Getting Started with Uclid5

January 2018

Contents

List of Uclid5 Examples

1. Introduction

UCLID5 is a modeling language that supports verification and synthesis. The UCLID5 toolchain aims to:

1. Enable modeling of finite and infinite state transition systems.
2. Verification of safety and hypersafety (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 the UCLID5 modeling language and toolchain.

1.1. Getting Started: A Simple Uclid5 Model

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

  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 {
    unroll (3);
    check;
    print_results;
  }
}
```

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

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.

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 `int`, which corresponds to mathematical integers.[2]

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`.

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 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 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 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`

[1] This is not required by the UCLID5 syntax but is a good design practice.

[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.

[3] 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.

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.

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.zip`.

- Add the `uclid` binary to your path.

  `$ export PATH=$PATH:$PWD/uclid-0.9/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 model as a command-line argument.

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-fib-module.ucl4
```

This should produce the following output.

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

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

3. Verification Techniques

4. Compositional Verification: Procedures and Modules

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.

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

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

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.

$\langle TypeDecl \rangle$ ::= `type` $\langle Id \rangle$ '=' $\langle Type \rangle$ ';'
| `type` $\langle Id \rangle$ ';'

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

$\langle InputsDecl \rangle$ ::= `input` $\langle IdList \rangle$ ':' $\langle Type \rangle$ ';'

$\langle OutputsDecl \rangle$ ::= `output` $\langle IdList \rangle$ ':' $\langle Type \rangle$ ';'

$\langle VarsDecl \rangle$ ::= `var` $\langle IdList \rangle$ ':' $\langle Type \rangle$ ';'

$\langle$*ConstsDecl*$\rangle$ ::= `const` $\langle$*IdList*$\rangle$ '`:`' $\langle$*Type*$\rangle$ '`;`'

$\langle$*SharedVarsDecl*$\rangle$ ::= `sharedvar` $\langle$*IdList*$\rangle$ '`:`' $\langle$*Type*$\rangle$ '`;`'

Function declarations refer to uninterpreted functions.

$\langle$*FuncDecl*$\rangle$ ::= `function` $\langle$*Id*$\rangle$ '`(`' $\langle$*IdTypeList*$\rangle$ '`)`' '`:`' $\langle$*Type*$\rangle$ '`;`'

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.

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

$\langle$*ProcReturnArg*$\rangle$ ::= `returns` '`(`' $\langle$*IdTypeList*$\rangle$ '`)`'

$\langle$*RequireExprs*$\rangle$ ::= (`requires` $\langle$*Expr*$\rangle$ '`;`')*

$\langle$*EnsureExprs*$\rangle$ ::= (`ensures` $\langle$*Expr*$\rangle$ '`;`')*

$\langle$*ModifiesExprs*$\rangle$ ::= (`modifies` $\langle$*IdList*$\rangle$ '`;`')*

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

$\langle$*InstanceDecl*$\rangle$ ::= `instance` $\langle$*Id*$\rangle$ '`:`' $\langle$*Id*$\rangle$ $\langle$*ArgMapList*$\rangle$ '`;`'

$\langle$*ArgMapList*$\rangle$ ::= '`(`' '`)`'
| '`(`' $\langle$*ArgMap*$\rangle$ '`,`' $\langle$*ArgMap*$\rangle$ '`)`'

$\langle$*ArgMap*$\rangle$ ::= $\langle$*Id*$\rangle$ '`:`' '`(`' '`)`'
| $\langle$*Id*$\rangle$ '`:`' '`(`' $\langle$*Expr*$\rangle$ '`)`'

Axioms refer to assumptions while a **specification declaration** can be a `property` or `invariant` of the design. Note `property` and `invariant` are synonyms.

$\langle$*AxiomDecl*$\rangle$ ::= `axiom` $\langle$*Id*$\rangle$ '`:`' $\langle$*Expr*$\rangle$ '`;`'
| `axiom` $\langle$*Expr*$\rangle$ '`;`'

$\langle$*SpecDecl*$\rangle$ ::= $\langle$*PropertyKW*$\rangle$ $\langle$*Id*$\rangle$ '`:`' $\langle$*Expr*$\rangle$ '`;`'
| $\langle$*PropertyKW*$\rangle$ $\langle$*Expr*$\rangle$ '`;`'

$\langle$*PropertyKW*$\rangle$ ::= `property`
| `invariant`

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

$\langle$*InitDecl*$\rangle$ ::= `init` '`{`' $\langle$*Statement*$\rangle$* '`}`'

$\langle$*NextDecl*$\rangle$ ::= `next` '`{`' $\langle$*Statement*$\rangle$* '`}`'

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*⟩ ‘`[`’ ⟨*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*⟩ ‘*’ ⟨*E10*⟩

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

⟨*E10*⟩ ::= ⟨*UnOp*⟩ ⟨*E11*⟩ | ⟨*E11*⟩

⟨*UnOp*⟩ ::= ‘−’ | ‘!’ | ‘~’

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

⟨*E11*⟩ ::= ⟨*E12*⟩ ‘[’ ⟨*Expr*⟩ (‘,’ ⟨*Expr*⟩)* ‘]’
| ⟨*E12*⟩ ‘[’ ⟨*Expr*⟩ (‘,’ ⟨*Expr*⟩)* = ⟨*Expr*⟩ ‘]’
| ⟨*E12*⟩ ‘[’ ⟨*Expr*⟩ ‘:’ ⟨*Expr*⟩ ‘]’
| ⟨*E12*⟩

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

⟨*E12*⟩ ::= ⟨*E13*⟩ '`(`' ⟨*ExprList*⟩ '`)`'
| ⟨*E13*⟩ ('`.`' ⟨*Id*⟩)+
| ⟨*E13*⟩ ('`->`' ⟨*Id*⟩)+

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

⟨*E12*⟩ ::= `false` | `true` | ⟨*Number*⟩
| ⟨*Id*⟩ | ⟨*Id*⟩ '`::`' ⟨*Id*⟩
| '`{`' ⟨*Expr*⟩ ('`,`' ⟨*Expr*⟩)* '`}`'
| `ite` '`(`' ⟨*Expr*⟩ '`,`' ⟨*Expr*⟩ '`,`' ⟨*Expr*⟩ '`)`'

A.4. Types

⟨*Type*⟩ ::= ⟨*PrimitiveType*⟩
| ⟨*EnumType*⟩
| ⟨*TupleType*⟩ | ⟨*RecordType*⟩
| ⟨*ArrayType*⟩
| ⟨*SynonymType*⟩
| ⟨*ExternalType*⟩

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 length of the bit-vector.

⟨*PrimitiveType*⟩ ::= `bool` | `int` | ⟨*BitVectorType*⟩

Enumerated types are defined using the `enum` keyword.

⟨*EnumType*⟩ ::= `enum` '`{`' ⟨*IdList*⟩ '`}`'

Tuple types are declared using curly brace notation.

⟨*TupleType*⟩ ::= '`{`' ⟨*Type*⟩ ('`,`' ⟨*Type*⟩)* '`}`'

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*⟩