



2.9	State Assignments and the Transition Relation . . . . .	12
2.10	Expressions . . . . .	13
2.10.1	Truth expressions . . . . .	13
2.10.2	Term expressions . . . . .	14
2.10.3	Bit-vector expressions . . . . .	15
2.10.4	Operator Precedence . . . . .	15
2.10.5	Enumerated type expression . . . . .	16
2.10.6	Function expressions . . . . .	16
2.10.7	Function expressions returning enum type . . . . .	17
2.10.8	Predicate expressions . . . . .	17
2.10.9	Nondeterminism . . . . .	17
2.10.10	Case expressions . . . . .	17
2.11	Modules . . . . .	18
2.12	The Control Module . . . . .	19
2.12.1	External Variables . . . . .	19
2.12.2	Storage Variables . . . . .	20
2.12.3	Commands and Assignments . . . . .	20
<b>3</b>	<b>Verification with UCLID</b>	<b>22</b>
3.1	Bounded Model Checking . . . . .	22
3.2	Correspondence Checking . . . . .	22
3.3	Inductive Invariant Checking . . . . .	23
3.4	Quantifiers and Antecedent Instantiation . . . . .	23
3.5	Running UCLID . . . . .	26
3.5.1	Compile-time Messages . . . . .	26
3.5.2	Run-time Output . . . . .	26
3.5.3	Counterexamples . . . . .	28
3.5.4	Files Generated . . . . .	28
<b>4</b>	<b>Examples</b>	<b>28</b>
4.1	Common Data & Control Structures . . . . .	28
4.1.1	Memories . . . . .	28
4.1.2	Queue . . . . .	31

4.1.3	Stack . . . . .	35
4.1.4	Process Arrays . . . . .	37
4.2	Simple Pipelined Datapath . . . . .	41
4.3	Bit-vector Examples . . . . .	48
4.3.1	Bit-vector Constants . . . . .	49
4.3.2	Bit-vector Examples . . . . .	49

# 1 Introduction

The UCLID system is used to specify and verify systems that have infinite or unbounded state. The core features of the modeling language include *uninterpreted function and predicate symbols*, an *arithmetic of counters*, *bit-vector arithmetic* and restricted *lambda expressions*.<sup>1</sup> Verification methods that can be used with UCLID include *inductive invariant checking*, *correspondence checking* (such as in the style of Burch and Dill [3]), *proving simulation diagrams* showing that one machine simulates another, and *bounded model checking*. All of these verification techniques use *symbolic simulation* in one form or another.

The UCLID system comprises of the UCLID specification language and the verification engine. The UCLID language can be used to specify a state machine, where each state variable falls into one of three classes: Boolean, enumerated, or uninterpreted symbols. The UCLID verification engine comprises of a symbolic simulator that can be “configured” for different kinds of verification tasks, and a decision procedure for a logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLUF).

This document is intended to serve as an overview of UCLID, as a tutorial, and as a reference manual for the UCLID user. In Section 2, we describe the syntax and semantics of the UCLID language. In Section 3, we describe its modeling and verification capabilities. These capabilities are illustrated with examples in Section 4. For theoretical details about how the tool works, we refer the reader to our upcoming technical report.

## 1.1 Structure of the UCLID system

Figure 1 shows the structure of the UCLID system. The user writes a UCLID specification in a file with a `.ucl` suffix. The UCLID front end consists of a lexer, parser, and type checker. Depending on the commands in the input file, the symbolic simulator and the decision procedure may get called several times. The decision procedure can, in principle, use either a BDD package or a SAT solver – currently, we only provide back-end scripts for SAT solvers. For each property of interest, UCLID checks the formula corresponding to that property. It outputs either that the formula is valid, or a counterexample when the formula is invalid.

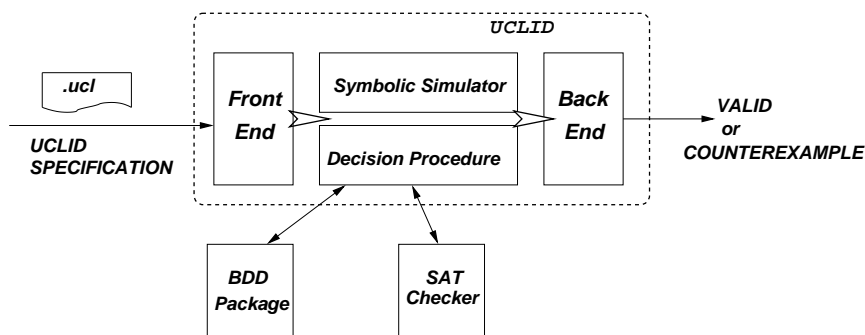


Figure 1: Structure of the UCLID system

<sup>1</sup>The name UCLID stands for Uninterpreted symbols, Counter arithmetic, and Lambda expressions for Infinite Domains.

UCLID is written in Moscow ML [8], which is a light-weight implementation of Standard ML that uses the Caml light runtime system. UCLID version 3.0 can be used with a range of SAT solvers.

## 1.2 Changes From Version 1.0

The main change in UCLID version 3.0 is the addition of bit-vector arithmetic. UCLID supports arbitrarily sized bit-vectors as well as most bit-vector operations as described later in Section 2.10. There are two bit-vector decision procedures included in this release of UCLID. They can be accessed by specifying the `-bitvec` command line option with either `eager` or `lazy`. The eager encoding option bit-blasts all bit-vector operations, while the lazy encoding option uses the technique described in [1].

## 1.3 Installation and Bug Reports

UCLID version 3.0 runs on UNIX platforms. Installation instructions may be found in the README file in the distribution. Please send bug reports to `bbrady@eecs.berkeley.edu`. When reporting a bug, please send us the specification file, the program output, and a description of the hardware and software platforms UCLID was run on, apart from any other helpful information you can provide.

# 2 The UCLID Specification Language

This section describes the syntax and semantics of the UCLID language. We present the semantics informally in the discussion accompanying the description of each syntactic construct.

The overall syntax of UCLID is very similar to the input language of the CMU version of the SMV model checker [7]. However, there are several differences, and we will point these out where necessary.

A specification in UCLID is divided into two logical sections. The first part describes the model of the system to be verified. The format of this part is very similar to that of SMV. The second part, also called the *control* section or module, specifies how the symbolic simulation is to be configured for the verification task at hand. One can view this latter portion as comprising of commands that one might ordinarily type at the cursor of an interactive tool.

## 2.1 Format

The overall format of a UCLID specification is as follows:

```
MODEL <modelname>

<typedefs>

<Global Constants>

<modules>
```

<Control module>

<modelname> denotes the name of the specification being checked. <typedefs> is an optional section containing type definitions of user-defined enumerated types. Constants with global scope are defined in the following <Global Constants> section. This is followed by the specification of one or more modules. Each module has five subsections: an INPUT section that has declarations of inputs to the module, a VAR section for declaring state variables and macro variables, a CONST section for declaring constants, a DEFINE section for defining macros, and a ASSIGN section for defining the initial state and state transition relation of the module. The last section is the <Control module> section. This includes three mandatory subsections: the EXTVAR section for declaring external variables, the STOREVAR section for declaring storage variables, and the EXEC section for listing the commands to be used in the simulation. The optional subsections are VAR, CONST and DEFINE, which serve the same function as for ordinary modules. Note that the term “MODEL” is somewhat of a misnomer, since a UCLID specification contains both the model as well as commands to run the simulation.

## 2.2 Language Overview

Before describing the language in detail, we present a simple example of a UCLID specification. Consider the example of a traffic light that changes based on the value of an internal timer, as shown in figure 2.

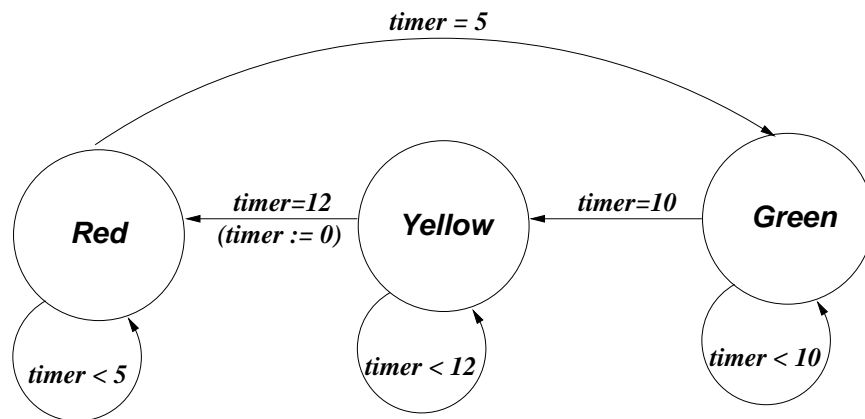


Figure 2: A timed traffic light

The UCLID specification of this system is given below:

```
MODEL timedSignal

typedef signal : enum{red, yellow, green};

CONST
ZERO : TERM;

MODULE trafficLight
```

```

INPUT

VAR
(* state variables *)
light : signal;
timer : TERM;
(* macro variables *)
FIVE : TERM;
TEN : TERM;
TWELVE : TERM;

CONST

DEFINE
FIVE := succ^5(ZERO);
TEN := succ^5(FIVE);
TWELVE := succ^2(TEN);

ASSIGN
init[light] := red;
next[light] := case
  (light = red) & (timer < FIVE) : red;
  (light = red) & (timer = FIVE) : green;
  (light = green) & (timer < TEN) : green;
  (light = green) & (timer = TEN) : yellow;
  (light = yellow) & (timer < TWELVE) : yellow;
  (light = yellow) & (timer = TWELVE) : red;
  default : light;
esac;

init[timer] := ZERO;
next[timer] := case
  (light = yellow) & (timer = TWELVE) : ZERO;
  default : succ(timer);
esac;

(*----- CONTROL MODULE -----*)
CONTROL

EXTVAR

STOREVAR
initRedCondition : TRUTH;

VAR
redCondition : TRUTH;

CONST

DEFINE
redCondition := (trafficLight.light = red) =>
  (trafficLight.timer <= trafficLight.FIVE);

```

```

EXEC
initRedCondition := redCondition;
print(trafficLight.light);
decide(initRedCondition);
simulate(1);
decide(redCondition);
simulate(1);
decide(redCondition);
simulate(1);
decide(redCondition);

```

A traffic signal is modeled as an enumerated type with three values. Constants in UCLID can be Boolean, of enumerated type, or uninterpreted symbols (we refer to such uninterpreted symbols as symbolic constants). Thus, the integer constant ZERO is modeled as a symbolic constant, and is declared globally. Within the module `trafficLight`, the VAR and CONST segments consist of variable and constant declarations respectively. Variables and constants declared here have names local to the module; however, these identifiers may be referenced anywhere outside the module by prefixing the identifier with the name of the module followed by a “.”. The DEFINE segment has the same role as in CMU SMV – it is used to define “macros” for commonly occurring shared sub-expressions. The ASSIGN segment consists of assignments of initial values to state variables and specifications of the next state functions. The case expression is used for conditional assignments, just as in SMV.

The main syntactic additions (to the SMV style) illustrated in this example include the successor function symbol(`succ`), and the CONTROL module. The condition `redCondition`, defined in the control module, checks that the timer in the red state is always less than 5. We can easily see that this condition is always true for the specified model. The storage variable `initRedCondition` is used to store the initial value of this condition. In the above example, bounded model checking has been used to check the validity of `redCondition` for 3 steps. A `print` command is used to print the initial value of the state variable `trafficLight.light`. Note that the storage variable is unnecessary in this example; the formula stored in `initRedCondition` may be decided by inserting a `decide(redCondition)` statement before the first `simulate` command.

## 2.3 Keywords and Lexical Conventions

The lexical analyzer of UCLID is case-sensitive. The following alphabetic strings are reserved keywords (some are reserved for future use).

```

MODEL CONTROL EXTVAR STOREVAR EXEC typedef enum initialize simulate
decide print printexpr FORALL MODULE INPUT VAR of CONST DEFINE ASSIGN
SPEC TERM TRUTH FUNC PRED BITVEC BITVECFUNC succ pred case esac default
init next Lambda EXISTS verify model define if then else for endfor while
do switch array vector process function module procedure include boolean
integer signal input output OUTPUT local in end assert prove

```

Names of identifiers (state variables, macro variables, constants of all types) may be any sequence of symbols in  $\{A-Z, a-z, 0-9, \_ \}$  beginning with an alphabetic character. Space, newline and tab are white

spaces and are ignored. UCLID has ML-style comments, where the comment is enclosed begins with “(“ and ends with “\*)”. Nesting of comments is allowed.

While describing syntax in the discussion that follows, we will enclose within quotes all strings recognized as tokens by the parser. Identifiers will be denoted by the strings “id”, “id0”, “id1”, etc.

## 2.4 Data Types and Type Declarations

There are six classes of data types in UCLID, as listed below:

1. `TRUTH`, the Boolean data type;
2. `TERM`, the data type for uninterpreted function symbols of arity 0;
3. `FUNC`, the data type for uninterpreted function symbols of arity greater than 0. Functions of this type take arguments of type `TERM` and return a value of type `TERM`;
4. `BITVEC`, the data type for uninterpreted bit-vector function symbols of arity 0;
5. `BITVECFUNC`, the data type for uninterpreted bit-vector function symbols of arity greater than 0. Functions of this type take arguments of type `BITVEC` and return a value of type `BITVEC`;
6. `PRED`, the data type for uninterpreted predicate symbols of arity greater than 0. Predicates of this type take arguments of type `TERM` and return a value of type `TRUTH`;
7. *Enumerated Types*, which are C-style enumerated types;
8. *Functions returning enumerated types*.

Enumerated types are the only user-defined types in UCLID. They must be declared at the very beginning of the UCLID specification using a `typedef` declaration, as given below:

```
type_decl ::= "typedef" id0 ":"  
           "enum" "{" id1 "," ... idn "}" ";"
```

An example is illustrated below:

```
typedef signal : enum{red, yellow, green};
```

The scope of `typedef` declarations is global. A `typedef` declaration is mandatory for each enumerated type. After the `typedef` declaration, the enumerated type is to be referred by the type defined in that declaration.

Variable and constant declarations are made in `INPUT`, `VAR`, or `CONST` sections.<sup>2</sup> Types have the syntax

---

<sup>2</sup>External and storage variables are also declared in a similar fashion, but we will deal with these separately in section 2.12.

```

type ::= "TRUTH" | "TERM"
      | "BITVEC" "[" integer "]"
      | "BITVECFUNC" "[" integer "]"
      | "FUNC" "[" integer "]"
      | "PRED" "[" integer "]" | id
      | "FUNC" "[" integer "]" "of" id

```

Consider the following examples. Identifiers of type `TERM` and `TRUTH` are declared in a straightforward manner as shown below:

```

foo : TRUTH;
bar : TERM;
baz : BITVEC[16]; (* bitwidth is 16 *)

```

For functions and predicates, in addition to declaring the type, the user must also declare the arity. For functions returning an enumerated type, the enumerated type is also specified. In the examples below, `f` is a function of 10 arguments, `p` is a predicate of 4 arguments, and `manySignalLights` is a function of one argument that returns the type `signal`.

```

b : BITVECFUNC[8]; (* 8 is the output size, not arity *)
f : FUNC[10];
p : PRED[4];
manySignalLights : FUNC[1] of signal;

```

Identifiers of type `FUNC` or `PRED` are useful in modeling arrays, lookup tables or memories, queues and similar data structures, using lambda expressions. Note the subtle difference between `FUNC`s and `BITVECFUNC`s. The argument to `BITVECFUNC`s is the size of the bit-vector returned by the `BITVECFUNC`s, and not the arity, as is the case with `FUNC`s and `PRED`s.

Note that a bit-vector function, a function, and a predicate of arity 0 may also be defined; however, in general, these do not always behave the same as if they were defined as `BITVEC`, `TERM`, and `TRUTH` respectively. Functions or predicates of arity 0 are best defined as having `BITVEC`, `TERM`, `TRUTH`, or an enumerated type, as necessary.

## 2.5 Constants

UCLID constants are of two kinds: *primitive* constants, and *symbolic* constants. *Primitive* constants are either of type `TRUTH` or of enumerated type. The primitive constants of type `TRUTH` are 1 and 0. Primitive constants of an enumerated type `E` are the values in the set specified in the type declaration for `E`. Primitive constants do not have to be declared in a `CONST` declaration.

All constants other than primitive constants are *symbolic*. In UCLID version 3.0, there can be no symbolic constants of enumerated type, or of a `FUNC` type that returns enumerated type. All symbolic constants must be declared in a `CONST` declaration, either globally or within a module.

The syntax of a `CONST` declaration is as follows:

```

const_decl ::= "CONST"
            id1 ":" type1 ";"
            id2 ":" type2 ";"
            ...

```

Examples of CONST declarations are given below:

```

CONST
  b0 : TRUTH;      (* symbolic Boolean constant *)
  T0 : TERM;       (* symbolic constant of type TERM *)
  f0 : FUNC[3];    (* symbolic constant of type FUNC with arity 3 *)
  bv0 : BITVEC[8]; (* symbolic constant of type BITVEC with size 8 *)
  bf0 : BITVECFUNC[16]; (* symbolic constant of type BITVECFUNC with *)
                          (* size 16 *)

```

## 2.6 Input Variables

Inputs to a module must be declared in the INPUT section of the module. Variables declared in this manner are called *input variables*.

Input variables are typically used to provide inputs to a module from the CONTROL module, where the value of the input signal in a given step may be controlled by the user. An input variable for module  $M$  might also be a variable in another module  $M'$  that  $M$  references; the declaration is needed only if  $M$  precedes  $M'$  in the file.

The syntax of an INPUT declaration is as follows:

```

input_decl ::= "INPUT"
            id1 ":" type1 ";"
            id2 ":" type2 ";"
            ...

```

UCLID version 3.0 does not support instantiation of modules within another module. We plan to implement this in the next version, and that will make different use of the INPUT section (substituting actual arguments for formal arguments).

## 2.7 State Variables

A state of a UCLID model is an assignment of values to state variables.

The state variables of each module are declared in the VAR section of that module. A state variable may be of any of the six kinds of types discussed in section 2.4. The syntax of a state variable declaration is as follows

```

var_decl ::= "VAR"
          id1 ":" type1 ";"
          id2 ":" type2 ";"
          ...

```

In addition, to state variables, auxiliary and macro variables may be used to improve readability of the specification, and in verification. These variables must also be declared in a VAR declaration. They are typically defined in the DEFINE section.

## 2.8 Macro Definitions

The DEFINE section of a module is used to define macros, especially for shared subexpressions, so as to improve readability. The syntax of a DEFINE declaration is as follows

```

defines ::= "DEFINE"
          id1 "==" expr1 ";"
          id2 "==" expr2 ";"
          ...

```

Whenever any identifier that appears on the left hand side (LHS) of a DEFINE statement appears in an expression subsequent to its definition, it is replaced by the expression on the right hand side (RHS) of its DEFINE statement. It is an error to use a DEFINE identifier before its definition; circular definitions will also result in an error.

The RHS of a DEFINE statement is an expression whose syntax is defined in section 2.10.

## 2.9 State Assignments and the Transition Relation

The initial state assignment and the transition relation for state variables within a module are defined in the ASSIGN section.

The syntax of an ASSIGN declaration is as follows

```

assigns ::= "ASSIGN"
          lval1 "==" expr1 ";"
          lval2 "==" expr2 ";"
          ...

lval ::= "init" "[" id "]" | "next" "[" id "]"

```

Notice that UCLID syntax differs from SMV syntax in that we use square brackets instead of parentheses with the `init` and `next` strings.

An l-value, denoted above by `lval`, denotes either the initial state value of a state variable `v` (written `init[v]`), or the next state value of `v` (written `next[v]`). The expression on the RHS of an `init`

assignment is evaluated prior to the simulation's run-time, and assigned to be the initial value of the state variable referenced on the LHS. For a `next` assignment, the expression is evaluated as the simulation is run, and will be the next state value of the state variable referenced on the LHS.

Expressions on the RHS of a next state assignment of a variable may reference the next state values of other state variables. It is therefore possible to have a combinational dependency amongst state variables arising from next state assignments. The UCLID interpreter extracts these dependencies automatically and evaluates the state variables in a suitable order. Circular dependencies are reported as errors; the interpreter in UCLID version 3.0 does not reproduce the dependencies in case of an error. The RHS of an initial state assignment may include other state variables, but no combinational dependencies are resolved, and if one arises, it is reported as a compile-time error. If the initial or next state of a state variable is assigned more than once, the last assignment is the only one that applies.

## 2.10 Expressions

Expressions in UCLID are generated according to the following syntax:

```

expr ::= simple-expr
      | case-expr    /* Case expression */
      | nondet-expr  /* Nondeterministic expression */

simple-expr ::= truth-expr /* Truth expression */
           | term-expr    /* Term expression */
           | bv-expr     /* Bit-vector expression */
           | enum-expr   /* Enum type expression */
           | func-expr   /* Function expression */
           | bv-func-expr /* Bit-vector Function expression */
           | enum-fexpr  /* Enum type Function expression */
           | pred-expr   /* Predicate expression */

```

Note that parentheses can always be put around expressions, except for case-expressions and nondeterministic expressions, which don't need any. Parentheses also cannot be placed around `FORALL` expressions, which are introduced in Section 2.12.3.

### 2.10.1 Truth expressions

Truth expressions or Boolean expressions, have type `TRUTH`. Their syntax is as follows:

```

truth-expr ::= "false" | "true" /* primitive Boolean constants */
           | id      /* symbolic Boolean constant or variable */
           | "next" "[" id "]"
               /* Next state value of state variable */
           | "~" truth-expr1 /* Not */
           | truth-expr1 "&" truth-expr2 /* And */

```

```

| truth-expr1 "|" truth-expr2 /* Or */
| truth-expr1 "=>" truth-expr2 /* Implication */
| truth-expr1 "<=>" truth-expr2 /* Equivalence */
| term-expr1 "=" term-expr2 /* Equality */
| term-expr1 "!=" term-expr2 /* Not-equality */
| bv-expr1 "=" bv-expr2 /* Equality */
| bv-expr1 "!=" bv-expr2 /* Not-equality */
| enum-expr1 "=" enum-expr2 /* Equality */
| enum-expr1 "!=" enum-expr2 /* Not-equality */
| term-expr1 "<" term-expr2 /* Less than */
| term-expr1 ">" term-expr2 /* Greater than */
| term-expr1 "<=" term-expr2 /* Less than or Equal */
| term-expr1 ">=" term-expr2 /* Greater than or Equal */
| bv-expr1 "<" bv-expr2 /* Signed Less than */
| bv-expr1 ">" bv-expr2 /* Signed Greater than */
| bv-expr1 "<=" bv-expr2 /* Signed Less than or Equal */
| bv-expr1 ">=" bv-expr2 /* Signed Greater than or Equal */
| bv-expr1 "<u" bv-expr2 /* Unsigned Less than */
| bv-expr1 ">u" bv-expr2 /* Unsigned Greater than */
| bv-expr1 "<=u" bv-expr2 /* Unsigned Less than or Equal */
| bv-expr1 ">=u" bv-expr2 /* Unsigned Greater than or Equal */
| pred-expr "(" term-expr1 "," term-expr2 ... ","
           term-exprn ")" /* Predicate application */
| pred-expr "(" bv-expr1 "," bv-expr2 ... ","
           bv-exprn ")" /* Predicate application */
| simple-case-expr /* defined later */

```

## 2.10.2 Term expressions

Term expressions have type TERM; they may be viewed as integers, although there are no primitive integer constants defined. Their syntax is as follows:

```

term-expr ::= id /* symbolic constant or variable */
| "next" "[" id "]"
           /* Next state value of state variable */
| "succ" "(" term-expr ")" /* Successor (+1) */
| "pred" "(" term-expr ")" /* Predecessor (-1) */
| "succ^" k "(" term-expr ")" /* +k, for constant positive
                             integer k */
| "pred^" k "(" term-expr ")" /* -k, for constant positive
                             integer k */
| func-expr "(" term-expr1 "," term-expr2 ... ","
           term-exprn ")" /* Function application */
| simple-case-expr

```

### 2.10.3 Bit-vector expressions

Bit-vector expressions have type BITVEC; Their syntax is as follows:

```
bv-expr ::= id      /* symbolic constant or variable */
| integer /* Integer constant */
| "next" "[" id "]"
           /* Next state value of state variable */
| bv-expr "&&" bv-expr /* Bit-vector bitwise AND */
| bv-expr "||" bv-expr /* Bit-vector bitwise OR */
| bv-expr "^" bv-expr /* Bit-vector bitwise XOR */
| "!!" bv-expr          /* Bit-vector bitwise NOT */
| bv-expr "+_" integer bv-expr /* Bit-vector addition */
| bv-expr "-_" integer bv-expr /* Bit-vector subtraction */
| bv-expr "*_" integer bv-expr /* Bit-vector multiplication */
| bv-expr "/_" integer integer /* Bit-vector division, unsigned */
| bv-expr "//_" integer integer /* Bit-vector division, signed */
| bv-expr "%_" integer integer /* Bit-vector mod, unsigned */
| bv-expr "%%" integer integer /* Bit-vector mod, signed */
| bv-expr ">>_" integer bv-expr /* Bit-vector logical shift right */
| bv-expr "A>>_" integer bv-expr /* Bit-vector arithmetic shift right */
| bv-expr "<<_" integer bv-expr /* Bit-vector shift left */
| bv-expr "@" bv-expr          /* Bit-vector concatenation */
| bv-expr "#" "[" integer ":" integer "]" /* Bit-vector extraction */
| bv-expr "<S" integer          /* Bit-vector sign extension */
| func-expr "(" bv-expr1 "," bv-expr2 ... ","
              bv-exprn ")" /* Function application */
| simple-case-expr
```

Important: note that each bit-vector operator has an associated width (the integer following the operator notation) and this width must be specified. Also note for bit-vector division and mod the second argument must be an integer power of 2 (i.e., division and mod by a constant power of 2).

### 2.10.4 Operator Precedence

The precedence of operators is given below, from highest to lowest precedence.

```
-                (unary)
* / // % %%
+ -
@
<S #[:]
>> A>> <<
!!
```

```

^^
&&
||
^^
=, !=, <, >, <=, >=
~
&
|
=> <=>

```

All operators associate to the left, except for =>, ~, !! and - (unary negation), which associate to the right. Note that the size portion of all bit-vector operators has been intentionally removed from the precedence list above for clarity.

### 2.10.5 Enumerated type expression

Enumerated type expressions evaluate to a user-defined enumerated type; their syntax is very similar to that of term expressions.

```

enum-expr ::= id /* primitive constant or variable */
           | "next" "[" id "]"
             /* Next state value of state variable */
           | enum-fexpr "(" term-expr1 "," term-expr2 ... ","
             term-exprn ")" /* Enum Function application */
           | simple-case-expr

```

### 2.10.6 Function expressions

Function expressions evaluate to functions that take arguments of type TERM or BITVEC and return values of type TERM or BITVEC, respectively. A powerful feature of UCLID is to be able to define functions whose body changes over steps. This allows functions to model memories, queues, lists and other useful data structures.

```

func-expr ::= id /* symbolic constant or variable */
           | "next" "[" id "]"
             /* Next state value of state variable */
           | "Lambda" "." "(" id1 "," id2 ... "," idn
             ")" term-expr
           | "Lambda" "." "(" id1 "," id2 ... "," idn
             ")" bv-expr

```

The list of arguments to the Lambda operator must have at least one element. Also, the arguments to a Lambda must be declared as symbolic constants. Both of these hold good for the Lambda operator in sections 2.10.7 and 2.10.8.

### 2.10.7 Function expressions returning enum type

Function expressions that take arguments of type TERM and return values of a user-defined enumerated type are also very useful.

```
enum-fexpr ::= id /* symbolic constant or variable */
            | "next" "[" id "]"
              /* Next state value of state variable */
            | "Lambda" "." "(" id1 "," id2 ... "," idn
                          ")" enum-expr
```

### 2.10.8 Predicate expressions

Predicate expressions evaluate to functions that take arguments of type BITVEC or TERM and return values of type TRUTH. Using the ability of UCLID to express lambda expressions, we can build, for example, predicate expressions that represent boolean state tables of arrays of processes.

```
pred-expr ::= id /* symbolic constant or variable */
            | "next" "[" id "]"
              /* Next state value of state variable */
            | "Lambda" "." "(" id1 "," id2 ... "," idn
                          ")" truth-expr
```

### 2.10.9 Nondeterminism

The UCLID syntax allows for expressions that evaluate to sets of values. Internally, fresh symbolic Boolean constants are generated to encode sets of values as an “if-then-else” expression conditioned on the values of these constants. These fresh Boolean constants have names of the form  $\_pN$  where  $N$  is a natural number, and sometimes get assigned values in a counterexample.

The syntax of nondeterministic expressions is as follows:

```
nondet-expr ::= "{" simple-expr1 "," simple-expr2 ... ","
                  simple-exprn "}"
```

### 2.10.10 Case expressions

Conditional assignments are made using case expressions. The syntax of a case expression is as follows.

```
case-expr ::= simple-case-expr
            | lambda-case-expr

simple-case-expr ::= "case"
                  truth-expr1 ":" gen-expr1 ";"
```

```

        truth-expr2 ":" gen-expr2 ";"
        ...
        default ":" gen-exprn ";"
"esac"

lambda-case-expr ::= "Lambda" "." "(" id1 "," id2 ... "," idm ")"
"case"
    truth-expr1 ":" gen-expr1 ";"
    truth-expr2 ":" gen-expr2 ";"
    ...
    default ":" gen-exprn ";"
"esac"

gen-expr ::= truth-expr
| term-expr
| bv-expr
| enum-expr
| "{" truth-expr1 "," ... "," truth-exprn "}"
| "{" term-expr1 "," ... "," term-exprn "}"
| "{" bv-expr1 "," ... "," bv-exprn "}"
| "{" enum-expr1 "," ... "," enum-exprn "}"

```

Note that we use the C-style default for the last item in the case as opposed to the SMV-style 1. Nesting of case expressions is allowed, as noted by including `case-expr` in the previous grammar definitions.

## 2.11 Modules

A module is used to collect together related state variables and associated constants, macro definitions and state assignments. UCLID version 3.0 has limited module support, and provides essentially two features. First, we allow local naming, where variables with same names can be declared in different modules. Second, we also allow the use of input signals from other modules, including the Control module. This latter feature allows the user to configure a simulation as needed. Note that UCLID version 3.0 does not allow one to instantiate modules within other modules.

The syntax of a module definition (other than the Control module) is as follows:

```

module ::= "MODULE" id
        "INPUT"
        ... /* input variable declarations */
        "VAR"
        ... /* state variable and macro declarations */
        "CONST"
        ... /* symbolic constant declarations */
        "DEFINE"

```

```

        ... /* macro definitions */
"ASSIGN"
        ... /* state variable assignments */

```

## 2.12 The Control Module

The Control module allows the user to configure the symbolic simulation for the verification task at hand. In section 3, we describe some of the verification techniques that UCLID can be used for, and how the Control module can be used for those techniques.

The syntax of the Control module is as follows:

```

control ::= "CONTROL"
          "EXTVAR"
          ... /* external variable declarations */
          "STOREVAR"
          ... /* storage variable declarations */
          "VAR"
          ... /* macro variable declarations */
          "CONST"
          ... /* symbolic constant declarations */
          "DEFINE"
          ... /* macro definitions */
          "EXEC"
          ... /* simulator commands */

```

The VAR, CONST and DEFINE segments of the Control module serve exactly the same purpose as for any other module, and have the same syntax. The VAR, CONST and DEFINE sections are optional. The VAR segment will not contain any declarations for state variables as there are no state variables in the Control module.

### 2.12.1 External Variables

In symbolic simulation, the user might sometimes wish to control the value a state variable takes at a specific step. For example, in correspondence checking using the method pioneered by Burch and Dill, one side of the commutative diagram is a simulation that first performs flushing, projection, and then executes a step of the specification machine, while the other side of the diagram first executes a step of the implementation machine, and then performs flushing. In this case, the flush signal needs to take on specific values at specific steps, and these steps are different depending upon which side of the commutative diagram we are trying to simulate.

The *external variable* is a feature of UCLID that addresses this problem. An external variable is a user-controlled input to the system that can be assigned specific values at specific steps. An external variable declaration includes, in addition to the type declaration, an assignment of the default value that the variable takes, as shown here:

```

extvar_decl ::= "EXTVAR"
            id1 ":" type1 "==" expr1 ";"
            id2 ":" type2 "==" expr2 ";"
            ...

```

External variables are also declared as inputs to modules before they are declared in the Control module (however, when they are declared as inputs, no default value is assigned). It is an error to declare an external variable that is not an input to any module.

The value of an external variable at step  $i$  is used in the simulation at step  $i + 1$ . For example, for external variable `flush`, the assignment

```
flush[3] := false;
```

means that the value of `flush` used in the fourth step of simulation is 0.

Examples of the use of external variables may be found in section 4. In particular, notice how external variables have been used in section 4.2 to do correspondence checking.

### 2.12.2 Storage Variables

During symbolic simulation, one might wish to store intermediate values of variables and expressions for later reference. Storage variables serve precisely this purpose.

The syntax of a storage variable declaration is as follows:

```

storevar_decl ::= "STOREVAR"
              id1 ":" type1 ";"
              id2 ":" type2 ";"
              ...

```

### 2.12.3 Commands and Assignments

The EXEC section of the Control module contains 4 kinds of commands and two kinds of assignments. The syntax of an EXEC section is as follows:

```

exec ::= "EXEC"
       stmt1 ";"
       stmt2 ";"
       ...

stmt ::= "simulate" "(" integer ")" /* Simulate command */
       | "initialize" /* Re-initialize all state */
       | "decide" "(" gen-truth-expr ")" /* Decide command */
       | "print" "(" id ")" /* Print the value of a state variable */

```

```

| "print" "(" `string` ")" /* Print any arbitrary string
                             enclosed in double quotes */
| "printexpr" "(" expr ")" /* Print the value of an expr */
| id ":=" expr /* Storage variable assignment */
| id "[" integer "]" ":=" expr
                             /* external variable assignment */

```

The *simulate* command takes an integer argument  $k$  that specifies the number of steps the symbolic simulation is to be run for, and simulates the system for  $k$  steps. The *initialize* command re-initializes all state variables in the system to their initial value. This is useful, for instance, while doing correspondence checking.

The *decide* command takes as argument a “generalized” truth-expression. The syntax of this generalized truth expression is given below:

```

gen-truth-expr ::= truth-expr
                | forall-truth-expr "=>" truth-expr
                | forall-truth-expr1 "=>" forall-truth-expr2

forall-truth-expr ::= "FORALL" "(" id1 "," id2 ... "," idn ")"
                    truth-expr

```

A generalized truth expression is either an ordinary truth-expression, as introduced in section 2.10.1, or an expression of the form  $A \Rightarrow C$  where the antecedent  $A$  has some variables (of type TERM) universally quantified, while the consequent  $C$  may or may not have universally quantified variables (of type TERM). The list of arguments to the FORALL operator must have at least one element. We will describe how this syntactic feature is used in section 3.4.

UCLID version 3.0 provides two commands for printing: *print* and *printexpr*. The *print* command has two variants. The first allows one to print the value of any state variable at any step. The second allows the user to print an arbitrary string enclosed in double quotation marks, primarily for pretty formatting of the output. The *printexpr* command allows one to print the value of any expression (respecting the syntax of *expr*) at the current simulation step.

The size of the output generating by printing the values of state variables and expressions produces blows up very quickly as the number of simulation steps increases; we therefore strongly discourage printing state variables and expressions after a very large number of simulation steps unless they are known to be small.

Assignments to storage variables are similar to macro definitions. The storage variable name appears on the LHS of the assignment, and it can be assigned an expression of its type. Assignments to external variables also need to specify the step of simulation the RHS expression is to be evaluated at. At that step, the expression is evaluated and the value is used wherever the external variable is used. The natural number specifying the simulation step is written in square brackets on the LHS next to the external variable name.

### 3 Verification with UCLID

UCLID version 3.0 can be used for applying several verification methods. All of these methods build on a symbolic simulator and decision procedure for the CLUF logic. We describe the more commonly used techniques in this section, along with an overview of how to run UCLID and interpret its output. Using the primitive constructs described in Section 2, the user can easily develop techniques based on symbolic simulation other than those listed below.

#### 3.1 Bounded Model Checking

Plain symbolic simulation or *bounded model checking* can be done by simply running the `simulate` command, specifying the number of steps as an argument. The `decide` command can then be used to check the validity of a property of interest in a given state. This can be a very useful tool to find bugs in a specification.

Bounded model checking can be used to check safety properties (state invariants) for a bounded number of simulation steps. If the property does not hold for any state, UCLID generates a counterexample that can be used to generate a trace showing how the bug may be exploited. However, if the property holds for all states in the simulation, we cannot make any assertions about whether it will continue to hold for future steps.

Limited checking of liveness properties is also possible. For example, if we wish to check if a process releases a lock eventually (starting from an initial state) and if the symbolic simulation leads to such a state, then, we can assert that the property does indeed hold. However, we cannot find counterexamples for such a liveness property (if it does not hold on a truncated run).

Examples of verification tasks that use symbolic simulation are given in section 2.2, 4.1.1, 4.1.2 and 4.1.3. The example in section 2.2 illustrates the use of UCLID for bounded model checking. In bounded model checking, all state variables are initialized to their initial state values using the `init` statement. To check the validity of safety properties of interest after each step of simulation, the user inserts `decide` commands after the corresponding `simulate` commands. If the formulas are valid at each step up to  $k$  steps starting from an initial state  $s_0$ , then the safety property of interest holds for the first  $k$  steps starting from  $s_0$ . We have found bounded model checking to be useful in catching bugs, especially as a first step before trying to verify the system using techniques such as correspondence checking or inductive invariant checking.

#### 3.2 Correspondence Checking

Correspondence checking involves simulating two different sides of a commutative diagram and checking the validity of the property of interest at the end [3, 9]. Thus, the outline of the verification task, as specified in the Control module of a UCLID specification, will be as follows:

1. Assign values of external variables at specific steps in the simulation, using external variable assignments.
2. Run the simulation for one side of the diagram, using the `simulate` command.
3. Save the values of relevant state variables using storage variables.

4. Re-initialize to the start state, using the `initialize` command.
5. (Re-)Assign values of external variables at different steps.
6. Run the simulation for the other side of the diagram.
7. (Optional) Save the values of relevant state variables in storage variables.
8. Construct a formula for the property of interest, and check its validity by using the `decide` command.

An example of correspondence checking, the verification of a simple pipelined datapath, is illustrated in detail in section 4.2.

### 3.3 Inductive Invariant Checking

Another verification technique that UCLID can be used on is one-step inductive invariant checking. In this technique, the starting state is initialized to a most general state. The system is symbolically simulated for one step. Then, a property of the form  $Inv \Rightarrow Next(Inv)$  is checked, where  $Inv$  denotes a formula for the invariant property we wish to verify, and  $Next(Inv)$  is its next-state version.

In general, the property  $Inv$  will need to be augmented by several other auxiliary invariants, just as is often the case in theorem proving. The user has to come up with “lemmas” to prove the inductive invariant, but the process of checking the validity of these lemmas is entirely automatic. The UCLID counterexample generator is very useful in providing hints for lemmas.

### 3.4 Quantifiers and Antecedent Instantiation

Formulas that are checked for validity using the `decide` command are formulas in the CLUF logic. This logic can express any property in quantifier-free first order logic involving counter arithmetic. It is often the case that properties of interest involve quantifiers. In particular, many properties involve the use of the universal ( $\forall$ ) quantifier. UCLID version 3.0 provides limited support for specifying properties with universal quantifiers.

There are three classes of quantified formulas that UCLID version 3.0 can handle:

1. *Universal Quantification on the outside of a quantifier-free formula:* The general form of a property of this kind is

$$\forall i_1. \forall i_2 \dots \forall i_k. P(i_1, i_2, \dots, i_k)$$

where  $P(i_1, i_2, \dots, i_k)$  is an arbitrary formula in CLUF where the  $i_j$ s have type `TERM`. Since a universally quantified formula is valid if and only if the a formula without the quantifiers is valid (i.e., a formula in which the  $i_j$ s appear free), this case can be expressed by simply dropping the quantifiers, and expressing the quantifier-free formula in UCLID syntax.

For example, consider the formula below:

$$\forall i. \forall j. (i \neq j) \Rightarrow (f(i) \neq f(j))$$

This can be expressed in UCLID syntax quite simply as

$$(i \neq j) \Rightarrow (f(i) \neq f(j))$$

We have found that most properties fall under this case.

2. *Universal Quantification only over variables appearing in the antecedent:* The general form of a formula  $p$  of this kind is

$$(\forall i_1. \forall i_2 \dots \forall i_k. A(i_1, i_2, \dots, i_k)) \Rightarrow C$$

where  $A(i_1, i_2, \dots, i_k)$  and  $C$  are arbitrary formulas in CLUF, and  $i_1, i_2, \dots, i_k$  do not appear free in  $C$ .

Notice that by pulling out the universal quantifiers,  $p$  can be rewritten as

$$\exists i_1. \exists i_2 \dots \exists i_k. (A(i_1, i_2, \dots, i_k) \Rightarrow C)$$

Formula  $p$  can be verified in UCLID in two ways. The first method involves proving a more conservative version of  $p$ , namely the formula

$$\forall i_1. \forall i_2 \dots \forall i_k. (A(i_1, i_2, \dots, i_k) \Rightarrow C)$$

Notice that the above formula is of the kind handled in item 1 above, and so can be translated to an equivalent formula in CLUF.

Often, the more conservative property fails to hold, and other techniques are needed. The second method involves the use of *instantiation*. Instantiation is the process by which the universal quantifier over the antecedent of  $p$  is converted to a finite conjunction of instances of the antecedent. Each instance is generated by assigning a symbolic constant to a quantified variable, and dropping the universal quantifier over that variable. For example, the above formula  $p$  would get translated to a new formula  $p_{inst}$  given below

$$\left( \bigwedge_{j_1, \dots, j_k=1}^{n_1, n_2, \dots, n_k} A(t_{j_1}, t_{j_2}, \dots, t_{j_k}) \right) \Rightarrow C$$

This procedure is sound, but necessarily incomplete, because it would otherwise imply the decidability of first-order logic. In other words, if  $p_{inst}$  is valid, so is  $p$ , but  $p$  could be valid without  $p_{inst}$  being valid. We have found that using an instantiation technique is often useful in proving the validity of the property of interest.

UCLID version 3.0 incorporates a simple heuristic strategy to instantiate the antecedent, which has had some success. The strategy essentially involves instantiating each quantified variables with all relevant terms from the consequent formula  $C$ . Further details of this procedure may be obtained in our upcoming technical report.

Instantiation may be specified in the UCLID language as follows. For the property  $p$  given above, the user would write a corresponding UCLID formula (of type TRUTH) as given below (assume  $k = 2$ )

$$\text{FORALL}(i1, i2) \text{ A}(i1, i2) \Rightarrow C$$

where  $A$  and  $C$  are UCLID truth-expressions corresponding to  $A$  and  $C$  above, respectively.

3. *Universal Quantification performed separately over variables appearing in the antecedent and in the consequent:* The general form of a formula  $q$  of this kind is

$$(\forall i_1. \forall i_2 \dots \forall i_k. A(i_1, i_2, \dots, i_k)) \Rightarrow (\forall j_1. \forall j_2 \dots \forall j_n. C(j_1, j_2, \dots, j_n))$$

where  $A(i_1, i_2, \dots, i_k)$  and  $C(j_1, j_2, \dots, j_n)$  are arbitrary formulas in CLUF so that  $i_1, i_2, \dots, i_k$  do not appear free in  $C(j_1, j_2, \dots, j_n)$ , and  $j_1, j_2, \dots, j_n$  do not appear free in  $A(i_1, i_2, \dots, i_k)$ .

$q$  is equivalent to the following formula

$$\forall j_1. \forall j_2 \dots \forall j_n. (\forall i_1. \forall i_2 \dots \forall i_k. A(i_1, i_2, \dots, i_k) \Rightarrow C(j_1, j_2, \dots, j_n))$$

which in turn is equivalent to

$$(\forall i_1. \forall i_2 \dots \forall i_k. A(i_1, i_2, \dots, i_k) \Rightarrow C(j_1, j_2, \dots, j_n))$$

Notice that the last formula above is in the form of item 2 above. Therefore, we can handle this formula using the conservative approach and the instantiation techniques described in item 2.

However, UCLID version 3.0 allows the user to be explicit about which variables are being universally quantified in the consequent  $C$ . Thus, for  $k = 1$  and  $n = 2$ ,  $q$  may be written in UCLID as

$$\text{FORALL}(i1) A(i1) \Rightarrow \text{FORALL}(j1, j2) C(j1, j2)$$

In UCLID version 3.0, this will have exactly the same effect as writing

$$\text{FORALL}(i1) A(i1) \Rightarrow C(j1, j2)$$

Automatic instantiation is a fairly expensive operation — the formula blows up exponentially with increase in the number of variables to be instantiated. Fortunately, automatic instantiation need not always be done. Consider the class of properties that impose constraints on the values of state variables. In these cases, the user can encode the invariant into the `init` state assignment to those variables. Such an invariant has the form  $v = P$ , where  $v$  is a state variable and  $P$  is a case expression enumerating all the possible expressions  $v$  can evaluate to along with the conditions under which  $v$  can equal them.

In the case of inductive invariant checking, if the invariant formula on a variable  $v$  is denoted by  $Inv$ , then instead of checking the validity of a formula of the form  $Inv \Rightarrow Next(Inv)$ , we merely encode  $Inv$  into the initial state of  $v$ , simulate for one step, and then check  $Next(Inv)$ .

Consider the example of section 2.2. Suppose we wanted to prove the following property using inductive invariant checking:

$$(\text{trafficLight.timer} = \text{ZERO}) \Rightarrow (\text{trafficLight.light} = \text{red})$$

We could encode the invariant into the initial state as follows:

```
init[light] := case
    timer = ZERO : red;
    default      : {red, yellow, green};
esac;
```

## 3.5 Running UCLID

The various command-line options supported by UCLID version 3.0 may be obtained by running `uclid` without any arguments. Some of the more common options will be described here. To run UCLID on a model which uses bit-vectors, you must specify the `-bitvec` option followed by either `eager` or `lazy`. To choose the SAT solver use the `-sat` option followed by the name for one of the supported SAT solvers. See the usage to get a list of SAT solver names.

### 3.5.1 Compile-time Messages

The UCLID front-end catches lexical, syntax, and type errors, and reports them by raising an ML exception with an appropriate message. The messages are fairly self-explanatory, e.g., the message displayed below (obtained from the example in section 2.2 by deleting a declaration) indicates that the user has used a storage variable without declaring it:

```
bash$ uclid timedSignal.ucl bdd 0
Compiling model timedSignal...
Compiling module trafficLight...
File "timedSignal.ucl", line 84, characters 3-5:
! s1 := trafficLight.light;
!   ^^
! Undeclared storage variable

Uncaught exception:
Fail: Error encountered
```

The trailing ML message “Uncaught exception: Fail: Error encountered” can be ignored.

### 3.5.2 Run-time Output

The output of a UCLID run includes messages printed by UCLID and the results of print statements inserted by the user in the Control module. For example, a snippet of the output generated for the UCLID specification in section 2.2 (the portion upto the first `simulate`) is given below, annotated with explanation of what the output means. This output is for the non-verbose mode.

First, the compilation messages.

```
Compiling model timedSignal...
Compiling module trafficLight...
Compilation finished for UCLID model timedSignal.
```

Then, the initial values of all state variables are always printed.

```
INITIAL STATE:

Boolean State:
[]
```

```
Term State:
[trafficLight.timer:=ZERO
,]
```

```
Func State:
[]
```

```
Pred State:
[]
```

```
Enum State:
[trafficLight.light:=red
,]
```

```
Enum Func State:
[]
```

The user inserted print of the state variable `trafficLight.light` is done next.

```
At step 0: trafficLight.light:=red
```

Then, the decision procedure prints its output for the `decide` statement.

```
*****
      Decision Procedure
*****
Stats : Original formula size :<|tnodes|,|Tnodes|> = <3,6>
```

The line above shows statistics about the sizes of the directed acyclic graph used to represent the formula. `|tnodes|` is the number of nodes corresponding to truth-expressions. `|Tnodes|` is the number of nodes corresponding to term-expressions.

```
Writing SVC formula to file <timedSignal.svc>.....
Getting function, predicate symbols.....
Getting the PCEUF from the formula.....
Eliminating uninterpreted func symbols.....
Eliminating uninterpreted pred symbols.....
Obtaining propositional form.....
Stats : Propositional formula size :<|tnodes|,|Tnodes|> = <1,0>
```

```
+++ No Counter-Examples Found : Formula Valid +++
```

Finally, a short output by the `simulate(1)` statement.

```
Simulating...
.. step 1 complete
```

UCLID internally generates fresh variables for use as arguments to Lambdas and for generating symbolic Boolean constants to perform nondeterministic choice. These may appear in the output and are identified by names starting with `_i` and `_p` respectively. The verbose mode prints a lot of statistics, and is more for debugging purposes.

### 3.5.3 Counterexamples

Counterexamples generated by UCLID include two kinds of outputs. The first portion of the counterexample is a partial interpretation of the symbolic constants in the model. Symbolic Boolean constants are interpreted over  $\{\text{true}, \text{false}\}$  and symbolic constants are interpreted over the set of integers. Symbolic constants that are not assigned any interpretation are assigned “\_”, a don’t care symbol. The second part is a counterexample trace. An example of what a counterexample looks like is given in section 4.1.4.

### 3.5.4 Files Generated

UCLID generates many intermediate files, some of which might be useful for use with other tools. For a specification file with a model named `foo` (i.e., the first line of the file reads “MODEL `foo`”), the following files are generated: (let  $f$  be the last formula checked for validity)

1. `foo.cnf`: Conjunctive Normal Form (CNF) format of the boolean formula generated from  $f$ .
2. `foo.cnf_log`: SAT solver output log.
3. `foo.cnf_result`: Satisfying assignment to  $f$  (if  $f$  is satisfiable).
4. `foo.map`: Mapping between UCLID variables and CNF variables.

There are several other temporary files generated. These files help us catch bugs in our implementation and can safely be ignored. If you’re submitting a bug report, it doesn’t hurt to send these along.

## 4 Examples

The examples described in Sections 4.1 and 4.2 can be found in the `examples/term` folder distributed with UCLID version 3.0. Bit-vector versions of the examples can be found in `examples/bv`.

### 4.1 Common Data & Control Structures

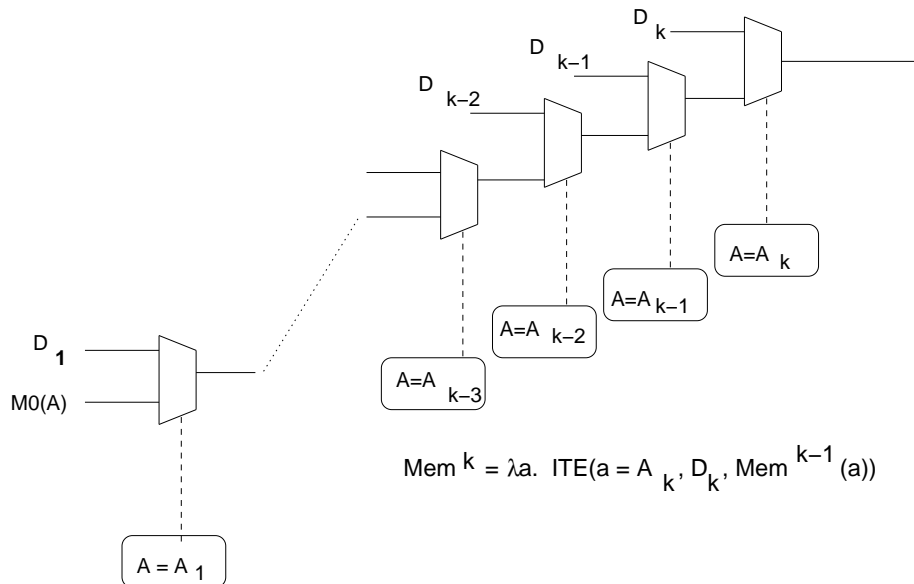
UCLID can be used to model common data and control structures such as memories, queues, stacks, and arrays of identical processes. In this section, we illustrate these with example specifications.

#### 4.1.1 Memories

The introduction of lambda notation allows us to model the effect of a sequence of read and write operations on a memory. At any point of system operation, a memory is represented by a function expression  $Mem$  denoting a mapping from addresses to values. The initial state of the memory is given by a function symbol  $M_0$ . This can be considered an uninterpreted function indicating an arbitrary memory state. The effect of a write operation with term expressions  $A$  and  $D$  denoting the address and data values yields a function expression  $Mem'$ :

$$Mem' = \lambda addr . ITE(addr = A, D, Mem(addr))$$

Figure 3 illustrates the effect of reading from a memory address  $A$  after a sequence of  $k$  writes to the memory at locations  $A_1, A_2, \dots, A_k$  with data  $D_1, D_2, \dots, D_k$  respectively.



*Effect of READ operation from the address A after a sequence of k WRITES at addresses  $A_1, A_2, \dots, A_k$  with data  $D_1, D_2, \dots, D_k$*

Figure 3: A model for memories in UCLID

The following example demonstrates the modeling of a simple memory module in UCLID. It consists of a single module named *m*. At each step of the simulation, a new address (*addr*) and a new data value (*data*) are generated. The function *Mem* is updated to reflect that *data* is written at address *addr*. The content of all other addresses other than *addr* remains unchanged.

```

MODEL memory

MODULE m

INPUT

VAR

Mem : FUNC[1];
addr : TERM;
data : TERM;

CONST

i : TERM;
m0 : FUNC[1];
d0 : TERM;
a0 : TERM;

newData : FUNC[1];

```

```

newAddr : FUNC[1];

DEFINE

ASSIGN

(* generate new data values in each step *)
init[data] := d0;
next[data] := newData(data);

(* generate new addr values in each step *)
init[addr] := a0;
next[addr] := newAddr(addr);

(* arbitrary initial state of memory *)
init[Mem] := Lambda(i). m0(i);
next[Mem] := Lambda(i).
    (* at each step, write "data" into the address "addr" *)
    case
        addr = i : data;    (* only change the content of "addr" *)
        default  : Mem(i); (* content of other addr remains unchanged *)
    esac;

```

The CONTROL module checks the property defined by the predicate *inv*. In the definition of *inv*, *Mem0* and *Mem1* denote the function *Mem* before and after one step of simulation. *Addr0* and *Data0* denote the address and the value for the write operation in the current step, respectively. The property *inv* checks that after the current step, the content of the memory at address *Addr0* is same as *Data0*. It also checks that the content of all other memory addresses remain unchanged.

```

(* control section of the model *)
CONTROL

EXTVAR

STOREVAR

Data0 : TERM;
Addr0 : TERM;
Mem0  : FUNC[1];
Mem1  : FUNC[1];

VAR

inv : PRED[1];

CONST

i : TERM;

DEFINE

```

```

(* the property to be checked *)
inv := Lambda(i).
  case
    i = Addr0 : Mem1(i) = Data0; (* the write at address Addr0 was performed correctly *)
    default   : Mem1(i) = Mem0(i); (* no other address gets written in this step *)
  esac;

EXEC

(* store the state variables before the simulation *)
Data0 := m.data;
Addr0 := m.addr;
Mem0 := m.Mem;

(* simulate for 1 step *)
simulate(1);

(* store the state variables after the simulation *)
Mem1 := m.Mem;

(* check if the property is universally valid *)
decide(inv(i));

```

### 4.1.2 Queue

A queue of arbitrary length can be modeled as a record  $Q$  having components  $\{Q.contents, Q.head, Q.tail\}$ . Conceptually, the contents of the queue are represented as some subsequence of an infinite sequence, where  $Q.contents$  is a function expression mapping an integer index  $i$  to the value of sequence element  $i$ .  $Q.head$  is a term expression indicating the index of the head of the queue, i.e., the position of the oldest element in the queue.  $Q.tail$  is a term expression indicating the index at which to insert the next element. The idea is illustrated in the figure of a queue in figure 4.

As shown in figure 4, we can represent a queue  $Q$  having an arbitrary state by letting  $Q.contents = c$ ,  $Q.head = h$ , and  $Q.tail = t$ , where  $c$  is an uninterpreted function and  $h$  and  $t$  are symbolic constants satisfying the constraint  $h \leq t$ . This constraint is enforced by including it in the antecedent of the formula whose validity we wish to check.

The operation testing if the queue is empty can be expressed quite simply as:

$$isEmpty(Q) = (Q.head = Q.tail)$$

Using this operation we can define the following three operations on the queue:

1.  $Pop(Q)$ : The pop operation on a non-empty queue returns a new queue  $Q'$  with the first element removed; this is modeled by incrementing the head (as shown in second step in figure 4).

$$Q'.head = ITE(isEmpty(Q), Q.head, \mathbf{succ}(Q.head))$$

2.  $First(Q)$ : This operation returns the element at the head of the queue, provided the queue is non-empty. It is defined as  $Q.contents(Q.head)$ .

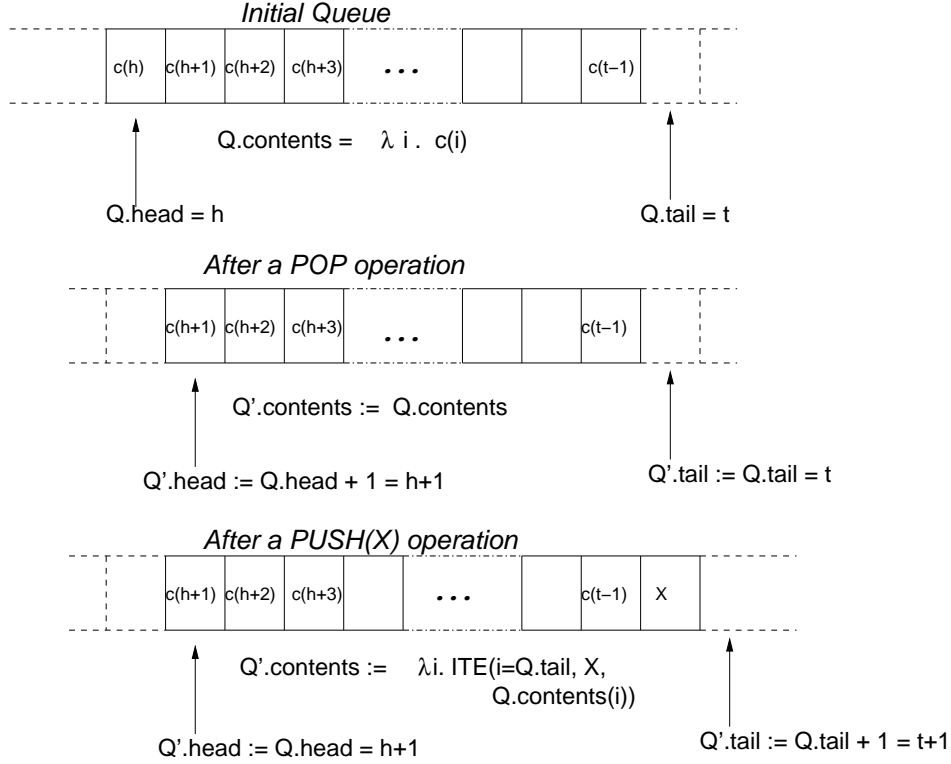


Figure 4: A model for queue in UCLID

3.  $\text{Push}(Q, X)$ : The push operation pushes data item  $X$  into  $Q$  and returns a new queue  $Q'$  where

$$\begin{aligned}
 Q'.\text{tail} &= \text{succ}(Q.\text{tail}) \\
 Q'.\text{contents} &= \lambda i. \text{ITE}(i=Q.\text{tail}, X, Q.\text{contents}(i))
 \end{aligned}$$

Assuming we start in a state where  $h \leq t$ ,  $Q.\text{head}$  will never be greater than  $Q.\text{tail}$  because of the conditions under which we increment the head.

Bounded length queues can be similarly expressed, with an additional constraint in the case of the push operation disallowing a push when the queue is full. In particular, to bound a queue to a maximum length of  $k$  (where  $k$  is an integer, not a symbolic constant), we add the condition for pushing that  $Q.\text{tail}$  is incremented only when  $Q.\text{tail} < \text{succ}^k(Q.\text{head})$ , where  $\text{succ}^k$  indicates  $k$  compositions of the successor operation.

A UCLID description of a queue is given below. It is similar to the description of the queue described above. The operations on the queue, namely PUSH, POP, and NOOP are defined as being of an *enumerated* type *optype*. This example illustrates the usage of *external* variables. Both *op* (the current operation) and *data* (data to be pushed) are external variables to the module  $Q$  (declared under INPUT in the module definition). The state variables of module  $Q$  are the contents, head and tail, and there is a macro variable for the first element of the queue.

The CONTROL module simulates the model for 6 steps, starting from an empty queue (initial value of

both *head* and *tail* is *h0*). The first 3 steps pushes *d0*, *d1*, *d2* into the queue, and the next 3 steps pops from the queue. Since *op* is an external variable to the module *Q*, we can define the operation at each step of simulation. In the model given below, the operations are PUSH for the first 3 steps (*op[0] = PUSH*, *op[1] = PUSH*, *op[2] = PUSH*) and POP for the next 3 steps (*op[3] = POP*, *op[4] = POP*, *op[5] = POP*). Similarly, the data to be pushed in step 1,2 and 3 are *d0*, *d1* and *d2* respectively. The final property checks if the elements pushed into the queue are the same as those popped off the queue, and if they are popped out in the same order that they were pushed in.

```

MODEL queue

typedef optype : enum{POP, PUSH, NOOP};

CONST

d0 : TERM;
d1 : TERM;
d2 : TERM;

MODULE Q

INPUT

op : optype;
data : TERM;

VAR

contents : FUNC[1];
head : TERM;
tail : TERM;
first : TERM;

CONST

i : TERM;
c0 : FUNC[1];
h0 : TERM;
t0 : TERM;

DEFINE

first := contents(head);

ASSIGN

init[head] := h0;
next[head] := case
    op = POP : succ(head); (* increment the head for POP operation *)
    default : head;
esac;

init[tail] := h0; (* same as the head *)

```

```

next[tail] := case
  op = PUSH : succ(tail); (* increment tail for a PUSH operation *)
  default   : tail;
esac;

(* contents maps an index to the content at the index *)
init[contents] := Lambda(i). c0(i);
next[contents] := Lambda(i).
  case
    (op = PUSH) & (i = tail) : data; (* insert data at the tail for PUSH*)
    default                   : contents(i);
  esac;

CONTROL

EXTVAR

op : optype := NOOP;
data : TERM := d0;

STOREVAR

item0 : TERM;
item1 : TERM;
item2 : TERM;

EXEC

(* the sequence of operations *)
op[0] := PUSH;
op[1] := PUSH;
op[2] := PUSH;
op[3] := POP;
op[4] := POP;
op[5] := POP;

(* the sequence of data items to be pushed in steps 0,1,2 *)
data[0] := d0;
data[1] := d1;
data[2] := d2;

(* simulate *)

simulate(3);
item0 := Q.first;
simulate(1);
item1 := Q.first;
simulate(1);
item2 := Q.first;
simulate(1);

```

```
(* check the order of insertion is same as the order of retrieving *)
decide((item0 = d0) & (item1 = d1) & (item2 = d2));
```

### 4.1.3 Stack

The modeling of stacks is done in a way very similar to the way queues are handled. A stack  $S$  is modeled as a record  $\{S.contents, S.top, S.bottom\}$ .  $S.contents$  serves the same purpose as in the example for the queue, mapping an index  $i$  to the content of the stack at index  $i$ .  $S.top$  denotes the top of the stack, and is the index of the latest element inserted into the stack.  $S.bottom$  remains constant and serves only for the purpose of checking if the stack is empty ( $S.top = S.bottom$ ).

We check the property that the sequence of operations, PUSH followed by a POP returns the same stack (i.e. the contents of the stack between  $S.top$  and  $S.bottom$  remain the same).

```
MODEL stack
```

```
(* type of operations of a Stack *)
typedef optype : enum{POP, PUSH, NOOP};
```

```
CONST
```

```
d0 : TERM;
```

```
MODULE S
```

```
INPUT
```

```
op : optype;
data : TERM;
```

```
VAR
```

```
contents : FUNC[1];
top : TERM;
bottom : TERM;
first : TERM;
```

```
CONST
```

```
i : TERM;
c0 : FUNC[1];
t0 : TERM;
b0 : TERM;
```

```
DEFINE
```

```
first := contents(top);
```

```
ASSIGN
```

```

(* top is the index of the element at the top of the stack *)
init[top] := t0;
next[top] := case
  op = POP : pred(top); (* decrement top for POP operation *)
  op = PUSH : succ(top); (* increment top for PUSH operation *)
  default : top;
esac;

(* the index of the bottom of the stack *)
init[bottom] := b0;
next[bottom] := bottom;

(* contents maps an index to it's content *)
init[contents] := Lambda(i). c0(i); (* start with arbitrary content of each index *)
next[contents] := Lambda(i).
case
  (op = PUSH) & (i = succ(top)) : data; (* insert data at the top of stack *)
  default : contents(i);
esac;

CONTROL

EXTVAR

op : optype := NOOP;
data : TERM := d0;

STOREVAR

item : TERM;
C0 : FUNC[1];
C1 : FUNC[1];

CONST

i : TERM;

EXEC

(* sequence of operation, a PUSH followed by POP *)
op[0] := PUSH;
op[1] := POP;

data[0] := d0;

C0 := S.contents;
simulate(1);
item := S.first;
simulate(1);
C1 := S.contents;

decide(item = d0);

(* check if the stack contents remain the same after

```

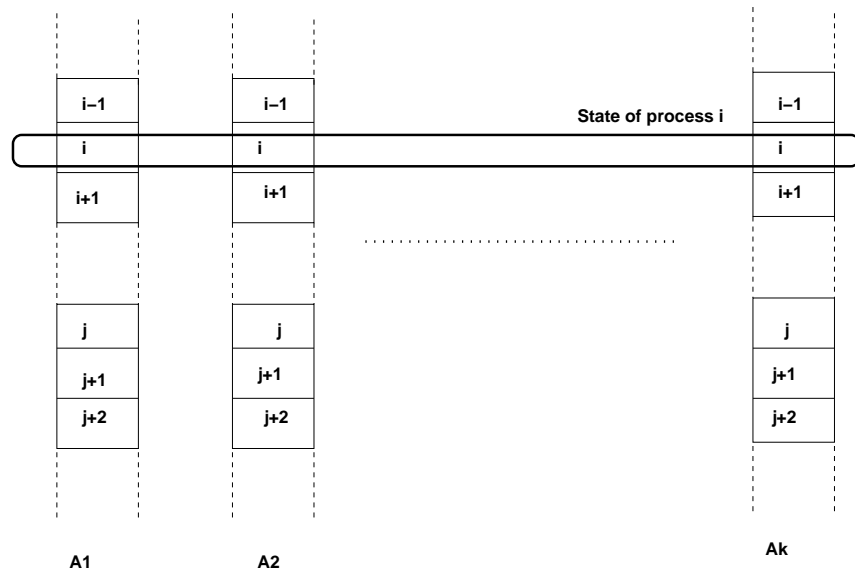
```

* a PUSH followed by a POP
*)
decide( ((i <= S.top) & (i >= S.bottom)) => (C0(i) = C1(i)) );

```

#### 4.1.4 Process Arrays

Lambda expressions can be used to represent systems containing an arbitrary number of identical processes. For each integer state variable of the process state, we use a function expression  $S$ , where  $S(i)$  denotes the value of this state variable for process  $i$ . Similarly, we represent a Boolean state variable as a predicate expression. Figure 5 illustrates this concept.

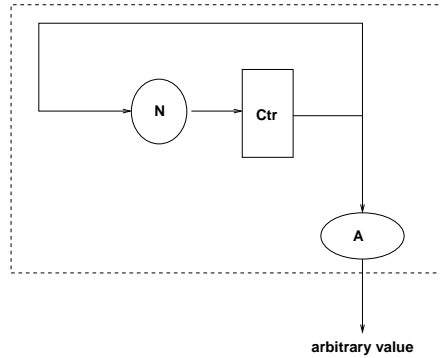


If  $\{a_1, \dots, a_k\}$  are the state variables for process  $i$ , then the set of identical processes can be represented by a set of lambda-expressions  $\{A_1, \dots, A_k\}$ , ( $A_j$  is a function expression if  $a_j$  is a term variable, is a predicate expression if  $a_j$  is a boolean variable). The state of the process  $i$  is given by  $\{A_1(i), \dots, A_k(i)\}$

Figure 5: Modeling an array of identical processes in UCLID

We include a very simple example involving arbitrary number of identical processes and a central scheduler process. The state of each process is defined by the function *state*, which maps a process (identified by its index) to its state (which can be either in critical section (*CS*) or non-critical section (*NONCS*)). Initially, a process with index  $i$  can either be in the *CS* or *NONCS*, depending on the value of the predicate  $X(i)$ . The process *scheduler* generates an arbitrary process index (*scheduler.pid*). At each step, the process identified by *scheduler.pid* is chosen to execute and it changes its state to one of *CS* or *NONCS* completely non-deterministically (given by the expression  $\{CS, NONCS\}$  in the next-state expression for *state*).

An interesting feature included in this model is the arbitrary value generator. The arbitrary value generator is used to generate an arbitrary process id on each step. It is modeled in the module *scheduler* with the help of *ctr*, *newCount* and *Arbval*. Figure 6 illustrates the modeling of random value generator.



**Ctr** is a term, **N** and **A** are uninterpreted functions of order 1.  
 The sequence  $\{A(\text{Ctr}), A(N(\text{Ctr})), A(N(N(\text{Ctr}))) \dots\}$  is the  
 output of the generator of random values, one value at each step of the simulation

Figure 6: Modeling a random value generator in UCLID

The predicate *inv* in CONTROL module defines the property that two different processes  $i, j$  cannot be in the state *CS* simultaneously. We want to check that *inv* holds for all pair of processes  $(i, j)$  after one step of simulation provided it holds for the initial step. The predicates *inv0* and *inv1* denote the property *inv* for the initial step and the next step respectively.

As we can clearly see, the property *inv* is not preserved by the system. This is because any process can change its state to *CS* regardless of the presence of other processes in the critical section. This example was included to demonstrate the modeling of identical process array and also to demonstrate the counter-examples generated when a property does not hold. The trace for the counter-example is included after the model.

```

MODEL process_array

(* a process can either be in critical section or non-critical section *)
typedef process_state : enum{CS, NONCS};

MODULE p_array

INPUT

scheduler.pid : TERM;

VAR

state : FUNC[1] of process_state;
pid : TERM;

CONST

i : TERM;
i0 : TERM;
  
```

```

X : PRED[1];

DEFINE

pid := scheduler.pid;

ASSIGN

(* Initial state of a process i is CS if X(i) is true *)
init[state] := Lambda(i).
case
  X(i)      : CS;
  default   : NONCS;
esac;

(* The process #pid changes its state non-deterministically
 * to either CS or NONCS
 * All other processes remain in the same state
 *)
next[state] := Lambda(i).
case
  i = pid : {CS, NONCS}; (* non-deterministic assignment *)
  default : state(i);
esac;

MODULE scheduler

INPUT

VAR

pid : TERM;
ctr : TERM;

CONST

Arbval : FUNC[1];
newCount : FUNC[1];

c0 : TERM;

DEFINE

ASSIGN

(* ctr and Arbval together generate arbitrary pid values *)
init[ctr] := c0;
next[ctr] := newCount(ctr);

(* generate arbitrary pid for each step *)
init[pid] := Arbval(c0);

```

```

next[pid] := Arbval(ctr);

CONTROL

EXTVAR

STOREVAR

inv0 : TRUTH;
inv1 : TRUTH;

VAR

inv : PRED[2];

CONST

i : TERM;
j : TERM;

DEFINE

(* check the property of mutual exclusion *)
inv := Lambda(i,j). (i != j & p_array.state(i) = CS) =>
    (p_array.state(j) = NONCS);

EXEC

inv0 := inv(i,j) & inv(j,i);
simulate(1);
inv1 := inv(i,j);

(* final property to be checked *)
decide(inv0 => inv1);

```

The counter-example trace for the property is shown below. There are two parts of each counter-example. Part 1 defines a *partial* interpretation to all the function and predicate symbols present in the formula to be checked. For example the predicate  $X$  evaluates to **true** for the process number 9, and evaluates to **false** for the process 11. Hence the initial state of process 9 is *CS*, and that of 11 is *NONCS*. A "–" stands for a *don't care* value.

The second part of the counter-example provides a *trace* of the counter-example. It shows some of the values of the state variables (namely *scheduler.pid*, *scheduler.ctr* and *p\_array.state*) at each step. Since we simulated for only 1 step, the counter-example prints the *trace* for the initial state (state 0) and the next state (state 1). In general, if we simulate for  $k$  steps, the *trace* contains the states  $0, 1, \dots, k$ . As we can see, the process chosen in the initial step was 11 (denoted by *scheduler.pid*). This process changes its state to *CS*

as shown in the *trace* for state 1. But another process, process 9 had its initial state as *CS*. Hence we have two different processes in the critical section, which is a counter-example. Variables like *p0* are internal variables used to non-deterministically chose an expression from a set of non-deterministic assignments. In this case *p0* denotes the choice between { *CS*, *NONCS*} for the next-state assignment for *p\_array.state*. The user can usually ignore the variables with prefix *p*, while interpreting the counter-example.

```

+++ Counter-Examples Found : Formula Not Valid +++

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
COUNTER-EXAMPLE GENERATED
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  Part 1 : Interpretations to func and predicates
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

i=11
j=9
p_array.X(11)=false
p_array.X(9)=true
scheduler.c0=_
scheduler.Arbval(_)=11
_p0=true

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  Part 2 : Counter Example Trace
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+-----+
|          State 0::          |
+-----+
scheduler.pid:=11
scheduler.ctr:=_
p_array.state::p_array.state[11] :=NONCS;p_array.state[9] :=CS
+-----+
|          State 1::          |
+-----+
scheduler.pid:=11
scheduler.ctr:=_
p_array.state::p_array.state[11] :=CS;p_array.state[9] :=CS

```

## 4.2 Simple Pipelined Datapath

In this section, we illustrate the use of UCLID to perform correspondence checking by checking a simple pipelined datapath against a non-pipelined specification version.

Consider the simple 3-stage pipelined arithmetic unit shown in figure 7. We wish to show that it has behavior equivalent to the unpipelined reference implementation displayed in figure 8. This example originated with

the first work on symbolic model checking [2], and has subsequently become a standard for verification research [4, 5, 6]. In our version, we make the verification task somewhat more challenging by using both stalling and forwarding to resolve read-after-write hazards in the pipeline. Previous versions used only forwarding, with the result that a new result is written to the register file on each step of operation.

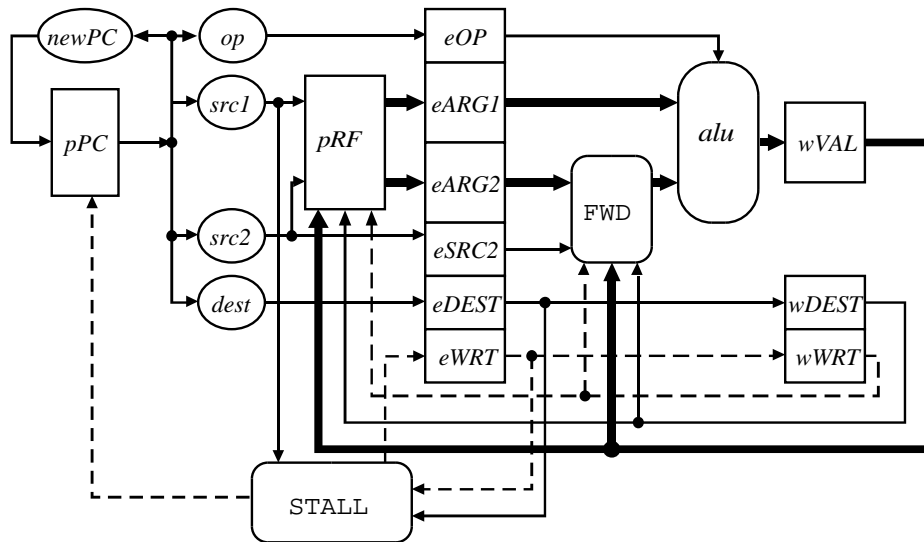


Figure 7: **Pipelined Version of ALU Circuit.** Read-after-write hazards are resolved for the first operand by stalling and for the second by forwarding.

Let us first consider the specification or reference model given in figure 8. It has two state variables: a term variable  $sPC$  serving as a program counter for fetching successive operations, and a function variable  $sRF$  indicating the contents of the register file. The behavior is defined in terms of 6 function symbolic constants:  $newpc$  indicating how to compute the next value of the program counter, decoding functions  $src1$ ,  $src2$ ,  $dest$ , and  $op$  defining the operands and ALU operation, and  $alu$  defining the functionality of the ALU.

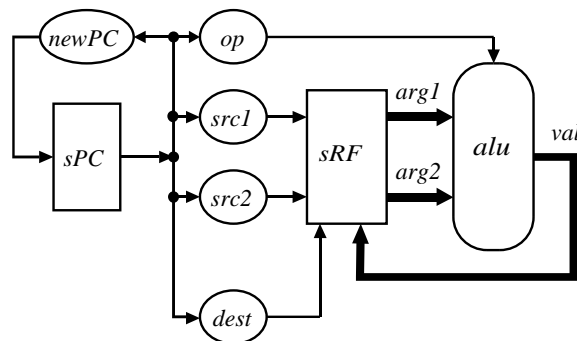


Figure 8: **Reference Version of ALU Circuit.** The state consists of the program counter  $sPC$  and the register file  $sRF$ .

Now consider the pipelined implementation illustrated in Figure 7. This version splits the computation into 3 stages: the fetch stage fetches the instruction and reads the operands from the register file. The execute

stage uses forwarding to get the proper first argument and computes the ALU result. Stalling takes place whenever the instruction in the execute stage has the same destination as the first source operand in the fetch stage. The effect of stalling is to disable the updating of the PC and to mark the instruction in the fetch stage as invalid. On the next step the instruction will be refetched and will read the correct result from the register file. The write-back stage writes the instruction result back to the register file. The pipeline has its own version of the program counter  $pPC$  and the register file  $pRF$ . It also contains a set of pipeline registers: those between the fetch and execute stages have names beginning with ‘ $e$ ’, while those between the execute and write-back stages have names beginning with ‘ $w$ .’

To verify that the pipelined implementation is equivalent to the unpipelined specification, we need to prove a correspondence between the implementation and the specification, as depicted in figure 9. In this figure,  $Q_{pipe}$  is an arbitrary start state of the pipelined implementation. After one step, the state of the pipelined implementation changes to  $Q'_{pipe}$ .  $Q_{spec}$  is the start state of the specification machine that corresponds to  $Q_{pipe}$ . It is obtained by first flushing all instructions in the pipeline, and then projecting the state of the pipeline to match the corresponding state of the specification. The goal of correspondence checking is to check if the state  $Q'_{spec}$ , obtained after 0 or 1 steps from  $Q_{spec}$ , corresponds to  $Q'_{pipe}$ .

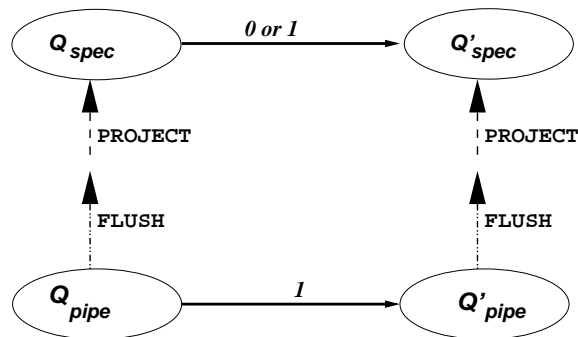


Figure 9: **Correspondence Diagram.**  $Q_{pipe}$  and  $Q'_{pipe}$  are states of the pipelined implementation.  $Q_{spec}$  and  $Q'_{spec}$  are states of the specification.

A natural way to model the specification and the implementation in UCLID is to create one module for each, and form the synchronous composition of the two. In order to check the correspondence of the implementation and specification modules, we will use three external variables: one to control flushing of the pipeline, another to run the specification only when necessary, and a third to project the state of the implementation onto the specification. Using these external variables, appropriate storage variables, and statements in the Control module, we can perform correspondence checking.

The code below is annotated with comments describing the modeling and verification in detail.

```
MODEL PipelinedDatapath

(* First we define symbolic constants that will be used in
   both implementation and specification modules *)
CONST

pc0 : TERM;          (* The initial value of the pc *)
```

```

rf0 : FUNC[1];      (* The initial value of the register file *)

newpc : FUNC[1];   (* The functional constant for generating
                    the next pc *)
src1 : FUNC[1];    (* Function to extract the first source *)
src2 : FUNC[1];    (* Function to extract the second source *)
op : FUNC[1];      (* Function to extract the op code *)
dest : FUNC[1];    (* Function to extract the destination *)
alu : FUNC[3];     (* ALU function *)

a : TERM;          (* Used as an argument to Lambdas *)

(***** IMPLEMENTATION MODULE *****)
MODULE impl

INPUT

    flush : TRUTH; (* This is an external variable. It is true
                    in a step when the pipeline is to be flushed
                    in that step *)

VAR

    (* ---- State variables ---- *)
    pPC : TERM;    (* Program counter *)
    pRF : FUNC[1]; (* Register file - modeled as a memory *)
    eOP : TERM;    (* Exec stage state: Operand *)
    eSRC2 : TERM;  (* Exec stage state: First source Reg ID *)
    eARG1 : TERM;  (* Exec stage state: First Operand *)
    eARG2 : TERM;  (* Exec stage state: Second Operand *)
    eDEST : TERM;  (* Exec stage state: Destination Reg ID *)
    eWRT : TRUTH;  (* Exec stage state: Valid bit *)
    wVAL : TERM;   (* Writeback stage state: Result from ALU *)
    wDEST : TERM;  (* Writeback stage state: Destination Reg *)
    wWRT : TRUTH;  (* Writeback stage state: Valid bit *)

    (* --- Macro variables --- *)
    stall : TRUTH; (* stall signal *)
    fwd : TERM;    (* Forwarded result *)

    (* Local constants -- used for giving arbitrary start state
       assignments to pipeline stage registers *)
CONST

    op0 : TERM;
    s0 : TERM;
    a1 : TERM;
    a2 : TERM;
    d0 : TERM;
    d1 : TERM;
    x0 : TERM;

    w0 : TRUTH;

```

```

w1 : TRUTH;

(* Macro definitions *)
DEFINE
  (* Stall when the destination of a valid instruction in execute
     stage is same as the source of the fetched instruction *)
  stall := eWRT & (src1(pPC) = eDEST);

  (* Forward the result of the writeback stage if it is valid, and
     its destination is the same as the 2nd source of the exec stage *)
  fwd := case
    wWRT & (wDEST = eSRC2) : wVAL;
    default : eARG2;
  esac;

(* State assignments *)
ASSIGN

  init[pPC] := pc0;          (* initially arbitrary *)
  next[pPC] := case         (* Remains the same only if the pipeline *)
    (flush | stall) : pPC;  (* is flushed, or if there is a stall *)
    default : newpc(pPC);
  esac;

  init[pRF] := rf0;         (* initially arbitrary *)
  next[pRF] := Lambda(a) .
    case
      wWRT & (a = wDEST) : wVAL; (* stores wVAL in wDEST if valid *)
      default : pRF(a);
    esac;

  init[eOP] := op0;
  next[eOP] := op(pPC);

  init[eSRC2] := s0;
  next[eSRC2] := src2(pPC);

  (* Note that we use "next[pRF]" below instead of "pRF". This avoids
     the need to perform forwarding or stalling when the write back
     destination matches one of the source operands in the fetch stage *)
  init[eARG1] := a1;
  next[eARG1] := next[pRF](src1(pPC));

  init[eARG2] := a2;
  next[eARG2] := next[pRF](src2(pPC));

  init[eDEST] := d0;
  next[eDEST] := dest(pPC);

  init[eWRT] := w0;
  next[eWRT] := ~stall & ~flush; (* valid when there is neither a
                                   stall nor a flush *)

  init[wVAL] := x0;

```

```

next[wVAL] := alu(eOP,eARG1,fwd); (* The first argument is read from
                                   the RF, but the second is
                                   forwarded *)

init[wDEST] := d1;
next[wDEST] := eDEST;

init[wWRT] := w1;
next[wWRT] := eWRT;

(***** SPECIFICATION MODULE *****)
MODULE spec

INPUT

isa : TRUTH;          (* isa is 1 if the specification can make a
                       step, 0 otherwise *)
project_impl : TRUTH; (* project_impl is 1 if we must project the
                       implementation's state onto the
                       specification's state, 0 otherwise *)

VAR

(* State variables *)
sPC : TERM;          (* The program counter *)
sRF : FUNC[1];      (* The register file *)

(* Macro Variables *)
arg1 : TERM;        (* First operand *)
arg2 : TERM;        (* Second operand *)
val : TERM;         (* ALU output value *)

CONST

pc1 : TERM;

(* Macro definitions *)
DEFINE

arg1 := sRF(src1(sPC));
arg2 := sRF(src2(sPC));
val := alu(op(sPC),arg1,arg2);

(* State variable assignments *)
ASSIGN

init[sPC] := pc0; (* initially arbitrary - but same as impl. *)
next[sPC] :=
  case
    project_impl : impl.pPC; (* Copy the implementation PC *)
    isa : newpc(sPC);        (* if a step can be made, generate
                               next value of PC *)

```

```

        default : sPC;
    esac;

    init[sRF] := rf0; (* initially arbitrary - but same as impl. *)
    next[sRF] := Lambda ( a ) .
        case
            project_impl : impl.pRF(a); (* Copy the implementation RF *)
                (isa & (a = dest(sPC))) : val; (* if a step can be made,
                                                store val at dest(sPC) *)
            default : sRF(a);
        esac;

(***** CONTROL Module *****)
CONTROL

(* external variable declarations *)
EXTVAR

(* These variables have already appeared as INPUTs. They are
   all initialized with a default value of 0. *)
flush : TRUTH := 0;
isa : TRUTH := 0;
project_impl : TRUTH := 0;

(* Storage variables. These will be used to store intermediate values
   during the simulation *)
STOREVAR

I_pc : TERM;      (* Impl PC after 1 step *)
I_rf : FUNC[1];  (* Impl RF after 1 step *)

S_pc0 : TERM;    (* Spec PC after 0 steps *)
S_rf0 : FUNC[1]; (* Spec RF after 0 steps *)
S_pc1 : TERM;    (* Spec PC after 1 step *)
S_rf1 : FUNC[1]; (* Spec RF after 1 step *)

CONST

al : TERM;      (* Argument to a Lambda *)

(* Statements begin *)
EXEC

(*-----*)
(* First, we simulate counter-clockwise from the left hand bottom to
   the right hand top of the diagram *)

(* One step of the implementation *)
simulate(1);

(* We need to flush for 2 steps. So set flush to 1 for the next
   2 steps *)
flush[1] := 1;

```

```

flush[2] := 1;

(* Flush the pipeline *)
simulate(2);

(* Store the state of the pipeline after flushing *)
I_pc := impl.pPC;
I_rf := impl.pRF;

(*-----*)
(* Reset all state variables to the initial state *)
initialize;

(*-----*)
(* Now, we simulate clockwise from the left hand bottom to the right
   hand top of the diagram *)

(* First flush for two steps *)
flush[0] := 1;
flush[1] := 1;
simulate(2);

(* Now project the impl. state onto the spec. *)
project_impl[2] := 1;
simulate(1);
(* Store the spec state variable values after 0 steps *)
S_pc0 := spec.sPC;
S_rf0 := spec.sRF;

(* Run the spec machine for 1 step *)
isa[3] := 1;
simulate(1);
(* Store the spec state variable values after 1 step *)
S_pc1 := spec.sPC;
S_rf1 := spec.sRF;

(*-----*)
(****** check for correspondence ******)
(* Does the PC and RF correspond, for either 0 or 1 steps of the spec.? *)
decide((S_pc1 = I_pc) & (S_rf1(a1) = I_rf(a1)) |
       (S_pc0 = I_pc) & (S_rf0(a1) = I_rf(a1)));

```

### 4.3 Bit-vector Examples

The main difference between modeling with TERMS and BITVECS is, not surprisingly, the use of bit-vector arithmetic. See the directory `examples/bv/arith` for UCLID examples which make use of bit-vector arithmetic. The same examples can be found in the following two sections.

### 4.3.1 Bit-vector Constants

In order to encode bit-vector constants, one has to build them from binary constants. One way to do this is as follows:

```
...
DEFINE
zero_bit := ( 0 # [0:0] );
one_bit := ( 1 # [0:0] );
ucl_hex_0 := ( zero_bit @ zero_bit @ zero_bit @ zero_bit );
ucl_hex_1 := ( zero_bit @ zero_bit @ zero_bit @ one_bit );
ucl_hex_2 := ( zero_bit @ zero_bit @ one_bit @ zero_bit );
ucl_hex_3 := ( zero_bit @ zero_bit @ one_bit @ one_bit );
ucl_hex_4 := ( zero_bit @ one_bit @ zero_bit @ zero_bit );
ucl_hex_5 := ( zero_bit @ one_bit @ zero_bit @ one_bit );
ucl_hex_6 := ( zero_bit @ one_bit @ one_bit @ zero_bit );
ucl_hex_7 := ( zero_bit @ one_bit @ one_bit @ one_bit );
ucl_hex_8 := ( one_bit @ zero_bit @ zero_bit @ zero_bit );
ucl_hex_9 := ( one_bit @ zero_bit @ zero_bit @ one_bit );
ucl_hex_a := ( one_bit @ zero_bit @ one_bit @ zero_bit );
ucl_hex_b := ( one_bit @ zero_bit @ one_bit @ one_bit );
ucl_hex_c := ( one_bit @ one_bit @ zero_bit @ zero_bit );
ucl_hex_d := ( one_bit @ one_bit @ zero_bit @ one_bit );
ucl_hex_e := ( one_bit @ one_bit @ one_bit @ zero_bit );
ucl_hex_f := ( one_bit @ one_bit @ one_bit @ one_bit );
...
```

Note in the above code that to create `zero_bit` and `one_bit` we extract the zeroth bit. If this isn't done, `zero_bit` and `one_bit` will be larger than a single bit due to the internal representation of integer constants in UCLID. We recommend that whenever you use a bit-vector constant that you explicitly size it using an extraction or sign-extension of the appropriate length.

### 4.3.2 Bit-vector Examples

The first example proves that adding a number to itself is equal to multiplication by 2.

```
MODEL add_mult_test
CONST (*globals*)
MODULE m
INPUT

VAR

CONST
a : BITVEC[8];

DEFINE
zero_bit := ( 0 # [0:0] );
one_bit := ( 1 # [0:0] );
ucl_hex_2 := ( zero_bit @ zero_bit @ one_bit @ zero_bit );
b := ( a +_8 a );
c := ( a *_8 ucl_hex_2 );
```

```

root := (b = c);

ASSIGN
CONTROL
EXTVAR
STOREVAR
VAR
CONST
DEFINE

EXEC
decide(m.root);

```

The next two examples are similar in nature to the first. For brevity, only the DEFINE section is shown. The rest of the examples are exactly the same as the first example in this section.

This model proves that adding a number to itself and shifting a number left by one are equivalent.

```

...
DEFINE
zero_bit := ( 0 # [0:0] );
one_bit := ( 1 # [0:0] );
ucl_lhex_1 := ( ucl_hex_1 # [0:0] );
b := (a +_8 a);
c := (a <<_8 ucl_lhex_1);
root := (b = c);
...

```

The second shows that multiplication by 2 and shifting left by 1 are equivalent.

```

...
DEFINE
zero_bit := ( 0 # [0:0] );
one_bit := ( 1 # [0:0] );
ucl_hex_1 := ( zero_bit @ zero_bit @ zero_bit @ one_bit );
b := (a <<_8 ucl_hex_1);
c := (a *_8 (ucl_hex_1 +_8 ucl_hex_1) );
root := (b = c);
...

```

## References

- [1] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.
- [2] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *28th Design Automation Conference*, 1991.

- [3] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.
- [4] D. Cyrluk and P. Narendran. Ground temporal logic: a logic for hardware verification. In D. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 247–259. Springer-Verlag, June 1994.
- [5] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV '98)*, LNCS 1427, pages 244–255. Springer-Verlag, June 1998.
- [6] A. J. Isles, R. Hojati, and R. K. Brayton. Computing reachable control states of systems modeled with uninterpreted functions and infinite memory. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV '98)*, LNCS 1427. Springer-Verlag, June 1998.
- [7] Kenneth McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1992.
- [8] Moscow ML. Available at <http://www.dina.dk/~sestoft/mosml.html>.
- [9] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions. In *Correct Hardware Design and Verification Methods (CHARME '99)*, September 1999.