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

```
module main {
  var a, b : int;

  init {
    a = 0b0; // binary literal
    b = 0x1; // hexadecimal literal
  }
  next {
    a, b = b, a + b;
  }

  property a_le_b: a <= b;

  control {
    unroll (3);
    decide;
  }
}
```

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

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 next item of interest in 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 `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`.

Default settings

$\langle statement \rangle$ ::= $\langle ident \rangle$ '`=`' $\langle expr \rangle$
 | '`for`' $\langle ident \rangle$ '`=`' $\langle expr \rangle$ '`to`' $\langle expr \rangle$ '`do`' $\langle statement \rangle$
 | '`{`' $\langle stat\text{-}list \rangle$ '`}`'
 | $\langle empty \rangle$

$\langle stat\text{-}list \rangle$ ::= $\langle statement \rangle$ '`;`' $\langle stat\text{-}list \rangle$ | $\langle statement \rangle$

Increase the two lengths

$\langle statement \rangle$ ::= $\langle ident \rangle$ '`=`' $\langle expr \rangle$
 | '`for`' $\langle ident \rangle$ '`=`' $\langle expr \rangle$ '`to`' $\langle expr \rangle$ '`do`' $\langle statement \rangle$
 | '`{`' $\langle stat\text{-}list \rangle$ '`}`'
 | $\langle empty \rangle$

$\langle stat\text{-}list \rangle$ ::= $\langle statement \rangle$ '`;`' $\langle stat\text{-}list \rangle$ | $\langle statement \rangle$

[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.