

Getting Started with Uclid5

Alpha release, version 0.8

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

January 2018

Contents

1. Introduction	5
1.1. Getting Started: A Simple UCLID5 Model	5
1.2. Installing UCLID5	7
1.2.1. Prerequisites	7
1.2.2. Detailed Installation Instructions	8
1.2.3. Running UCLID5	8
1.3. Looking Forward	9
2. Basics: Types and Statements	10
2.1. Types in UCLID5	10
2.2. Statements in UCLID5	10
2.2.1. For Loops	12
2.2.2. If and Case Statements	12
2.2.3. Expressions	12
2.3. Computation/Verification Model	12
2.3.1. Initialization	12
2.3.2. Next State Computation	12
2.3.3. Verification	13
2.3.4. Running UCLID5	13
3. Verification Techniques	14
3.1. Inductive Proofs	14
3.1.1. Debugging Counterexamples	14
3.1.2. Inductive Proof for the Fibonacci Model	16
3.2. Bounded Model Checking	17
3.2.1. Embedded assume and assert statements	17
3.2.2. Running UCLID5	18
3.3. Future Directions	19
4. Compositional Modeling and Abstraction	20
4.1. Common Type Definitions Across Modules	20
4.2. Uninterpreted Functions and Types	20
4.2.1. Uninterpreted Types	22
4.2.2. Uninterpreted Functions	22
4.3. Procedure Definition	22
4.3.1. Procedure Invocation	24

Contents

4.4. Module Instantiation and Scheduling	24
4.4.1. Accessing Instance Variables	24
4.5. Running UCLID5	24
4.5.1. Exercise: Inductive Proof of CPU model	24
A. Appendix: Uclid5 Grammar	26
A.1. Grammar of Modules and Declarations	26
A.2. Statement Grammar	28
A.3. Expression Grammar	29
A.4. Types	30
A.5. Control Block	31
A.6. Miscellaneous Nonterminals	31

List of Uclid5 Examples

1.1. A UCLID5 model that computes the Fibonacci sequence	6
2.1. Model of a simple ALU	11
3.1. UCLID5 Fibonacci model using induction in the proof script	14
3.2. UCLID5 Fibonacci model with <code>induction</code> and <code>print_cex</code>	15
3.3. Inductive proof for the Fibonacci model	17
3.4. Revisiting the Fibonacci model from Example 1.1.	18
4.1. Module <code>common</code> of the CPU model	20
4.2. Variable and Procedure Declarations of the <code>cpu</code> module	21
4.3. <code>init</code> and <code>next</code> blocks of the <code>cpu</code> module	22
4.4. Module <code>main</code> in the CPU model	23

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.

¹This 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 2. These are both of type `integer`, which corresponds to mathematical integers.²

The `init` block appears next and spans lines 4 to 7. It defines the initial values of the state variables in the module. We see that `a` is initialized to 0 while `b` is initialized to 1.

The `next` block appears after this and it defines the transition relation of the module. In the figure, the next statement spans from lines 8 to 10; `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?*

In our example, we have a single `invariant` that comprises that entire specification. Line 12 defines this `invariant`. It is named `a_le_b` and as the name suggests, it

²Mathematical 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.

1. Introduction

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 proving³ 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.

³We 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 Microsoft Windows, or a different shell should be similar.

- Unzip the archive.

```
$ unzip uclid-0.8.zip.
```

- Add the uclid binary to your path.

```
$ export PATH=$PATH:$PWD/uclid-0.8/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 the file containing the model as a command-line argument. When invoked, UCLID5 looks for a module named `main` and executes the commands in its control block.⁴ 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/` sub-directory. You can run UCLID5 on this model as:

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

This should produce the following output.

```
4 assertions passed.
0 assertions failed.
0 assertions indeterminate.
```

⁴The `--main` command line argument can be used to specify a different name for the “main” module.

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. `bool`: 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 4 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` in line 9.

The input `valid` is of type `bool`. 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 7 declares a type synonym for a `record`. It declares `result_t` as being a record 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 13.

The final point of interest, type-wise, is line 15. 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

Example 2.1 also provides examples of the most commonly used statements in UCLID5.

2. Basics: Types and Statements

```
1 module main
2 {
3   type cmd_t = enum { add, sub, mov_imm };
4   type result_t = record { valid : bool, value : bv8 };
5
6   input valid      : bool;
7   input cmd        : cmd_t;
8   input r1, r2     : bv3;
9   input immed      : bv8;
10  output result    : result_t;
11
12  var regs         : [bv3]bv8;
13  var r1val, r2val : bv8;
14  var cnt          : bv8;
15
16  init {
17    for i in range(0bv3, 7bv3) {
18      regs[i] = 1bv8;
19    }
20    cnt = 1bv8;
21    result.value = 1bv8;
22  }
23
24  next {
25    if (valid) { // Do we have a valid command?
26      r1val = regs[r1];
27      r2val = regs[r2];
28      case // Case-split on the operation to be performed.
29        (cmd == add)      : { regs[r1] = r1val + r2val; }
30        (cmd == sub)      : { regs[r1] = r1val - r2val; }
31        (cmd == mov_imm) : { regs[r1] = immed; }
32      esac
33      result.valid = true;
34      result.value = regs[r1];
35    } else {
36      result.valid = false;
37    }
38    // This code is only for testing this module.
39    cnt = cnt + cnt;
40  }
41
42  // Specification.
43  assume regindex_zero: (r1 == 0bv3 && r2 == 0bv3);
44  assume cmd_is_add: (cmd == add);
45  assume cmd_is_valid: (valid);
46  invariant result_eq_cnd: (cnt == result.value);
47
48  // Proof script.
49  control {
50    f = unroll (5);
51    check;
52    print_results;
53  }
54 }
```

Example 2.1.: Model of a simple ALU

2.2.1. For Loops

The `init` block 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.

2.2.2. If and Case Statements

Also worth pointing out are the `if` statement that appears on line 34, and the `case` statement that appears on line 38. 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++.

2.2.3. 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`.

2.3. Computation/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 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 54, 55 and 56 to constrain the input to the ALU to always be an add operation, where both operands refer to register index 0.

¹1bv8 here refers to the bit-vector value 1 of width 8.

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 57 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
6 assertions passed.
0 assertions failed.
0 assertions indeterminate.
```

UCLID5 is able to prove that the `invariant` on line 57 holds for all states reachable within 5 steps of the initial state, under the assumptions specified in lines 54-56.

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.

```
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    induction;
19    check;
20    print_results;
21  }
22 }
```

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.

3. Verification Techniques

```
$ uclid examples/tutorial/ex3.1-fib-induction.ucl
2 assertions passed.
1 assertions failed.
0 assertions indeterminate.
  FAILED -> induction (step) [Step #1] property
  a_le_b @ examples/tutorial/ex3-fib-induction.ucl, line 14
```

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.

3. Verification Techniques

```
2 assertions passed.
1 assertions failed.
0 assertions indeterminate.
  FAILED -> vobj [Step #1] property
    a_le_b @ examples/tutorial/ex3.2-fib-induction-cex.ucl,
    line 14
CEX for vobj [Step #1] property
a_le_b @ examples/tutorial/ex3.2-fib-induction-cex.ucl,
line 14
=====
Step #0
  a : -1
  b : 0
=====
=====
Step #1
  a : 0
  b : -1
=====
```

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 $a \leq b$ with $a = -1$ and $b = 0$. We execute the *next* block and now `a` gets the value of `b`, becoming 0 and `b` gets the value $a + b$, 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.

```

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  invariant a_b_ge_0: (a >= 0 && b >= 0);
16
17  // Part 3: Proof script.
18  control {
19    vobj = induction;
20    check;
21    print_results;
22    vobj->print_cex(a, b);
23  }
24 }

```

Example 3.3.: Inductive proof for the Fibonacci model

```

$ uclid examples/tutorial/ex3.3-fib-induction-proof.ucl
6 assertions passed.
0 assertions failed.
0 assertions indeterminate.

```

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 3.2 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 3.2, we are unrolling the model for 3 steps.

3.2.1. Embedded assume and assert statements

A second difference with between Example 1.1 and Example 3.2 is on lines 7 and 13.

Instead of using a module-level assumption declarations as in Example 1.1, we have an embedded assumption in the `init` block on line 7, and an embedded assertion in the

```

1 module main {
2   // System description.
3   var a, b : integer;
4
5   init {
6     // embedded assumptions.
7     assume (a <= b);
8     assume (a >= 0 && b >= 0);
9   }
10  next {
11    a, b = b, a + b;
12    // embedded assertion.
13    assert (a <= b);
14  }
15
16  // Proof script.
17  control {
18    unroll (3);
19    check;
20    print_results;
21  }
22 }

```

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

`next` block on line 13. A module-level assumption is assumed to hold by the solver at every step of execution, while an embedded assumption is assumed “instantaneously.” In particular, the assumption on line 7 tells the solver to assume that $a \leq b$ at the end of the `init` block. Notice that we are not assigning specific values to a and b , instead we are asking UCLID5 to consider all values of a and b where $a \leq b$, $a \geq 0$ and $b \geq 0$.

Similarly the assertion on line 13 is evaluated at that specific location in the code (the end of the `next` block), whereas a module-level assertion would be evaluated after the `init` block and after the `next` block is evaluated at each step.

3.2.2. Running Uclid5

Running UCLID5 on Example 3.2 shows that the embedded assertion does indeed hold for all states reachable within 3 steps of the initial state.

```

$ uclid examples/tutorial/ex3.4-fib-model-revisted.ucl
3 assertions passed.
0 assertions failed.
0 assertions indeterminate.

```

3.3. **Future Directions**

Future versions of UCLID5 will have support for the verification of properties specified in linear temporal logic (LTL), as well as 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 3-7 of module `cpu` declared in Example 4.2.

```
1 // This module declares types that are used in the rest of the model.
2 module common {
3   // addresses are uninterpreted types.
4   type addr_t = bv8;
5   type word_t = bv8;
6   // memory
7   type mem_t = [addr_t] word_t;
8   // CPU operation.
9   type op_t = enum { op_mov, op_add, op_sub, op_load, op_store };
10 }
```

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].

```

1 module cpu {
2   // Import type aliases from common.
3   type addr_t = common :: addr_t;
4   type mem_t  = common :: mem_t;
5   type word_t = common :: word_t;
6   type op_t   = common :: op_t;
7
8   // type of register file.
9   type regindex_t;
10  type regs_t = [regindex_t]word_t;
11
12  // imem is the program memory; an input to the module.
13  input imem : mem_t;
14
15  var dmem : mem_t;
16  var regs : regs_t;
17  var pc   : addr_t;
18
19  // These are uninterpreted functions.
20  function inst2op  (i : word_t) : op_t;
21  function inst2reg0 (i : word_t) : regindex_t;
22  function inst2reg1 (i : word_t) : regindex_t;
23  function inst2imm  (i : word_t) : word_t;
24  function inst2addr (i : word_t) : addr_t;
25
26  var inst : word_t;
27  var result : word_t;
28
29  procedure exec_inst(inst : word_t, pc : addr_t)
30    returns (result : word_t, pc_next : addr_t)
31    modifies regs;
32  {
33    var op      : op_t;
34    var r0ind   : regindex_t;
35    var rlind   : regindex_t;
36    var r0      : word_t;
37    var r1      : word_t;
38
39    op = inst2op(inst);
40    r0ind = inst2reg0(inst);
41    rlind = inst2reg1(inst);
42    r0 = regs[r0ind];
43    r1 = regs[rlind];
44    case
45      (op == op_mov)      : { result = inst2imm(inst); }
46      (op == op_add)      : { result = r0 + r1; }
47      (op == op_sub)      : { result = r0 - r1; }
48      (op == op_load)     : { result = dmem[inst2addr(inst)]; }
49      (op == op_store)    : { result = r0; }
50    esac
51    pc_next = pc + 1bv8;
52    regs[r0ind] = result;
53  }
54  // continued in Example 4.3.

```

Example 4.2.: Variable and Procedure Declarations of the cpu module

```

1  // continued from Example 4.2.
2
3  // Define initial state for the modules.
4  init {
5      assume (forall (r : regindex_t) :: regs[r] == 0bv8);
6      assume (forall (a : addr_t) :: dmem[a] == 0bv8);
7      pc = 0bv8;
8      inst = 0bv8;
9  }
10
11 next {
12     // get the next instruction.
13     inst = imem[pc];
14     // execute it
15     call (result , pc) = exec_inst(inst , pc);
16 }
17 }

```

Example 4.3.: `init` and `next` blocks of the `cpu` module

4.2.1. Uninterpreted Types

Example 4.2 shows the use of the *uninterpreted type*: `regindex_t` on line 9. 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 20-24 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. Procedure Definition

Besides modules, another way of managing model complexity in UCLID5 is through the use of *procedures*. Example 4.2 defines the *procedure* `exec_inst` on line 29. The procedure takes two arguments: `inst` and `pc` and returns two values: `result` and `pc_next`.

4. Compositional Modeling and Abstraction

```
1 module main {
2
3   // Import types
4   type addr_t      = common :: addr_t;
5   type mem_t       = common :: mem_t;
6   type word_t      = common :: word_t;
7   type op_t        = common :: op_t;
8   type regindex_t  = cpu :: regindex_t;
9
10  // instruction memory is the same for both CPUs.
11  var imem : mem_t;
12
13  // Create two instances of the CPU module.
14  instance cpu_i_1 : cpu(imem : (imem));
15  instance cpu_i_2 : cpu(imem : (imem));
16
17  init {
18  }
19
20  next {
21    // Invoke CPU 1 and CPU 2.
22    next (cpu_i_1);
23    next (cpu_i_2);
24  }
25
26  // These are our properties.
27  invariant eq_regs : (forall (ri : regindex_t) ::
28    cpu_i_1->regs[ri] == cpu_i_2->regs[ri]);
29  invariant eq_mem  : (forall (a : addr_t) ::
30    cpu_i_1->dmem[a] == cpu_i_2->dmem[a]);
31  invariant eq_pc   : (cpu_i_1->pc == cpu_i_2->pc);
32  invariant eq_inst : (cpu_i_1->inst == cpu_i_2->inst);
33
34  // Proof script.
35  control {
36    unroll(3);
37    check;
38    print_results;
39  }
40 }
```

Example 4.4.: Module main in the CPU model

4. Compositional Modeling and Abstraction

Note that all module-level state modified by the procedure must be declared using a `modifies` clause; this is shown on line 31.

4.3.1. Procedure Invocation

Procedures are invoked using the `call` keyword. In Example 4.3, procedure `exec_inst` is called on line 15 of Example 4.3. The actual arguments to the procedure are `inst` and `pc`, and the return values of the procedure will be stored in the variables `result` and `pc`.

4.4. Module Instantiation and Scheduling

Modules are instantiated using the `instance` keyword. Lines 14 and 15 of Example 4.4 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 4.4 invoke the next state transitions of the two instances of the `cpu` module.

4.4.1. Accessing Instance Variables

The state variables of an instantiated module are accessed using the `->` operator. The four invariants on lines 27-32 of Example 4.4, 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.

4.5. Running Uclid5

Executing UCLID5 on the complete CPU model shows that CPU is in fact deterministic.

```
$ uclid examples/tutorial/ex4.1-cpu.ucl
16 assertions passed.
0 assertions failed.
0 assertions indeterminate.
```

4.5.1. 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.

$\langle Model \rangle ::= \langle Module \rangle^*$

$\langle Module \rangle ::= \text{module } \langle Id \rangle \text{ '}' \langle Decl \rangle^* \langle ControlBlock \rangle? \text{'}'$

Declarations can be of the following types.

$\langle Decl \rangle ::= \langle TypeDecl \rangle$
| $\langle InputsDecl \rangle$
| $\langle OutputsDecl \rangle$
| $\langle VarsDecl \rangle$
| $\langle ConstsDecl \rangle$
| $\langle SharedVarsDecl \rangle$
| $\langle FuncDecl \rangle$
| $\langle ProcedureDecl \rangle$
| $\langle InstanceDecl \rangle$
| $\langle InitDecl \rangle$
| $\langle NextDecl \rangle$
| $\langle AxiomDecl \rangle$
| $\langle SpecDecl \rangle$

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

$\langle TypeDecl \rangle ::= \text{type } \langle Id \rangle \text{ '=' } \langle Type \rangle \text{ ';'}$
| $\text{type } \langle Id \rangle \text{ ';'}$

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

$\langle InputsDecl \rangle ::= \text{input } \langle IdList \rangle \text{ ':' } \langle Type \rangle \text{ ';'}$

$\langle OutputsDecl \rangle ::= \text{output } \langle IdList \rangle \text{ ':' } \langle Type \rangle \text{ ';'}$

$\langle VarsDecl \rangle ::= \text{var } \langle IdList \rangle \text{ ':' } \langle Type \rangle \text{ ';'}$

A. Appendix: UCLID5 Grammar

$\langle \text{ConstsDecl} \rangle \quad ::= \text{const} \ \langle \text{IdList} \rangle \text{' : ' } \langle \text{Type} \rangle \text{' ; '}$

$\langle \text{SharedVarsDecl} \rangle \quad ::= \text{sharedvar} \ \langle \text{IdList} \rangle \text{' : ' } \langle \text{Type} \rangle \text{' ; '}$

Function declarations refer to uninterpreted functions.

$\langle \text{FuncDecl} \rangle \quad ::= \text{function} \ \langle \text{Id} \rangle \text{' (' } \langle \text{IdTypeList} \rangle \text{') ' ' : ' } \langle \text{Type} \rangle \text{' ; '}$

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.

$\langle \text{ProcedureDecl} \rangle \quad ::= \text{procedure} \ \langle \text{Id} \rangle \text{' (' } \langle \text{IdTypeList} \rangle \text{') ' } \langle \text{ProcReturnArg} \rangle ?$
 $\quad \quad \quad \langle \text{RequireExprs} \rangle \ \langle \text{EnsureExprs} \rangle \ \langle \text{ModifiesExprs} \rangle$
 $\quad \quad \quad \text{' { ' } \langle \text{VarsDecls} \rangle^* \langle \text{Statement} \rangle^* \text{' } \text{' ; '}$

$\langle \text{ProcReturnArg} \rangle \quad ::= \text{returns} \ \text{' (' } \langle \text{IdTypeList} \rangle \text{') '}$

$\langle \text{RequireExprs} \rangle \quad ::= (\text{requires} \ \langle \text{Expr} \rangle \text{' ; ' })^*$

$\langle \text{EnsureExprs} \rangle \quad ::= (\text{ensures} \ \langle \text{Expr} \rangle \text{' ; ' })^*$

$\langle \text{ModifiesExprs} \rangle \quad ::= (\text{modifies} \ \langle \text{IdList} \rangle \text{' ; ' })^*$

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.

$\langle \text{InstanceDecl} \rangle \quad ::= \text{instance} \ \langle \text{Id} \rangle \text{' : ' } \langle \text{Id} \rangle \ \langle \text{ArgMapList} \rangle \text{' ; '}$

$\langle \text{ArgMapList} \rangle \quad ::= \text{' (' ') '}$
 $\quad \quad \quad | \ \text{' (' } \langle \text{ArgMap} \rangle \text{' , ' } \langle \text{ArgMap} \rangle \text{') '}$

$\langle \text{ArgMap} \rangle \quad ::= \langle \text{Id} \rangle \text{' : ' ' (' ') '}$
 $\quad \quad \quad | \ \langle \text{Id} \rangle \text{' : ' ' (' } \langle \text{Expr} \rangle \text{') '}$

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

$\langle \text{AxiomDecl} \rangle \quad ::= \langle \text{AxiomKW} \rangle \ \langle \text{Id} \rangle \text{' : ' } \langle \text{Expr} \rangle \text{' ; '}$
 $\quad \quad \quad | \ \langle \text{AxiomKW} \rangle \ \langle \text{Expr} \rangle \text{' ; '}$

$\langle \text{AxiomKW} \rangle \quad ::= \text{axiom} \ | \ \text{assume}$

$\langle \text{SpecDecl} \rangle \quad ::= \langle \text{PropertyKW} \rangle \ \langle \text{Id} \rangle \text{' : ' } \langle \text{Expr} \rangle \text{' ; '}$
 $\quad \quad \quad | \ \langle \text{PropertyKW} \rangle \ \langle \text{Expr} \rangle \text{' ; '}$

$\langle \text{PropertyKW} \rangle \quad ::= \text{property} \ | \ \text{invariant}$

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

$\langle \text{InitDecl} \rangle \quad ::= \text{init} \ \text{' { ' } \langle \text{Statement} \rangle^* \text{' } \text{' ; '}$

$\langle \text{NextDecl} \rangle \quad ::= \text{next} \ \text{' { ' } \langle \text{Statement} \rangle^* \text{' } \text{' ; '}$

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 $\langle LhsList \rangle$. As the name suggests, this is a list of syntactic forms that can appear on the left hand side of an assignment. $\langle Lhs \rangle$ are of four types: (i) identifiers, bitvector slices within identifiers, (iii) array indices, and (iv) fields within records.

```

<LhsList> ::= <Lhs> (',' <Lhs>)*

<Lhs> ::= <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.

$$\begin{aligned}
\langle Expr \rangle &::= \langle E1 \rangle \\
\langle E1 \rangle &::= \langle E2 \rangle \\
&| \text{'(' forall ' (' } \langle IdTypeList \rangle \text{') ' ':::' E1 ' (' } \\
&| \text{'(' exists ' (' } \langle IdTypeList \rangle \text{') ' ':::' E1 ' (' }
\end{aligned}$$

The usual logical and bitwise operators are allowed.

$$\begin{aligned}
\langle E2 \rangle &::= \langle E3 \rangle \text{'<==>'} \langle E2 \rangle | \langle E3 \rangle \\
\langle E3 \rangle &::= \langle E4 \rangle \text{'==>'} \langle E3 \rangle | \langle E4 \rangle \\
\langle E4 \rangle &::= \langle E5 \rangle \text{'&&'} \langle E4 \rangle | \langle E5 \rangle \text{'||'} \langle E4 \rangle | \\
&| \langle E5 \rangle \text{'&'} \langle E4 \rangle | \langle E5 \rangle \text{'|'} \langle E4 \rangle | \langle E5 \rangle \text{'^'} \langle E4 \rangle \\
&| \langle E5 \rangle
\end{aligned}$$

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

$$\begin{aligned}
\langle E5 \rangle &::= \langle E6 \rangle \langle RelOp \rangle \langle E6 \rangle \\
\langle RelOp \rangle &::= \text{'>'} | \text{'<'} | \text{'='} | \text{'!='} | \text{'>='} | \text{'<='} \\
\langle E6 \rangle &::= \langle E7 \rangle \text{'++'} \langle E6 \rangle \\
\langle E7 \rangle &::= \langle E8 \rangle \text{'+'} \langle E7 \rangle \\
\langle E8 \rangle &::= \langle E9 \rangle \text{'-'} \langle E9 \rangle \\
\langle E9 \rangle &::= \langle E10 \rangle \text{'*'} \langle E10 \rangle
\end{aligned}$$

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

$$\begin{aligned}
\langle E10 \rangle &::= \langle UnOp \rangle \langle E11 \rangle | \langle E11 \rangle \\
\langle UnOp \rangle &::= \text{'-'} | \text{'!'} | \text{'~'}
\end{aligned}$$

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

$$\begin{aligned}
\langle E11 \rangle &::= \langle E12 \rangle \text{'['} \langle Expr \rangle \text{'(' , ' } \langle Expr \rangle \text{')* '['} \\
&| \langle E12 \rangle \text{'['} \langle Expr \rangle \text{'(' , ' } \langle Expr \rangle \text{')* = } \langle Expr \rangle \text{' '['} \\
&| \langle E12 \rangle \text{'['} \langle Expr \rangle \text{'::' } \langle Expr \rangle \text{' '['} \\
&| \langle E12 \rangle
\end{aligned}$$

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

A. Appendix: UCLID5 Grammar

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

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

$\langle E12 \rangle$::= **false** | **true** | $\langle Number \rangle$
 | $\langle Id \rangle$ | $\langle Id \rangle$ ‘::’ $\langle Id \rangle$
 | ‘{’ $\langle Expr \rangle$ (‘,’ $\langle Expr \rangle$)* ‘}
 | **ite** ‘(’ $\langle Expr \rangle$ ‘,’ $\langle Expr \rangle$ ‘,’ $\langle Expr \rangle$ ‘)’

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.

$\langle PrimitiveType \rangle$::= **bool** | **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**.

$\langle Recordtype \rangle$::= **record** ‘{’ $\langle IdTypeList \rangle$ ‘}’

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

$\langle ArrayType \rangle$::= ‘[’ $\langle Type \rangle$ (‘,’ $\langle Type \rangle$)* ‘]’ $\langle Type \rangle$

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

$\langle SynonymType \rangle$::= $\langle Id \rangle$

$\langle ExternalType \rangle$::= $\langle Id \rangle$ ‘::’ $\langle Id \rangle$

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.

$$\begin{aligned}\langle \textit{ControlBlock} \rangle &::= \text{control} \ \{ \langle \textit{Cmd} \rangle^* \} \\ \langle \textit{Cmd} \rangle &::= (\langle \textit{Id} \rangle \text{'='})? (\langle \textit{Id} \rangle \text{'->'})? \langle \textit{Id} \rangle \\ &\quad (\text{'['} \langle \textit{IdList} \rangle \text{'}]'}?)? \langle \textit{ExprList} \rangle? \text{';'}\end{aligned}$$

A.6. Miscellaneous Nonterminals

$\langle \textit{IdList} \rangle$, $\langle \textit{IdTypeList} \rangle$ and $\langle \textit{ExprList} \rangle$ are non-empty, comma-separated list of identifiers, identifier/type tuples and expressions respectively.

$$\begin{aligned}\langle \textit{IdList} \rangle &::= \langle \textit{Id} \rangle \\ &\quad | \langle \textit{Id} \rangle \text{' ,' } \langle \textit{IdList} \rangle \\ \langle \textit{IdTypeList} \rangle &::= \langle \textit{Id} \rangle \text{' :' } \langle \textit{Type} \rangle \\ &\quad | \langle \textit{Id} \rangle \text{' :' } \langle \textit{Type} \rangle \text{' ,' } \langle \textit{IdTypeList} \rangle \\ \langle \textit{ExprList} \rangle &::= \langle \textit{Expr} \rangle \\ &\quad | \langle \textit{Expr} \rangle \text{' ,' } \langle \textit{ExprList} \rangle\end{aligned}$$