Getting Started with UCLID

January 17, 2018

1 Introduction

UCLID is a verification and synthesis focused modeling language. The UCLID toolchain aims to:

1. Enable modeling of finite and infinite state transition systems.
2. Verification of safety and k-safety properties on these systems.
3. Allow syntax-guided synthesis of models and model invariants on these transitions systems.

This document serves as introduction to UCLID modeling language and verification/synthesis toolchain.

1.1 A Simple UCLID Model

A simple UCLID module that computes the Fibonacci sequence is shown in Figure 1.1. Let us walk through each line in this model to understand the basics of UCLID.

The top-level syntactic structure in UCLID is a `module`. All modeling, verification and synthesis code in UCLID is contained within modules. In Figure 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. The following subsections will describe each of these sections of the module.

1.1.1 The System Model

This part of a UCLID 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

```
1  module main {
2    // Part 1: System description.
3    var a, b : int;
4
5    init {
6      a = 0b0; // binary literal
7      b = 0x1; // hexadecimal literal
8    }
9    next {
10     a, b = b, a + b;
11   }
12
13   // Part 2: System specification.
14   property a_le_b: a <= b;
15
16   // Part 3: Proof script.
17   control {
18     unroll (3);
19     decide;
20     print_results;
21   }
22 }
```

Figure 1: A UCLID model that computes the Fibonacci sequence.

variables: `a` and `b` on line 2. These are both of type `int`, which corresponds to mathematical integers.[1]

The `init` block appears next and spans lines 4 to 7. It defines the initial values of the states 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`.

1.1.2 The System Specification

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

In our example, we have a single `property` that comprises that entire specification. Line 12 defines this `property`. 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.

1.1.3 The Proof Script

The third and final part of the UCLID module is a set of commands to the UCLID verification engine. These tell how we should go about proving[2] that the system

[1] 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.

[2] Note we are using a very broad definition of the word prove here to refer to any systematic method that gives us assurance that the specification is (perhaps partially) satisfied.

satisfies itself 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 property: `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 property `a_le_b` is satisfied in the initial state and the 3 steps after the initial state. The `decide` command *checks* whether these proof obligations are satisfied and the `print_results` prints out the results of these checks.

1.2 Installing and Running UCLID

2 UCLID Grammar

TODO: Fix this.

Default settings

⟨*statement*⟩ ::= ⟨*ident*⟩ '=' ⟨*expr*⟩
| 'for' ⟨*ident*⟩ '=' ⟨*expr*⟩ 'to' ⟨*expr*⟩ 'do' ⟨*statement*⟩
| '{' ⟨*stat-list*⟩ '}'
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ ';' ⟨*stat-list*⟩ | ⟨*statement*⟩

Increase the two lengths

⟨*statement*⟩ ::= ⟨*ident*⟩ '=' ⟨*expr*⟩
| 'for' ⟨*ident*⟩ '=' ⟨*expr*⟩ 'to' ⟨*expr*⟩ 'do' ⟨*statement*⟩
| '{' ⟨*stat-list*⟩ '}'
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ ';' ⟨*stat-list*⟩ | ⟨*statement*⟩