in the code.[1] The following subsections will describe each of these sections of the module.

[1]This is not required by Uclid5 syntax, as `invariant` declarations and assumptions can be interleaved with `init`, `next`, `var` declarations as well other types of declarations. However, keeping these conceptually different parts separate is good design practice. Uclid5 does require that if a `control` block is specified, then it is the very last element of a module.

```
1  module main {
2    // Part 1: System description.
3    var a, b : integer;
4
5    init {
6      a = 0;
7      b = 1;
8    }
9    next {
10     a', b' = b, a + b;
11   }
12
13   // Part 2: System specification.
14   invariant a_le_b: a <= b;
15
16   // Part 3: Proof script.
17   control {
18     unroll (3);
19     check;
20     print_results;
21   }
22 }
```

Example 1.1.: A UCLID5 model that computes the Fibonacci sequence

The System Model

This part of a UCLID5 module describes the functionality of the transition system that is being modeled: it tells us *what the system does*.

The first item of interest within the module `main` are *state variables*. These are declared using the `var` keyword. The module `main` declares two state variables: `a` and `b` on line 3. These are both of type `integer`, which corresponds to mathematical integers.[2]

The `init` block appears next and spans lines 5 to 8. It defines the initial values of the state variables in the module. The notation `a'` refers to the value of the state variable `a` at the end of the current "step", which in this case refers to initial state. The model is specifying that after the `init` block is executed, `a` and `b` have the values 0 and 1 respectively.

The `next` block appears after this and it defines the transition relation of the module. In the figure, the next statement spans from lines 9 to 11; `a` is assigned to the (old) value of `b`, while `b` is assigned to the value `a` + `b`.

The System Specification

The specification answers the question: *what is the system supposed to do?*.

[2]Mathematical integer types, as opposed to the machine integer types present in languages like C/C++ and Java, do not have a fixed bit-width and do not overflow.

In our example, we have a single invariant that comprises that entire specification. Line 14 defines this invariant. It is named a_le_b and as the name suggests, it states that a must be less than or equal to b for every reachable state of the system.

The Proof Script

The third and final part of the UCLID5 module is a set of commands to the UCLID5 verification engine. These tell how we should go about proving[3] that the system satisfies its specification.

The proof script is contained within the control block. The commands here execute the system for 3 steps and check whether all of the systems properties (in this case, we only have one invariant: a_le_b) are satisfied for each of these steps.

The command unroll executes the system for 3 steps. This execution generates four *proof obligations*. These proof obligations ask whether the system satisfies the invariant a_le_b in the initial state and in each of the 3 states reached next. The check command *checks* whether these proof obligations are satisfied and the print_results prints out the results of these checks.

1.2. Installing Uclid5

Public releases of the UCLID5 can be obtained at: https://github.com/uclid-org/uclid/releases. For the impatient, the short version of the installation instructions is: download the archive with the latest release, unzip the archive and add the 'bin/' subdirectory to your PATH.

More detailed instructions for installation are as follows.

1.2.1. Prerequisites

UCLID5 has two prerequisites.

1. UCLID5 requires that the Java™ Runtime Environment be installed on your machine. You can download the latest Java Runtime Environment for your platform from https://www.java.com.com.

2. UCLID5 uses the Z3 SMT solver. You can install Z3 from: https://github.com/Z3Prover/z3/releases. Make sure the 'z3' or 'z3.exe' binary is in your path after Z3 installed. Also make sure, the shared libraries for libz3 and libz3java are in the dynamic library load path (LD_LIBRARY_PATH on Unix-like systems).

UCLID5 has been tested with Java™ SE Runtime Environment version 1.8.0 and Z3 versions 4.5.1 and 4.6.0.

[3] We are using a broad definition of the word "prove" here to refer to any systematic method that gives us assurance that the specification is satisfied.

1.2.2. Detailed Installation Instructions

First, down the platform independent package from `https://github.com/uclid-org/uclid/releases`.

Next, follow these instructions which are provided for the bash shell running on a Unix-like platform. Operations for Micosoft Windows, or a different shell should be similar.

- Unzip the archive.

  `$ unzip uclid-0.9.1.zip.`

- Add the `uclid` binary to your path.

  `$ export PATH=$PATH:$PWD/uclid-0.9.1/bin/`

- Check that the `uclid` works.

  `$ uclid`

  This should produce output similar to the following.

  ```
  $ uclid

      Usage: uclid [options] filename [filenames]
      Options:
        -h/--help : This message.
        -m/--main : Set the main module.
        -d/--debug : Debug options.

  Error : Unable to find main module.
  ```

1.2.3. Running Uclid5

Invoke UCLID5 on a model is easy. Just run the `uclid` binary and provide a list of files containing the model as a command-line argument. When invoked, UCLID5 will parse each of these files and look for a module named `main` among them. It will execute the commands in the `main` module's control block. The `--main` command line argument can be used to specify a different name for the "main" module. Note only the `main` module's control blocks will be executed, even if the `main` module instantiates other modules with control blocks. If no `main` module is found, UCLID5 will exit with an error, as we saw in the previous section when `uclid` was invoked without arguments.

Example 1.1 is part of the UCLID5 distribution in the `examples/tutorial/` subdirectory. You can run UCLID5 on this model as:

```
$ uclid examples/tutorial/ex1.1-fib-model.ucl
```

This should produce the following output.

```
Successfully parsed 1 and instantiated 1 module(s).
4 assertions passed.
0 assertions failed.
0 assertions indeterminate.
Finished execution for module: main.
```

1.3. Looking Forward

This chapter has provided an brief overview of UCLID5's features and toolchain. The rest of this tutorial will take a more detailed looked at more of UCLID5's features.

2. Basics: Types and Statements

This chapter will provide an overview of Uclid5's type system and modelling features. Let us start with Example 2.1, a model of a simple arithmetic logic unit (ALU).

2.1. Types in Uclid5

Types supported by Uclid5 are of the following kinds:

1. `integer`: the type of mathematical integers.
2. `boolean`: the Boolean type. This type has two values: `true` and `false`.
3. `bvW`: The family of bit-vector types parameterized by their width (W).
4. `enum`: enumerated types.
5. Tuples and records.
6. Array types.
7. Uninterpreted types.

An enumerated type is used in line 2 of Example 2.1. This declares a *type synonym*: `cmd_t` is an alias for the enumerated type consisting of three values: `add`, `sub` and `mov_imm`. The input `cmd` is then declared to be of type `cmd_t` on line 8.

The input `valid` is of type `boolean`. Register indices `r1` and `r2` are bit-vectors of width 3 (`bv3`), while `immed`, `result`, `r1val` and `r2val` are bit-vectors of width 8 (`bv8`).

Line 3 declares a type synonym for a `record`. It declares `result_t` as consisting of two fields: a Boolean field `valid` and a bit-vector field `value`. The output `result` is declared to be of type `result_t` on Line 9.

The final point of interest, type-wise, is line 10. The state variable `regs` is declared to be of type array: indices to the array are of type `bv3` and elements of the array are of type `bv8`. This is used to model an 8-entry register file, where each register is a bit-vector of width 8.

2.2. Statements in Uclid5

Computation in Uclid5 can be either procedural or parallel. Procedural computation is performed by defining a `procedure` (and in the `init` block) while parallel computation occurs in the `next` block.

```
module main {
  type cmd_t = enum { add, sub, mov_imm };
  type result_t = record { valid : boolean, value : bv8 };

  input  valid  : boolean;
  input  cmd    : cmd_t;
  input  r1, r2 : bv3;
  input  immed  : bv8;
  output result : result_t;
  var    regs   : [bv3]bv8;
  var    cnt    : bv8;

  procedure set_init_state()
    modifies regs, cnt, result;
  {
    for i in range(0bv3, 7bv3) { regs[i] = 1bv8; }
    cnt, result.value = 1bv8, 1bv8;
  }

  init {
    call set_init_state();
  }

  procedure exec_cmd()
    returns (r : result_t)
    modifies regs;
  {
    var r1val, r2val  : bv8;
    if (valid) {
      r1val, r2val = regs[r1], regs[r2];
      case
        (cmd == add)      : { regs[r1] = r1val + r2val; }
        (cmd == sub)      : { regs[r1] = r1val - r2val; }
        (cmd == mov_imm)  : { regs[r1] = immed; }
      esac
      r.valid, r.value = true, regs[r1];
    } else { r.valid = false; }
  }

  next {
    call (result') = exec_cmd();
    cnt' = cnt + cnt;
  }

  assume regindex_zero    : (r1 == 0bv3 && r2 == 0bv3);
  assume cmd_is_add       : (cmd == add) && valid;
  invariant result_eq_cnt : (cnt == result.value);

  control {
    f = unroll (5);
    check;
    print_results;
  }
}
```

Example 2.1.: Model of a simple ALU

2.2.1. Parallel vs. Procedural Assignments

Assignments inside procedures and the `init` block are called **procedural assignments** and must be of the form `variable = expression;` Assignments inside `next` block are **parallel assignments** and must be of the form `variable' = expression;`. Mathematically, parallel assignments compute the next state of the transition system described by the model.

An example showing the use of sequential assignments is the following:

```
  x = 1;
  x = x + 2;
  x = x + 3;
```

Recall that these procedural assignments *must* appear inside a procedure or in the `init` block. Executing these three statements will result `x` having the value 6.

In contrast, the following sequence of parallel assignments is **not** allowed and will result in a compiler error.

```
  // Error, will not compile.
  x' = 1;
  x' = x + 2;
  x' = x + 3;
```

Only a single parallel assignment to a state/output variable is allowed in a code block. Furthermore, since parallel assignments are computed in data-flow order, the order in which they are specified does not matter. This means that the following two snippets of code are equivalent:

```
next {
  x' = x + 1;
  y' = x' + 1;
}
```

```
next {
  y' = x' + 1;
  x' = x + 1;
}
```

UCLID5 will determine that since `y'` depends on the value of `x'`, `x'` has to be computed first. This value is then used in the computation of `y'`. This is regardless of the order in which these assignments appear in the `next` block.

Note also that the assignment to `x'` uses the value of the variable `x` *at the beginning of the current step* of the transition system (i.e., the "old" value of `x`). In contrast the assignment to `y'` uses the "new" value of `x`, which is the value of `x` at the *end* of this step of the transition system. It is important to think carefully about which version of a variable (`var` or `var'`) must be used in a particular assignment.

2.2.2. Procedures

Example 2.1 demonstrates how sequential computation is used in concert with parallel computation. The `procedure set_init_state` (lines 13–18) is used to initialize the values of the registers (state variable `regs`), the variables `cnt` and `result`. Since this procedure updates the module's state variables, a `modifies` clause (line 14) is required to explicitly specify that these updates are intended. The procedure is called on line 21 in the `init` block. Updates to a state variable not mentioned in a modifies clause will result in a compilation error.

Procedures can also be used inside `next` blocks to perform next-state assignments to state/output variables, in addition to making procedural assignments. In Example 2.1, consider `procedure exec_cmd` which executes a single ALU command and returns (line 25) a single value of type `result_t`. The procedure is invoked on line 41 in the `next` block, and its return value is assigned to the output variable `result`. Note we are again using the notation `result'` to refer to parallel assignment.

We emphasize that assignments inside procedures do not assign primed variables. However, if a state variable is defined to be modified by a procedure (mentioned in its modifies clause), then its next-state value is the value that variable has upon return from that procedure. Put another way, the post-state of the procedure determines the next-state assignment of all state variables modified by it. In our example, `procedure exec_cmd` modifies the state variable `regs`, and thus determines its next-state value.

2.2.3. For Loops

The procedure `set_init_state` uses a `for` loop to initialize each value in the array `regs` to the bit-vector value 1.[1] The loop iterates over the values between 0 and 7 (both-inclusive).

The range over which a `for` loop iterates must be defined by two numeric literals.

2.2.4. If and Case Statements

Also worth pointing out are the `if` statement that appears on line 28, and the `case` statement that appears on line 31. Syntax for `if` statements should be familiar.

`case` statements are delimited by `case` and `esac` and contain within them a list of boolean expressions and associated statement blocks. These expressions are evaluated in the order in which appear, and if any of them evaluate to `true`, the corresponding block is executed. If none of the case-expressions evaluate to `true`, nothing is executed. The keyword `default` can be used as a "catch-all" case like in C/C++.

2.2.5. Expressions

The syntax for expressions in UCLID5 is similar to languages like C/C++/Java. Index `i` of array `regs` is accessed using the syntax `regs[i]`. Field `value` in the record `result` is accessed as `result.value`.

[1] `1bv8` here refers to the bit-vector value 1 of width 8.

2.3. Verification Model

This section briefly describes the execution semantics of Example 2.1.

2.3.1. Initialization

Execution of the model in Example 2.1 starts with the `init` block. This block invokes `set_init_state` and assigns initial values to `regs`, `cnt` and `result.value`. The other variables (e.g. `r1val` and `r2val`) are not assigned to in the `init` block and will be initialized non-deterministically.

2.3.2. Next State Computation

The next state of each state variable in the model is computed according to the `next` block. Any variables not assigned to in the `next` block retain their "old" values.

The `input` variables of the model are assigned (possibly different) non-deterministic values for each step of the transition system. These values can be controlled by using assumptions. Indeed, the model uses the three assumptions on lines 45–46 to constrain the input to the ALU to always be an `add` operation, where both operands refer to register index 0.

2.3.3. Verification

As in Example 1.1, the verification script in Example 2.1 unrolls the transition system for 5 steps and checks if the `invariant` on line 47 is violated in any of these steps.

2.3.4. Running Uclid5

Running UCLID5 on Example 2.1 produces the following output.

```
$ uclid examples/tutorial/ex2.1-alu.ucl
Successfully parsed 1 and instantiated 1 module(s).
6 assertions passed.
0 assertions failed.
0 assertions indeterminate.
Finished execution for module: main.
```

UCLID5 is able to prove that the `invariant` on line 46 holds for all states reachable within 5 steps of the initial state, under the assumptions specified in lines 45–46.

3. Verification Techniques

In the examples covered thus far, we have only used UCLID5 for bounded model checking of invariants. UCLID5 can also be used to do unbounded inductive proofs and also provides support for debugging counterexamples. This chapter will describe these features of UCLID5. Further features are being implemented and will be described in a future version of this document.

3.1. Inductive Proofs

Let us revisit the model from Example 1.1. This is now shown again in Example 3.1, but with a different proof script. Instead of using the `unroll` command for bounded model checking, we are using the `induction` command to attempt an inductive proof.

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

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

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

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

Example 3.1.: UCLID5 Fibonacci model using induction in the proof script

3.1.1. Debugging Counterexamples

Let us try running UCLID5 on Example 3.1 with the new proof script.

```
$ uclid examples/tutorial/ex3.1-fib-induction.ucl
Successfully parsed 1 and instantiated 1 module(s).
1 assertions passed.
1 assertions failed.
0 assertions indeterminate.
  FAILED -> induction (step) [Step #1]
  property a_le_b @ ex3.1-fib-induction.ucl, line 14
Finished execution for module: main.
```

Uh oh, we seem to have a problem! UCLID5 is telling us that the inductive proof failed. We can try to examine why the proof failed by using the `print_cex` command to examine the counterexample to the proof.

```
1  module main {
2    // Part 1: System description.
3    var a, b : integer;
4
5    init {
6      a = 0;
7      b = 1;
8    }
9    next {
10     a', b' = b, a + b;
11   }
12
13   // Part 2: System specification.
14   invariant a_le_b: a <= b;
15
16   // Part 3: Proof script.
17   control {
18     vobj = induction;
19     check;
20     print_results;
21     vobj.print_cex(a, b);
22   }
23 }
```

Example 3.2.: UCLID5 Fibonacci model with `induction` and `print_cex`

The only changes between Example 3.1 and Example 3.2 are on lines 18 and 21. `vobj` on line 18 is a reference to the verification conditions generated by the `induction` command. On line 21, we pass this reference to the `print_cex` command which prints out the values of `a` and `b` for the counterexample.

Running UCLID5 on Example 3.2 produces the following.

```
Successfully parsed 1 and instantiated 1 module(s).
1 assertions passed.
1 assertions failed.
0 assertions indeterminate.
  FAILED -> vobj: induction (step) [Step #1]
  property a_le_b @ ex3.2-fib-induction-cex.ucl, line 14
CEX for vobj: induction (step) [Step #1]
property a_le_b @ ex3.2-fib-induction-cex.ucl, line 14
=================================
Step #0
  a : -1
  b : 0
=================================
=================================
Step #1
  a : 0
  b : -1
=================================
Finished execution for module: main.
```

To understand the counterexample, it is helpful to review how the inductive proof engine works. When inductively proving the `invariant` `a_le_b`, UCLID5 considers some arbitrary state that satisfies this property, executes the `next` block, and checks whether `a_le_b` holds on the resultant state.

The counterexample shows us that we do start in a state where $\mathtt{a} \leq \mathtt{b}$ with $\mathtt{a} = -1$ and $\mathtt{b} = 0$. We execute the `next` block and now `a` gets the value of `b`, becoming 0 and `b` gets the value $\mathtt{a} + \mathtt{b}$, becoming -1. This new state does not satisfy the invariant!

What is the real problem here? Taking a closer look at Example 3.2, we see that this specific counterexample can never occur in our model because `a` and `b` are always ≥ 0. But UCLID5 does not know this when attempting the inductive proof. Therefore, we have to strengthen the inductive argument with this information in order to help UCLID5's proof.

3.1.2. Inductive Proof for the Fibonacci Model

Example 3.3 shows the same model as Example 3.2, but with a stronger induction hypothesis. UCLID5's inductive engine will now start in an arbitrary state that assumes that both invariants `a_le_b` and `a_b_ge_0` hold and attempt to prove that both of these still hold after the `next` block is executed.

Let us now run UCLID5 on this new model.

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

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

  // Part 2: System specification.
  invariant a_le_b: a <= b;
  invariant a_b_ge_0: (a >= 0 && b >= 0);

  // Part 3: Proof script.
  control {
    vobj = induction;
    check;
    print_results;
    vobj.print_cex(a, b);
  }
}
```

Example 3.3.: Inductive proof for the Fibonacci model

```
Successfully parsed 1 and instantiated 1 module(s).
$ uclid examples/tutorial/ex3.3-fib-induction-proof.ucl
4 assertions passed.
0 assertions failed.
0 assertions indeterminate.
Finished execution for module: main.
```

Success! We have shown that our system model satisfies its specification.

3.2. Bounded Model Checking

Let us return to the model of Example 1.1 which is reproduced as Example 3.4 with a few changes. We used the `unroll` command for verification. This command performs bounded model checking and takes a single argument – the number of steps to unroll the model for. In Example 3.4, we are unrolling the model for 3 steps. We have introduced the constant `flag` on line 4. A constant holds a symbolic value that does not change during computation. The initial value of the constant is assigned non-deterministically and can be controlled using assumptions.

3.2.1. Embedded assume and assert statements

A second difference with between Example 1.1 and Example 3.4 is on lines 12–14, 24 and 25. Instead of using a module-level assumption declarations as in Example 1.1, we have three embedded assumptions in the `set_init` procedure on lines 12–14, and two embedded assertions in the `next` block on lines 23 and 25. A module-level assumption is assumed to hold for the solver at every step of execution, while an embedded assumption is assumed "instantaneously." In particular, the assumptions on lines 12–14 tells the solver to assume that `a` $\leq$ `b`, `a >= 0` and `b >= 0` at the end of the `set_init` procedure. Notice that we are not assigning specific values to `a` and `b`, instead we are asking UCLID5 to consider potential values of `a` and `b` such that `a` $\leq$ `b`, `a` ≥ 0 and `b` ≥ 0.

Similarly the assertions on lines 23 and 25 are evaluated at that specific location in the code. In particular the assertion on line 23 is only checked when `flag` is `true`, while the assertion one line 25 is checked when `flag` is `false`. Since `flag` is always `true` in our model, the assertion on line 25 will never fire. In contrast, note that a module-level assertion would be evaluated after the `init` block and after each execution of the `next` block.

3.2.2. Running Uclid5

Running UCLID5 on Example 3.4 shows that the embedded assertions do indeed hold for all states reachable within 3 steps of the initial state.

```
module main {
  // System description. blah
  var a, b   : integer;
  const flag : boolean;

  procedure set_init()
    modifies a, b;
  {
    havoc a;
    havoc b;
    // embedded assumptions.
    assume (a <= b);
    assume (a >= 0 && b >= 0);
    assume (flag);
  }

  init {
    call set_init();
  }
  next {
    a', b' = b, a + b;
    if (flag) {
      assert (a' <= b');
    } else {
      assert (false);
    }
  }

  // Proof script.
  control {
    unroll (3);
    check;
    print_results;
  }
}
```

Example 3.4.: Revisiting the Fibonacci model from Example 1.1.

```
$ uclid examples/tutorial/ex3.4-fib-model-revisted.ucl
Successfully parsed 1 and instantiated 1 module(s).
6 assertions passed.
0 assertions failed.
0 assertions indeterminate.
Finished execution for module: main.
```

3.3. Specifications in Linear Temporal Logic

UCLID5 supports the specification of module behavior using linear temporal logic (LTL).

Example 3.5 shows a UCLID5 model of an intersection with two traffic lights. Lines 3–39 define the functionality of the traffic light; this part of the model should be familiar. The current state of the lights are stored in the variables `light1` and `light2`, and these switch from `red` to `green` to `yellow` and back to `red`. The variables `step1` and `step2` can be thought of timers, and ensure that each light stays red for three transitions, green for two transitions and stays yellow for a single transition.

The LTL properties are on lines 41, 42 and 43. The property `always_one_red` specifies a safety property which states that at least one of the two lights must be `red` in every particular cycle. The notation $\mathtt{G}(\phi)$ refers to the LTL globally operator, while the notation $\mathtt{F}(\phi)$ refers to the LTL eventually (future) operator. Other supported operators include next-time: $\mathtt{X}(\phi)$, (strong-)until: $\mathtt{U}(\phi_1, \phi_2)$ and weak-until: $\mathtt{W}(\phi_1, \phi_2)$. `always_one_red` is safety property. The property `eventually_green` is an example of liveness property, and specifies that both lights become `green` infinitely often.

The command for bounded verification of LTL properties is the `bmc` command. This is invoked on line 46 and specifies which properties must be checked within the square brackets. (If no properties are specified, and the square brackets are omitted `bmc` checks all LTL properties in the module.)

3.3.1. Running Uclid5

Running UCLID5 on Example 3.5 produces the following output.

```
$ uclid run examples/traffic-light.ucl
Running (fork) uclid.UclidMain examples/traffic-light.ucl
Successfully parsed 1 and instantiated 1 module(s).
44 assertions passed.
0 assertions failed.
0 assertions indeterminate.
Finished execution for module: main.
```

The output shows that all properties are verified.

```
module main
{
  type light_t = enum { red, yellow, green };
  var step1, step2   : integer;
  var light1, light2 : light_t;

  init {
    light1, step1 = red,   2;
    light2, step2 = green, 1;
  }

  next {
    call (light1', step1') = next_light(light1, step1);
    call (light2', step2') = next_light(light2, step2);
  }

  procedure next_light(light : light_t, step : integer)
   returns (lightP : light_t, stepP: integer)
  {
    if (step == 0) {
      case
      (light == green) : {
        lightP = yellow;
        stepP = step;
      }
      (light == yellow) : {
        lightP = red;
        stepP = 2;
      }
      (light == red) : {
        lightP = green;
        stepP = 1;
      }
    esac
    } else {
      lightP = light;
      stepP = step - 1;
    }
  }

  property[LTL] always_one_red: G(light1 == red || light2 == red);
  property[LTL] eventually_green:
    G(F(light1 == green)) && G(F(light2 == green));

  control {
    v = bmc[always_one_red, eventually_green](10);
    check;
    print_results;
    v.print_cex(light1, step1, light2, step2);
  }
}
```

Example 3.5.: Example of using LTL specifications in UCLID5.

Exercises

1. Does the property always_one_red hold if the assignment to step2 on line 9 is changed to 2 (from 1)? Why or why not? Make this change, print-out and understand the counterexample if one exists.

2. Find a way to modify the model so that the property eventually_green is violated. Examine and understand the counter-example generated by UCLID5 when this happens.

3.4. Future Directions

Future versions of UCLID5 will have support for synthesizing invariants using Syntax-Guided Synthesis (SyGuS).

4. Compositional Modeling and Abstraction

This chapter describes Uclid5's features for compositional and modular verification, and the use of abstraction.

We will use a running example of a CPU model constructed in the Uclid5 and use bounded model checking to prove that the execution of this CPU is deterministic: i.e. we show that given two identical instruction memories, the state updates performed by this CPU will be identical.

4.1. Common Type Definitions Across Modules

Example 4.1 shows a module that defines only type synonyms. Such a module can be used to share type definitions across other modules. The types declared in Example 4.1 are *imported* in lines 2-5 of module `cpu` declared in Example 4.2.

```
// This module declares types that are used in the rest of the model.
module common {
  // addresses are uninterpreted types.
  type addr_t = bv8;
  type word_t = bv8;
  // memory
  type mem_t = [addr_t]word_t;
  // CPU operation.
  type op_t   = enum { op_mov, op_add, op_sub, op_load, op_store };
}
```

Example 4.1.: Module `common` of the CPU model

Isolating commonly used types into a single module in this manner allows the construction of large models parameterized by this types. These common types can be changed and the ramifications of these changes on the model's behavior can be studied easily.

4.2. Uninterpreted Functions and Types

A convenient mechanism for abstraction in Uclid5 is through the use of uninterpreted functions and types. This is one of the novel modeling aspects for transition systems introduced by the original UCLID system [2].

```
module cpu {
  type addr_t = common.addr_t;
  type mem_t  = common.mem_t;
  type word_t = common.word_t;
  type op_t   = common.op_t;
  type regindex_t; // type of register file.
  type regs_t = [regindex_t]word_t;

  input imem          : mem_t; // program memory.
  var dmem            : mem_t; // data memory.
  var regs            : regs_t;
  var pc              : addr_t;
  var inst, result    : word_t;

  function inst2op   (i : word_t) : op_t;
  function inst2reg0 (i : word_t) : regindex_t;
  function inst2reg1 (i : word_t) : regindex_t;
  function inst2imm  (i : word_t) : word_t;
  function inst2addr (i : word_t) : addr_t;

  procedure exec_inst(inst : word_t, pc : addr_t)
    returns (result : word_t, pc_next : addr_t)
    modifies regs, dmem;
  {
    var op            : op_t;
    var r0ind, r1ind : regindex_t;
    var r0, r1        : word_t;

    op = inst2op(inst);
    r0ind, r1ind = inst2reg0(inst), inst2reg1(inst);
    r0, r1 = regs[r0ind], regs[r1ind];
    case
      (op == op_mov)      : { result = inst2imm(inst); }
      (op == op_add)      : { result = r0 + r1; }
      (op == op_sub)      : { result = r0 - r1; }
      (op == op_load)     : { result = dmem[inst2addr(inst)]; }
      (op == op_store)    : { result = r0; dmem[inst2addr(inst)] = r0; }
    esac
    pc_next = pc + 1bv8;
    regs[r0ind] = result;
  }

  init {
    assume (forall (r : regindex_t) :: regs[r] == 0bv8);
    assume (forall (a : addr_t) :: dmem[a] == 0bv8);
    pc, inst = 0bv8, 0bv8;
  }

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

Example 4.2.: The cpu module in the CPU model

```
module main {

  // Import types
  type addr_t     = common.addr_t;
  type mem_t      = common.mem_t;
  type word_t     = common.word_t;
  type op_t       = common.op_t;
  type regindex_t = cpu.regindex_t;

  // instruction memory is the same for both CPUs.
  var imem : mem_t;

  // Create two instances of the CPU module.
  instance cpu_i_1 : cpu(imem : (imem));
  instance cpu_i_2 : cpu(imem : (imem));

  init {
  }

  next {
    // Invoke CPU 1 and CPU 2.
    next (cpu_i_1);
    next (cpu_i_2);
  }

  // These are our properties.
  invariant eq_regs :
    (forall (ri : regindex_t) :: cpu_i_1.regs[ri] == cpu_i_2.regs[ri]);
  invariant eq_mem  :
    (forall (a : addr_t) :: cpu_i_1.dmem[a] == cpu_i_2.dmem[a]);
  invariant eq_pc   : (cpu_i_1.pc == cpu_i_2.pc);
  invariant eq_inst : (cpu_i_1.inst == cpu_i_2.inst);

  // Proof script.
  control {
    unroll(3);
    check;
    print_results;
  }
}
```

Example 4.3.: Module `main` in the CPU model

4.2.1. Uninterpreted Types

Example 4.2 shows the use of the *uninterpreted type*: `regindex_t` on line 6. The index type to the register is an *abstract* type, as opposed to a specific type (e.g. `bv3`). This allows us to reason about an abstract register file that has an undefined (and unbounded) number of entries, as opposed to proving facts about some specific register file implementation, potentially enabling more general proofs about system behavior.

4.2.2. Uninterpreted Functions

Values belonging to an uninterpreted type can be created using *uninterpreted functions*. The functions `inst2op`, `inst2reg0`, `inst2reg1`, `inst2imm` and `inst2addr` on lines 15-19 of Example 4.2 are all examples of uninterpreted functions. An uninterpreted function f is a function about which we know nothing, except that it is a function; i.e. $\forall x_1, x_2.\ x_1 = x_2 \implies f(x_1) = f(x_2)$.

Uninterpreted functions allow us to reason about an abstract CPU model without considering specific instruction encodings or decoder models. This could potentially lead to more general proofs as well as more scalable automated proofs.

4.3. Module Instantiation and Scheduling

Modules are instantiated using the `instance` keyword. Lines 14 and 15 of Example 4.3 show two instantiations of the module `cpu`. For each instance, the module input `imem` is mapped to the state variable `imem` of module `main`.

Scheduling of instantiated modules is explicit and synchronous. The two `next` statements on lines 22 and 23 of Example 4.3 invoke the next state transitions of the two instances of the `cpu` module.

4.3.1. Accessing Instance Variables

The state variables of an instantiated module are accessed using the `.` operator. The four invariants on lines 27-32 of Example 4.3, refer to the registers, memory, pc and instruction variables of the two instantiated modules. These invariants state that both instances must have identical values for these state variables.

4.4. Running Uclid5

Executing Uclid5 on the complete CPU model shows that CPU is in fact deterministic.

```
$ uclid ex4.1-cpu.ucl ex4.2-cpu.ucl ex4.3-cpu.ucl
Successfully parsed 3 and instantiated 1 module(s).
16 assertions passed.
0 assertions failed.
0 assertions indeterminate.
Finished execution for module: main.
```

Note the file `ex4.1-cpu.ucl` contains three modules. Each of these modules could potentially have `control` blocks. Which UCLID5 is invoked on this model, it executes only the `control` block of the `main` module. If we had included a `control` block for the `alu` module and wished to verify properties of this module, we would have to invoke UCLID5 on this specific module using the `--main` command-line option.

4.4.1. Exercise: Inductive Proof of CPU model

Prove determinism of the CPU model using induction rather than bounded model checking. You will need to add strengthening inductive invariants relating the two CPU instances.

Bibliography

[1] UCLID Verification System. Available at `http://uclid.eecs.berkeley.edu`.

[2] 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. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

A. Appendix: Uclid5 Grammar

This appendix describes UCLID5's grammar.

A.1. Grammar of Modules and Declarations

A model consist of a list of modules. Each module consists of a list of declarations followed by an optional control block.

⟨*Model*⟩ ::= ⟨*Module*⟩*

⟨*Module*⟩ ::= `module` ⟨*Id*⟩ '`{`' ⟨*Decl*⟩* ⟨*ControlBlock*⟩? '`}`'

Declarations can be of the following types.

⟨*Decl*⟩ ::= ⟨*TypeDecl*⟩
| ⟨*InputsDecl*⟩
| ⟨*OutputsDecl*⟩
| ⟨*VarsDecl*⟩
| ⟨*ConstsDecl*⟩
| ⟨*SharedVarsDecl*⟩
| ⟨*FuncDecl*⟩
| ⟨*ProcedureDecl*⟩
| ⟨*InstanceDecl*⟩
| ⟨*InitDecl*⟩
| ⟨*NextDecl*⟩
| ⟨*AxiomDecl*⟩
| ⟨*SpecDecl*⟩

Type declarations declare either a type synonym or an uninterpreted type.

⟨*TypeDecl*⟩ ::= `type` ⟨*Id*⟩ '=' ⟨*Type*⟩ '`;`'
| `type` ⟨*Id*⟩ '`;`'

Variable declarations can refer to inputs, outputs, state variables or shared variables.

⟨*InputsDecl*⟩ ::= `input` ⟨*IdList*⟩ '`:`' ⟨*Type*⟩ '`;`'

⟨*OutputsDecl*⟩ ::= `output` ⟨*IdList*⟩ '`:`' ⟨*Type*⟩ '`;`'

⟨*VarsDecl*⟩ ::= `var` ⟨*IdList*⟩ '`:`' ⟨*Type*⟩ '`;`'

⟨*ConstsDecl*⟩ ::= `const` ⟨*IdList*⟩ ‘`:`’ ⟨*Type*⟩ ‘`;`’

⟨*SharedVarsDecl*⟩ ::= `sharedvar` ⟨*IdList*⟩ ‘`:`’ ⟨*Type*⟩ ‘`;`’

⟨*DefineDecl*⟩ ::= `define` ⟨*Id*⟩ ‘`(`’ ⟨*IdTypeList*⟩ ‘`)`’ ‘`:`’ ⟨*Type*⟩ ‘=’ ⟨*Expr*⟩ ‘`;`’

Function declarations refer to uninterpreted functions.

⟨*FuncDecl*⟩ ::= `function` ⟨*Id*⟩ ‘`(`’ ⟨*IdTypeList*⟩ ‘`)`’ ‘`:`’ ⟨*Type*⟩ ‘`;`’

Procedure declarations consist of a formal parameter list, a list of return values and types, followed by optional pre-/post-conditions and the list of state variables modified by procedure.

⟨*ProcedureDecl*⟩ ::= `procedure` ⟨*Id*⟩ ‘`(`’ ⟨*IdTypeList*⟩ ’)’ ⟨*ProcReturnArg*⟩?
⟨*RequireExprs*⟩ ⟨*EnsureExprs*⟩ ⟨*ModifiesExprs*⟩
‘`{`’ ⟨*VarsDecls*⟩* ⟨*Statement*⟩* ‘`}`’

⟨*ProcReturnArg*⟩ ::= `returns` ‘`(`’ ⟨*IdTypeList*⟩ ‘`)`’

⟨*RequireExprs*⟩ ::= (`requires` ⟨*Expr*⟩ ‘`;`’)*

⟨*EnsureExprs*⟩ ::= (`ensures` ⟨*Expr*⟩ ‘`;`’)*

⟨*ModifiesExprs*⟩ ::= (`modifies` ⟨*IdList*⟩ ‘`;`’)*

Instance declarations allow the instantiation (duh!) of other modules. They consist of the instance name, the name of the module being instantiated and the list of mappings for the instances’ inputs, output and shared variables.

⟨*InstanceDecl*⟩ ::= `instance` ⟨*Id*⟩ ‘`:`’ ⟨*Id*⟩ ⟨*ArgMapList*⟩ ‘`;`’

⟨*ArgMapList*⟩ ::= ‘`(`’ ‘`)`’
| ‘`(`’ ⟨*ArgMap*⟩ ‘`,`’ ⟨*ArgMapList*⟩ ‘`)`’

⟨*ArgMap*⟩ ::= ⟨*Id*⟩ ‘`:`’ ‘`(`’ ‘`)`’
| ⟨*Id*⟩ ‘`:`’ ‘`(`’ ⟨*Expr*⟩ ‘`)`’

Axioms refer to assumptions while a **specification declaration** refers to design `invariants`. Note `axiom` and `assume` are synonyms, as are `property` and `invariant`.

⟨*AxiomDecl*⟩ ::= ⟨*AxiomKW*⟩ ⟨*Id*⟩ ‘`:`’ ⟨*Expr*⟩ ‘`;`’
| ⟨*AxiomKW*⟩ ⟨*Expr*⟩ ‘`;`’

⟨*AxiomKW*⟩ ::= `axiom` | `assume`

⟨*SpecDecl*⟩ ::= ⟨*PropertyKW*⟩ ⟨*Id*⟩ ‘`:`’ ⟨*Expr*⟩ ‘`;`’
| ⟨*PropertyKW*⟩ ⟨*Expr*⟩ ‘`;`’

⟨*PropertyKW*⟩ ::= `property` | `invariant`

Init and **next** blocks consist of lists of statements.

⟨*InitDecl*⟩ ::= `init` ‘`{`’ ⟨*Statement*⟩* ‘`}`’

⟨*NextDecl*⟩ ::= `next` ‘`{`’ ⟨*Statement*⟩* ‘`}`’

Assignment statements in the `next` block declaration must assign primed variables only, and are concurrently evaluated.

A.2. Statement Grammar

Statements are the following types, most of which should be familiar. Note the support for simultaneous assignment à la Python. The keyword `next` allows for synchronous scheduling of instantiated modules.

⟨*Statement*⟩ ::= `skip` ‘`;`’
| `assert` ⟨*Expr*⟩ ‘`;`’
| `assume` ⟨*Expr*⟩ ‘`;`’
| `havoc` ⟨*Id*⟩ ‘`;`’
| ⟨*LhsList*⟩ ‘=’ ⟨*ExprList*⟩ ‘`;`’
| `call` ‘`(`’ ⟨*LhsList*⟩ ‘`)`’ ‘=’ ⟨*Id*⟩ ⟨*ExprList*⟩ ‘`;`’
| `next` ‘`(`’ ⟨*Id*⟩ ‘`)`’ ‘`;`’
| ⟨*IfStmt*⟩
| ⟨*CaseStmt*⟩
| ⟨*ForLoop*⟩

Assignments and **call** statements refer to the nonterminal ⟨*LhsList*⟩. As the name suggests, this is a list of syntactic forms that can appear on the left hand side of an assignment. ⟨*Lhs*⟩ are of four types: (i) identifiers, bitvector slices within identifiers, (iii) array indices, and (iv) fields within records.

⟨*LhsList*⟩ ::= ⟨*Lhs*⟩ (‘`,`’ ⟨*Lhs*⟩)*

⟨*Lhs*⟩ ::= ⟨*Id*⟩
| ⟨*Id*⟩ ‘’
| ⟨*Id*⟩ ‘`[`’ ⟨*Expr*⟩ ‘`:`’ ⟨*Expr*⟩ ‘`]`’
| ⟨*Id*⟩ ‘`[`’ ⟨*ExprList*⟩ ‘`]`’
| ⟨*Id*⟩ (‘`.`’ ⟨*Id*⟩)+

If statements are as per usual. “Braceless” if statements are not permitted.

⟨*IfStmt*⟩ ::= `if` ‘`(`’ ⟨*IfExpr*⟩ ‘`)`’ ‘`{`’ ⟨*Statement*⟩* ‘`}`’
`else` ‘`{`’ ⟨*Statement*⟩* ‘`}`’
| `if` ‘`(`’ ⟨*IfExpr*⟩ ‘`)`’ ‘`{`’ ⟨*Statement*⟩* ‘`}`’

⟨*IfExpr*⟩ ::= ⟨*Expr*⟩ | *

Case statements are as follows.

⟨*CaseStmt*⟩ ::= `case` ⟨*CaseBlock*⟩* `esac`

⟨*CaseBlock*⟩ ::= ⟨*Expr*⟩ ‘`:`’ ‘`{`’ ⟨*Statement*⟩* ‘`}`’
| `default` ‘`:`’ ‘`{`’ ⟨*Statement*⟩* ‘`}`’

For loops allow iteration over a statically defined range of values.

⟨*ForLoop*⟩ ::= `for` ⟨*Id*⟩ `in range` ‘`(`’ ⟨*Number*⟩ ’,’ ⟨*Number*⟩ ‘`)`’
‘`{`’ ⟨*Statement*⟩* ‘`}`’

A.3. Expression Grammar

Let us turn to **expressions**, which may be quantified.

⟨*Expr*⟩ ::= ⟨*E1*⟩

⟨*E1*⟩ ::= ⟨*E2*⟩
| ‘`(`’ `forall` ‘`(`’ ⟨*IdTypeList*⟩ ‘`)`’ ‘`::`’ E1 ‘`)`’
| ‘`(`’ `exists` ‘`(`’ ⟨*IdTypeList*⟩ ‘`)`’ ‘`::`’ E1 ‘`)`’

The usual logical and bitwise operators are allowed.

⟨*E2*⟩ ::= ⟨*E3*⟩ ‘<==>’ ⟨*E2*⟩ | ⟨*E3*⟩

⟨*E3*⟩ ::= ⟨*E4*⟩ ‘==>’ ⟨*E3*⟩ | ⟨*E4*⟩

⟨*E4*⟩ ::= ⟨*E5*⟩ ‘`&&`’ ⟨*E4*⟩ | ⟨*E5*⟩ ‘`||`’ ⟨*E4*⟩ |
| ⟨*E5*⟩ ‘`&`’ ⟨*E4*⟩ | ⟨*E5*⟩ ‘`|`’ ⟨*E4*⟩ | ⟨*E5*⟩ ‘`^`’ ⟨*E4*⟩
| ⟨*E5*⟩

As are relational operators, bitvector concatentation (++) and arithmetic.

⟨*E5*⟩ ::= ⟨*E6*⟩ ⟨*RelOp*⟩ ⟨*E6*⟩

⟨*RelOp*⟩ ::= ‘>’ | ‘<’ | ‘=’ | ‘`!=`’ | ‘>=’ | ‘<=’

⟨*E6*⟩ ::= ⟨*E7*⟩ ‘++’ ⟨*E6*⟩

⟨*E7*⟩ ::= ⟨*E8*⟩ ‘+’ ⟨*E7*⟩

⟨*E8*⟩ ::= ⟨*E9*⟩ ‘−’ ⟨*E9*⟩

⟨*E9*⟩ ::= ⟨*E10*⟩ ‘$\star$’ ⟨*E10*⟩

The unary operators are arithmetic negation (unary minus), logical negation and bitwise negation of bitvectors.

$\langle E10 \rangle$ ::= $\langle UnOp \rangle$ $\langle E11 \rangle$ | $\langle E11 \rangle$

$\langle UnOp \rangle$ ::= '`-`' | '`!`' | '`~`'

Array select, update and bitvector select operators are defined à la Boogie.

$\langle E11 \rangle$ ::= $\langle E12 \rangle$ '`[`' $\langle Expr \rangle$ ('`,`' $\langle Expr \rangle$)* '`]`'
| $\langle E12 \rangle$ '`[`' $\langle Expr \rangle$ ('`,`' $\langle Expr \rangle$)* = $\langle Expr \rangle$ '`]`'
| $\langle E12 \rangle$ '`[`' $\langle Expr \rangle$ '`:`' $\langle Expr \rangle$ '`]`'
| $\langle E12 \rangle$

Function invocation, record selection, and access to variables in instantiated modules is as follows.

$\langle E12 \rangle$ ::= $\langle E13 \rangle$ '`(`' $\langle ExprList \rangle$ '`)`'
| $\langle E13 \rangle$ ('`.`' $\langle Id \rangle$)+
| $\langle E13 \rangle$ ('`->`' $\langle Id \rangle$)+

And finally, we have the terminal symbols, identifiers, tuples and the if-then-else operator.

$\langle E12 \rangle$ ::= `false` | `true` | $\langle Number \rangle$
| $\langle Id \rangle$ | $\langle Id \rangle$ '`::`' $\langle Id \rangle$
| '`{`' $\langle Expr \rangle$ ('`,`' $\langle Expr \rangle$)* '`}`'
| `if` '`(`' $\langle Expr \rangle$ '`)`' `then` $\langle Expr \rangle$ `else` $\langle Expr \rangle$

A.4. Types

$\langle Type \rangle$ ::= $\langle PrimitiveType \rangle$
| $\langle EnumType \rangle$
| $\langle TupleType \rangle$ | $\langle RecordType \rangle$
| $\langle ArrayType \rangle$
| $\langle SynonymType \rangle$
| $\langle ExternalType \rangle$

Supported primitive types are Booleans, integers and bit-vectors. Bit-vector types are defined according the regular expression 'bv[0-9]+' and the number following 'bv' is the width of the bit-vector.

$\langle PrimitiveType \rangle$ ::= `boolean` | `integer` | $\langle BitVectorType \rangle$

Enumerated types are defined using the `enum` keyword.

$\langle EnumType \rangle$ ::= `enum` '`{`' $\langle IdList \rangle$ '`}`'

Tuple types are declared using curly brace notation.

$\langle TupleType \rangle$ ::= '`{`' $\langle Type \rangle$ ('`,`' $\langle Type \rangle$)* '`}`'

Record types use the keyword `record`.

⟨*Recordtype*⟩ ::= `record` ‘`{`’ ⟨*IdTypeList*⟩ ‘`}`’

Array types are defined using square brackets. The list of types within square brackets defined the array's index type.

⟨*ArrayType*⟩ ::= ‘`[`’ ⟨*Type*⟩ (‘`,`’ ⟨*Type*⟩)* ‘`]`’ ⟨*Type*⟩

Type synonyms are just identifiers, while external types refer to synonym types defined in a different module.

⟨*SynonymType*⟩ ::= ⟨*Id*⟩

⟨*ExternalType*⟩ ::= ⟨*Id*⟩ ‘`::`’ ⟨*Id*⟩

A.5. Control Block

The control block consists of a list of commands. A command can have an optional result object, an optional argument object, an optional list of command parameters and finally an optional list of argument expressions.

⟨*ControlBlock*⟩ ::= `control` ‘`{`’ ⟨*Cmd*⟩* ‘`}`’

⟨*Cmd*⟩ ::= (⟨*Id*⟩ ‘=’)? (⟨*Id*⟩ ‘->’)? ⟨*Id*⟩
(‘`[`’ ⟨*IdList*⟩ ‘`]`’)? ⟨*ExprList*⟩? ‘`;`’

A.6. Miscellaneous Nonterminals

⟨*IdList*⟩, ⟨*IdTypeList*⟩ and ⟨*ExprList*⟩ are non-empty, comma-separated list of identifiers, identifier/type tuples and expressions respectively.

⟨*IdList*⟩ ::= ⟨*Id*⟩
| ⟨*Id*⟩ ‘`,`’ ⟨*IdList*⟩

⟨*IdTypeList*⟩ ::= ⟨*Id*⟩ ‘`:`’ ⟨*Type*⟩
| ⟨*Id*⟩ ‘`:`’ ⟨*Type*⟩ ‘`,`’ ⟨*IdTypeList*⟩

⟨*ExprList*⟩ ::= ⟨*Expr*⟩
| ⟨*Expr*⟩ ‘`,`’ ⟨*ExprList*⟩