Institut für Computersysteme
ETH Zürich

N. Wirth, 15.4.94 / 5.11.96

Lola: An Object–Oriented Logic Description Language

Lola is a notation (language) for specifying synchronous, digital circuits (logic). In many ways it resembles a procedural programming language. However, Lola texts describe static circuits rather than dynamic processes. The objects occurring in a description are variables representing signals, operators representing gates, and composite structures of such objects.

1. Identifiers, Integers, Logical Values, and Comments

Identifiers are used to denote constants, variables, and types.

```
identifier =    letter {letter | digit} ["'"].
integer =       digit {digit}.
LogicValue =    "'0" | "'1".
```

Comments are sequences of characters enclosed by the brackets (* and *), and they may occur between any two symbols within a Lola text.

2. Basic and Simple Types

Every variable in Lola has a type. It is either a basic, predefined type or a structured type defined in the program text. The basic types are denoted by BIT, TS, or OC, which differ only in their rules governing assignment. (TS denotes a tri–state bus, OC an open–collector bus). Variables of a basic type have a logic value, denoted by '0 or '1.

```
SimpleType =    BasicType | identifier ["(" ExpressionList ")"].
BasicType =     "BIT" | "TS" | "OC".
ExpressionList = expression {"," expression}.
```

3. Array types

Array types consist of an array of elements, all of which have the same type. A numeric expression indicates the number of elements in the array. Elements are identified by an index value. Indices range from 0 to the array's length minus 1.

```
type =    { "[" expression "]" } SimpleType.
```

4. Constant Declarations

Constant declarations serve to introduce identifiers denoting a constant, numeric value.

```
ConstDeclaration =  identifier ":=" expression ";".
```

5. Variable Declarations

Variable declarations serve to introduce identifiers denoting a logic variable and to associate it with a type. All variables declared in an identifier list have the same type.

```
VarDeclaration =    IdList ":" type ";".
IdList =            identifier {"," identifier}.
```

6. Expressions

Expressions serve to combine variables with logical operators to define new values. The operators are negation, logical conjunction (and), disjunction (or), and difference (xor). Operands are of any basic type. Elements of an array are selected by an index: *a.5*, *a.i.* If the index is an expression, the form a[exp] is used.

+	logical disjunction (or)
–	logical difference (exclusive or)
*	logical conjunction (and)
~	negation (not)

A *multiplexer* is denoted by MUX(s: a, b) and is equal to ~s*a + s*b.

A *register* provides the means to specify a value depending on previous values in time (sequential circuit). The value of REG(ck: en, d) in the *next* clock cycle is equal to d in the current clock cycle, if en = '1. If en = '0, the previous value is retained. en is called the register's *enable* signal. In the short form REG(en, d), the clock signal does not appear explicitly and is assumed to be a *global clock*, as is typical for synchronous circuits. The short notation REG(d) stands for REG('1, d).

A *latch*, denoted by LATCH(g, d), is a storage element which holds a logic value while g = '0. If g = '1, the value d is acquired (and the latch is transparent).

A basic *SR–latch* with (active–low) set and reset signals is expressed as SR(s', r').

Apart from logic expressions there exist numeric expressions. They follow the same rules of composition; their operators are those of addition, subtraction, multiplication, division, and exponentiation and (integer) logarithm to the base 2.

```
selector =    {"." identifier | "." integer | "[" expression [ ".." expression] "]"}.
factor =      identifier selector | LogicValue | integer | "(" expression ")" |
              "MUX" "(" expression ":" expression "," expression ")" |
              "REG" "(" [expression ":"][expression ","] expression ")" |
              "LATCH" "(" expression "," expression ")" |
              "SR" "(" expression "," expression ")" |
              "~" factor | "EXP" factor | "LOG" factor.
term =        factor {("*" | "/" | "DIV" | "MOD") factor}.
expression =  term {("+" | "–") term}.
```

Note that the selector parameter(s) of MUX is followed by a colon instead of a comma.

A selector of the form [m .. n], applied to an array variable *a*, denotes the subarray *a[m], a[m+1], ... , a[n].* It can be used as actual parameter in unit assignments (see 7.2.8) corresponding to a formal input variable.

LOG n (n >= 0) is the binary logarithm, rounded down to the next integer. LOG 0 = 0. EXP x (x >= 0) is equal to 2↑x.

7. Assignments

Assignments serve to define a variable's value, which is specified as that of an expression. The form v := x stands for "let v be equal to x". Hence, an assignment must be understood as a variable's *definition* (in contrast to an identifier's declaration). v and x do not have the same roles, and this asymmetry is emphasized by the use of the symbol := instead of the symmetric equal sign (=).

If a variable is of type BIT, the expression must be of any basic type, and only a single assignment (definition) is allowed.

If the variable's type is TS, the statement must specify a condition (representing a tri–state gate). Arbitrarily many assignment to the same variable are permitted. However, the value of the bus is defined only if some condition's value is '1.

```
assignment =      identifier selector ":=" [condition "|"] expression.
condition =       expression.
```

If the variable's type is OC (open–collector), arbitrarily many assignments to the same bus variable are permitted. The bus value is '1, unless any one of the assigned expressions has the value '0 (wired–or).

8. Control Statements

Statements are either assignments or composites of assignments, namely repeated or conditional assignments.

```
relation =            expression ("=" | "#" | "<" | "<=" | ">" | ">=") expression.
IfStatement =         "IF" relation "THEN" StatementSequence
                      {"ELSIF" relation "THEN" StatementSequence}
                      ["ELSE" StatementSequence]
                      "END".
constructor =         "[" expression {"," expression} "]".
parameter =           expression | constructor.
ForStatement =        "FOR" identifier ":=" expression ".." expression "DO" StatementSequence "END".
UnitAssignment =      identifier selector "(" parameter {"," parameter} ")".
statement =           [assignment | UnitAssignment | PosAssignment | IfStatement | ForStatement].
StatementSequence =   statement {";" statement}.
```

The expressions in a for–statement must be numeric, and they specify the range of integer values for which the constituent statement sequence is defined. The identifier associated with the control variable is considered as being local to the for–statement, i.e. does not exist in the for–statement's context. The control variable typically serves as index to array variables.

A constructor denotes a list of signals. It can be used as an actual parameter corresponding to a formal array variable with the same number of elements.

9. Modules

A module specifies variables and a circuit involving these variables. A module may also contain definitions of composite types. Modules are the textual units for compilation.

```
InType =      {"[" expression "]"} "BIT".
InOutType =   {"[" expression "]"} ("TS" | "OC").
OutType =     {"[" expression "]"} ("BIT" | "TS" | "OC").
ImportList =  "IMPORT" identifier {"," identifier} ";".
module =      "MODULE" identifier ";" [ImportList]
              {TypeDeclaration ";"}
              ["CONST" {ConstDeclaration}]
              ["IN" {IdList ":" InType ";"}]
              ["INOUT" {IdList ":" InOutType ";"}]
              ["OUT" {IdList ":" OutType ";"}]
              ["VAR" {VarDeclaration}]
              ["CLOCK" expression ";"]
              ["BEGIN" StatementSequence]
              "END" identifier ".".
```

Note that declarations introduce identifiers for variables, and statements define their values. The identifier at the end of the module's declaration must match the one following the symbol MODULE.

The clock declaration serves to specify the value of the global clock, which is taken as default in a factor of the form REG(en, d) (see Sect. 6).

Example: The following circuit represents an 8–bit binary adder with inputs x (x.0 ... x.7), y (y.0 ... y.7), and the carry *ci.* Its outputs are the sum s (s.0 ... s.7) and the carry *co.*

```
MODULE Adder;
    CONST N := 8;
    IN x, y: [N] BIT; ci: BIT;
    OUT s: [N] BIT; co: BIT;
    VAR c: [N] BIT;
BEGIN
    s.0 := x.0 – y.0 – ci; c.0 := (x.0 * y.0) + (x.0 – y.0)*ci;
    FOR i := 1 .. N–1 DO
        s.i := x.i – y.i – c[i–1]; c.i := (x.i – y.i) + (x.i – y.i) * c[i–1]
    END ;
    co := c[N–1]
END Adder.
```

10. Composite Types and Unit Assignments

In addition to basic types and array types, composite types can be declared. This facility may be compared to record types in programming languages, and variables (instances) of such types correspond to components of circuits, i.e. to objects being part of a circuit. A type declaration specifies a composite type, of which instances are introduced by variable declarations. The heading of a type declaration contains up to four sections:

1. The section headed by the symbol IN declares input signals to which no assignments within the type declaration are permitted. The identifiers act as formal names for expressions specified externally in unit assignments, where the expressions appear in the form of parameters. The types of the formal names must be BIT or arrays thereof. The corresponding actual expressions must be of any basic type, or an array thereof.

2. The section headed by the symbol INOUT declares signals to which assignments within the type declaration are permitted. As in the case of inputs, the identifiers act as formal names for signals declared outside the type declaration. Their types must be TS or OC or arrays thereof.

3. The section headed by the symbol OUT declares actual variables. Their type must be BIT, TS, OC, or an array thereof. These output variables are accessible in the scope (type declaration) in which the composite variable is declared. There they are denoted by the composite variable's identifier followed by the output identifier as selector (the latter acting like a field identifier of a record). No assignments are permitted outside the declaration in which the output is declared.

4. The section headed by the symbol VAR declares actual variables. They are not accessible outside the type declaration.

Summary

mode	allowed types	types of corresponding actual parameters
IN	BIT	BIT, TS, OC
INOUT	TS, OC	TS, OC
OUT	BIT, TS, OC	
VAR	BIT, TS, OC, declared type	

Consider the following example:

```
TYPE AddElem;
    IN x, y, ci: BIT;
    OUT z, co: BIT;
    VAR h: BIT;
BEGIN h := x – y; z := h – ci; co := (x * y) + (h * ci)
END AddElem
```

A variable u of type *AddElem* (i.e. an instance of an *AddElem*) is introduced by the declaration:

```
u: AddElem
```

The inputs appear in the form of parameters (expressions) in a statement called *unit assignment:*

```
u(a, b, c)
```

The components of u are obtained by substitution of the actual expressions for the corresponding formal identifiers:

```
u.h := a – b;
u.z := u.h – c;
u.co := (a * b) + (u.h * c)
```

An 8–bit adder with inputs X and Y can now be declared as consisting of 8 identical elements

```
U: [8] AddElem
```

defined by the following assignments:

```
U.0(X.0, Y.0, '0);
FOR i := 1 .. 7 DO U.i(X.i, Y.i, U[i–1].co) END
```

and the sum is represented by the variables U.0.z ... U.7.z .

```
TypeDeclaration =   "TYPE" identifier ["*"] ["(" IdList ")"] ";"
                    ["CONST" {ConstDeclaration}]
                    ["IN" {IdList ":" InType ";"}]
                    ["INOUT" {IdList ":" InOutType ";"}]
                    ["OUT" {IdList ":" OutType ";"}]
                    ["VAR" {VarDeclaration}]
                    ["BEGIN" StatementSequence]
                    "END" identifier.
```

The number of bracket pairs in a formal type specifies the number of indices used for this parameter. If an expression is specified, it indicates the length of the corresponding actual arrays given in unit assignments. The identifier at the end of the declaration must match the one following the symbol TYPE.

11. Generic Types

Declared types can be supplied with parameters and therefore define an entire class of types. The parameters are numeric quantities and are used, for example, to specify the length of arrays. Example:

```
TYPE Counter(N);
   IN ci: BIT;
   OUT co: BIT; q: [N] BIT;
   VAR c: [N] BIT;
BEGIN q.0 := REG(q.0 – ci); c.0 := q.0 * ci;
   FOR i := 1 .. N–1 DO q.i := REG(q.i – c[i–1]); ci := q.0 * c[i–1] END ;
   co := c[N–1]
END Counter
```

An instance u of a counter with 8 elements is declared as

```
u: Counter(8)
```

yielding the variables

```
u.co, u.q.0. ... , u.q.7  and  u.c.0, .. , u.c.7
```

Note that u.c is local, i.e. not accessible outside the type declaration. A corresponding unit assignment with enable signal e is now expressed by

```
u(e)
```

12. Placement Information

Variables may be attributed with a list of integer values. The language does not specify their semantics. However, their typical use is as coordinates of the positions of parts generating the respective signals in physical devices. The values are assigned to a variable by a position assignment.

```
position =          expression {"," expression} | "[" position {";" position} "]" .
PosAssignment =     identifier selector "::" position.
```

If the attributed variable is an array, a list of positions can be specified, whose elements are attributed to the corresponding elements of the array.

Examples for variables *x, y: BIT; z: [4] BIT:*

```
x :: 2;          y :: 3, 4, 5;          z :: [6,7; 8,9; 10,11]
```

If variables local to a type declaration are attributed, the specified values are added to the corresponding attributes of every instance of the type. This implies that the attributed coordinates of the local variables are relative to the origin of the instance.

13. Lola Syntax

```
identifier  =        letter {letter | digit} [""].
integer  =           digit {digit}.
LogicValue =         "'0" | "'1".

BasicType =          "BIT" | "TS" | "OC".
SimpleType =         BasicType | identifier ["(" ExpressionList ")"].
ExpressionList =     expression {"," expression}.

type =               { "[" expression "]" } SimpleType.
ConstDeclaration =     identifier ":=" expression ";".
VarDeclaration =     IdList ":" type ";".
IdList =             identifier {"," identifier}.

selector =           {"." identifier | "." integer | "[" expression [ ".." expression] "]"}.
factor =             identifier selector | LogicValue | integer | "(" expression ")" |
                     "MUX" "(" expression ":" expression "," expression ")" |
                     "REG" "(" [expression ":"][expression ","] expression ")" |
                     "LATCH" "(" expression "," expression ")" |
                     "SR" "(" expression "," expression ")" |
                     "~" factor | "EXP" factor | "LOG" factor.
term =               factor {("*" | "/" | "DIV" | "MOD") factor}.
expression =         term {("+" | "-") term}.

condition =          expression.
assignment =         identifier selector ":=" [condition "|"] expression.
relation =           expression ("=" | "#" | "<" | "<=" | ">" | ">=") expression.
IfStatement =        "IF" relation "THEN" StatementSequence
                     {"ELSIF" relation "THEN" StatementSequence}
                     ["ELSE" StatementSequence]
                     "END" .
ForStatement =       "FOR" identifier ":=" expression ".." expression "DO" StatementSequence "END" .
constructor =        "[" expression {"," expression} "]".
parameter =          expression | constructor.
UnitAssignment =     identifier selector "(" parameter {"," parameter} ")".
position =           expression {"," expression} | "[" position {";" position} "]" .
PosAssignment =      identifier selector "::" position.
statement =          [assignment | UnitAssignment | PosAssignment | IfStatement | ForStatement].
StatementSequence =    statement {";" statement}.

InType =           {"[" expression "]"} "BIT".
InOutType =        {"[" expression "]"} ("TS" | "OC").
OutType =          {"[" expression "]"} ("BIT" | "TS" | "OC").
ImportList  =      "IMPORT" identifier {"," identifier} ";" .
module  =          "MODULE" identifier ";" [ImportList]
                   {TypeDeclaration ";"}
                   ["CONST" {ConstDeclaration}]
                   ["IN" {IdList ":" InType ";"}]
                   ["INOUT" {IdList ":" InOutType ";"}]
                   ["OUT" {IdList ":" OutType ";"}]
                   ["VAR" {VarDeclaration}]
                   ["CLOCK" expression ";"]
                   ["BEGIN" StatementSequence]
                   "END" identifier "." .

TypeDeclaration =    "TYPE" identifier ["*"] ["(" IdList ")"] ";"
                   ["CONST" {ConstDeclaration}]
                   ["IN" {IdList ":" InType ";"}]
                   ["INOUT" {IdList ":" InOutType ";"}]
                   ["OUT" {IdList ":" OutType ";"}]
                   ["VAR" {VarDeclaration}]
```

["BEGIN" [StatementSequence]
"END" identifier.

14. Lola–Compiler Error Numbers

0 undefined identifier
1 multiple definition of identifier
2 identifier too long
3 field identifier not visible
4 identifier mismatch
5 field identifier undefined
6 exponent base not 2
7 number too large
8 non–terminated comment
9 illegal constant

10 identifier expected
11 MODULE expected
12 . must be followed by identifier or number
14 (expected
15) expected
16] expected
17 bad factor
18 relation expected
19 , expected
20 : expected
21 := or (expected
22 THEN expected
23 .. expected
24 ; expected
25 . expected
26 END expected
27 DO expected
28 BIT, TS, OC, or identifier expected
29 = expected

32 indexed variable is not an array
34 . is not preceded by a record or an array variable
35 too few actual parameters
36 too many actual parameters
37 record type expected
38 expression is not a constant
39 integer expression expected
40 parameter type mismatch
42 index is not an integer
43 index out of range
44 incompatible types
46 illegal assignment, y not of type BIT
47 illegal assignment to input
48 illegal bus assignment
49 illegal TS–assignment to a non–bus

15. User Guide

A Lola module is compiled by the command

Lola.Compile *	(text in the marked viewer)
Lola.Compile @	(text starting at most recent selection)

After a successful compilation, the generated code is "executed", resulting in the *Data Structure* representing the circuit. This is followed by a simplification step and a search for cycles in combinational parts that would produce race conditions. The data structure can be reconverted into formulas shown by the command *Lola.Show.*

Simplification is based on the following axioms

~(~x) = x	LATCH('1, x) = x
x * '0 = '0	x * '1 = x
x + '0 = x	x + '1 = '1
x – '0 = x	x – '1 = ~x

Further commands are

LSB.Init	removes the Data Structure generated and collects free heap space
LSB.OpenLog	reopens the log viewer
LSB.ClearLog	delete text in log viewer

16. Implementation Notes

Knowledge about the Data Structure generated by the Lola System is necessary when building other tools, such as comparators with other structures (like layouts) or automatic circuit generators.

The following interface components of module LSB (Lola System Base) are relevant for the subsequent presentation:

```
CONST NameLen = 32;

(*values of fct*)  bit = 0; ts = 1; oc = 2; integer = 3; array = 4; record = 5; sect = 6;
      buf = 7; not = 8; and = 9; or = 10; xor = 11; mux = 12; mux1 = 13;
      reg = 14; reg1 = 15; latch = 16; sr = 17; tsg = 18; link = 19;

(*values of class*)  var = 0; in = 1; out = 2; io = 3;

TYPE Name = ARRAY NameLen OF CHAR;
    Signal = POINTER TO SignalDesc;
    Variable = POINTER TO VarDesc;

    SignalDesc = RECORD
            x, y: Signal;
            fct, val, u, v: SHORTINT
        END ;

    VarDesc = RECORD (SignalDesc)
            name: Name;
            class: SHORTINT;
            next, dsc: Variable
        END ;

VAR org, zero, one, clk: Variable;
    Log: Texts.Text;

PROCEDURE WriteName(VAR W: Texts.Writer; v: Variable);
PROCEDURE This(org: Variable; VAR name: ARRAY OF CHAR): Variable;
PROCEDURE New(f: SHORTINT; x, y: Signal): Signal;
```

```
PROCEDURE NewVar(f, val: SHORTINT; x, y: Signal; next: Variable; VAR name: ARRAY OF CHAR): Variable;
PROCEDURE Simplify(x: Variable);
PROCEDURE Loops(x: Variable);
PROCEDURE Show(x: Variable; lev: INTEGER);
PROCEDURE Assign(v: Variable);
```

The parameter x of procedures *Simplify, Loops, Show* must not be NIL.

The structure is a binary tree of *Variables* rooted in the global variable *org.* The field *name* indicates a variable's identifier, and *fct* indicates its type. If it is *Array* or *Record,* the field *dsc* denotes the list of the variable's components linked by the field *next.* Note that *org* designates a pseudo–variable standing for the record containing all global variables of the module. Its name is that of the module.

The "value" of a variable is represented by an expression tree rooted in field x. This tree consists of elements of type *Signal.* Each node denotes an operator specified by field *fct* and operands given by fields x and y. A node with *fct = not* uses y as its operand, and the operators MUX and REG, having 3 operands, are represented by a pair of nodes

x + y	(or, x, y)
~x	(not, NIL, x)
MUX(sel: x, y)	(mux, sel, (mux1, x, y))
REG(ck: en, d)	(reg, ck, (reg1, en, d))

The "value" x of a variable of type TS, to which expressions e0, e1, e2, ... are assigned, is a list of nodes with *fct = link*, linked by the y–field. The x–fields denote the expression trees for e0, e1, e2, ... The same holds for variables of type OC.

bus := e0 | x0	
bus := e1 | x1	bus.x = (ts, (link, (tsg, e0, x0), (link, (tsg, e1, x1), NIL)))
ocb := x0	
ocb := x1	ocb.x = (oc, (link, x0, (link, x1, NIL)))

The field y of *Variables* is used as a back–pointer designating their immediate ancestor. This field is used by procedure *WriteName* for composing a variable's full name. The field *val* is available for use by other programs, and *u* and *v* may contain position information in layouts.