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

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

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 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/` subdirectory. You can run UCLID5 on this model as:

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

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

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. `int`: the type of mathematical integers.
2. `bool`: the Boolean type. This type has two values: `true` and `false`.
3. `bv`W: The family of bit-vector types parameterized by their width (W).
4. `enum`: enumerated types.
5. Tuples and records.
6. Array 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, in Example 2.1 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.

```
1  // Model of an ALU
2  module main
3  {
4    // Synonym for an enumerated type.
5    type cmd_t = enum { add, sub, mov_imm };
6    // Synonym for a record type.
7    type result_t = record { valid : bool, value : bv8 };
8
9    input valid       : bool;
10   input cmd         : cmd_t;
11   input r1, r2      : bv3;
12   input immed       : bv8;
13   output result     : result_t;
14
15   var regs          : [bv3]bv8;
16
17   // Temporary var to hold register values.
18   var r1val, r2val  : bv8;
19
20   // Variable for the test harness.
21   var cnt           : bv8;
22
23   init {
24     // All registers initialized to one.
25     for i in range(0bv3, 7bv3) {
26       regs[i] = 1bv8;
27     }
28     cnt = 1bv8;
29     result.value = 1bv8;
30   }
31
32   next {
33     // Do we have a valid command?
34     if (valid) {
35       r1val = regs[r1];
36       r2val = regs[r2];
37       // Case-split on the operation to be performed.
38       case
39         (cmd == add)      : { regs[r1] = r1val + r2val; }
40         (cmd == sub)      : { regs[r1] = r1val - r2val; }
41         (cmd == mov_imm)  : { regs[r1] = immed; }
42       esac
43       // Set the output result.
44       result.valid = true;
45       result.value = regs[r1];
46     } else {
47       result.valid = false;
48     }
49     // This code is only for testing this module.
50     cnt = cnt + cnt;
51   }
52
53   // Specification.
54   assume regindex_zero:      (r1 == 0bv3 && r2 == 0bv3);
55   assume cmd_is_add:         (cmd == add);
56   assume cmd_is_valid:       (valid);
57   invariant result_eq_cnd:  (cnt == result.value);
58
59   // Proof script.
60   control {
61     f = unroll (5);
62     check;
63     print_results;
64   }
65 }
```


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).

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`, then 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

We now describe how the computation specified by Example 2.1 is specified.

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 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.

[1] `1bv8` 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.ucl4
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

Thus far, we have only used UCLID5 for bounded model checking. UCLID5 can also be used to do inductive proofs and provides support for debugging counterexamples. This chapter will describe these features of UCLID5.

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.

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

```
$ uclid examples/tutorial/ex3.1-fib-induction.ucl4
2 assertions passed.
1 assertions failed.
0 assertions indeterminate.
  FAILED -> induction (step) [Step #1] property
  a_le_b @ examples/tutorial/ex3-fib-induction.ucl4, 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 : int;
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 3.1.1 are on lines 18 and 21. Line 18 uses a result object that stores a reference to the verification conditions generated by the `induction` command. On line 21, we use this reference and the `print_cex` command to print out the values of `a` and `b` for the counterexample generated by these verification conditions.

Running UCLID5 on Example 3.1.1 produces the following.

```
2 assertions passed.
1 assertions failed.
0 assertions indeterminate.
  FAILED -> vobj [Step #1] property
  a_le_b @ examples/tutorial/ex3.2-fib-induction-cex.ucl4,
  line 14
CEX for vobj [Step #1] property
a_le_b @ examples/tutorial/ex3.2-fib-induction-cex.ucl4,
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 3.1.1, 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 invariant, and UCLID5 will then attempt to use this fact in its proof.

3.1.2. Inductive Proof for the Fibonacci Model

Example 3.3 shows the same model as Example 3.1.1 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 : int;

  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

```
$ uclid examples/tutorial/ex3.3-fib-induction-proof.ucl4
6 assertions passed.
0 assertions failed.
0 assertions indeterminate.
```

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

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.

⟨*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*⟩ ‘`;`’

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. 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.

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

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

⟨*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*⟩* ‘`}`’

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 width 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*⟩