The Language Lola, FPGAs, and PLDs in Teaching Digital Circuit Design

N. Wirth

Institut fur Computersysteme
ETH Zurich

Abstract. A language for describing digital circuits is presented, which is both simple and powerful enough for use in both practice and education. A system is described which transforms program texts into fusemaps for PLDs or into configurations of FPGAs. These devices are used in laboratory exercises to test the designed circuits ranging from simple assignments to processor architectures and communication controllers.

1 Background and Motivation

The traditional view of computer systems splits them into two major components: hardware and software. The former comprises all physical parts, the latter simply the rest. A less simplistic view recognises that systems are built as *hierarchies of abstractions,* of which only the lowest level is concrete, implemented using physical parts. The concept of abstraction layers is well known, but still inadequately mastered in the realm of software. It is equally present in hardware design. The best example of a successful abstraction is perhaps TTL technology; it allows "soft" hardware design, i.e. the design of circuits without knowledge of the physical properties of components (gates, register), but instead merely relying on the TTL-abstraction based on Boolean logic and gate delays. A second, albeit less successful example is the design of integrated semiconductor circuits using the Mead-Conway stick diagrams.

The borderline between hardware and software in this tower of abstractions is diffuse, and perhaps irrelevant. There is a growing recognition of strong similarities in the design processes of hardware on the one hand and software on the other. Not only software, but also hardware is becoming more complicated, reaching degrees of complexity that can be mastered only with the aid of precisely defined formalisms and tools based on them. They are texts which allow the formal, symbolic manipulation according to fixed rules. Instead of texts, graphic forms have traditionally been used to represent digital circuits. However, texts are much better suitable for formal manipulation, the main reason being the facility of textual substitution. Formalisms for the expression of digital circuits are called *hardware description languages* (HDL).

There exist obvious similarities between HDLs and conventional programming languages. In programs of both kinds there exist declarations of objects, i.e. of variables and signals, and there exist statements defining computations

using expressions of variables and operators. In both realms composite, structured objects can be expressed and repetitions can be formulated, which in the case of circuits stand for replications of circuit patterns.

A principal difference between programming languages and hardware description languages lies in the interpretation of their programs. Whereas in a traditional program statements are interpreted (executed) in strict sequence in time, those of a circuit description are interpreted concurrently, all of them. Repetition in time is implied by an inherent clock signal acting on registers. This holds for strictly *synchronous* circuits. Asynchronous circuits complicate circuit design in the same way as the introduction of concurrency in sequential programs complicates programming, namely very considerably.

We wish that these similarities of concepts and formulations be reflected by our courses educating computer scientists and engineers. The discovery of similarities between a new subject and a familiar discipline is a strong incentive for raising interest in the new subject. Hence, the use of an HDL, and of a structured approach similar to programming, increases the motivation of computer science students to study circuit design.

The process of executing a program written in an HDL differs from that of a programming language in a significant way: Whereas execution of a "software" program yields the desired computational results, the execution of a "hardware" program is supposed to yield a physical circuit. The path from program to result is therefore much longer and difficult in the latter case, as are testing and verification. Depending on the technology at hand, the latter path includes several non-formalized steps subject to numerous errors and oversights, particularly if manual labor is involved such as soldering and wire-wrapping.

An unavoidable step in transforming a circuit description (using an HDL) into a circuit is the creation of a layout. A layout specifies the geometric placement of components on a plane (circuit board or silicon wafer) and the routing of their connections on several planar layers. Modern systems aim at performing this transformation from logic specification to geometric layout automatically, even generating layout specifications in a manner that the physical circuit can be fabricated mechanically, be it in the form of a printed circuit board (PCB) or an integrated circuit (chip).

These steps are usually considered as too involved and too expensive to be included in student exercises. As a result, the generation of a physical circuit is replaced by its simulation by computer. This escape route, however, apart from dodging many practical issues, deprives the computer science student from the remnants of motivation, from the experience also of creating a genuine, functioning, physical device.

The recent advances of programmable logic devices (PLDs), in particular programmable gate arrays (FPGAs) paved the ground for a welcome alternative. They allow the realization of circuits by redefining the functionality of their individual cells and by reconfiguring their routing facilities. In the case of SRAM-based parts there is no limit to their being reused. Reconfiguration occurs by simply down-loading a bit-stream computed by a layout generator.

The user of an FPGA is not as free in designing a solution as the engineer using individual gates (e.g. in VLSI design). The constraints are determined by the existing choices of cell functions and routing resources inherent in the FPGA at hand. The simpler the cells and the more universal the routing resources, the fewer the constraints. It follows as a corollary that automatic layout generation for an FPGA is the more easily feasible, the finer-grained the FPGA cells are.

2 The Lola System - An Overview

The Lola System was conceived with an HDL as a starting point and the use of FPGAs and PLDs as an end point of the circuit design process is mind. The system is particularly well-suited, but not confined to use in courses and laboratory exercises.

A first task in the system's development was the choice of a language. The only contenders from the industrial sector were VHDL and Verilog, as they appear to be increasingly used due to their early standardization. Upon closer inspection, however, both turn out to be quite inappropriate for educational purposes. An obvious deterrent is the sheer size of their defining documentation and, as a consequence, their software. Nothing is more detrimental to a learning process than being overwhelmed. In this respect, the situation is deplorable in the field of "software" programming; we believe it is worse in the field of "hardware" programming. The complexity and wealth of features and facilities of both VHDL and Verilog is mind-boggling. What makes the situation even worse is the poorly hidden fact that these languages are designed to specify executable programs simulating the behaviour of circuits rather than the circuits themselves. This causes a diversion of the students' focus of attention: we aim at the abstract description of circuits as directly as possible, and certainly not via their simulation programs. Our vision of HDL texts are static descriptions rather than executable code.

Our reaction was to design a new HDL, one which would confine itself to the essential ingredients of circuits and thereby be highly suitable for, but not confined to educational use. The resulting simple language *Lola* is described in detail in Section 3. In order to exhibit the inherent similarities to programming languages, a syntax similar to that of Oberon [1] was chosen. The language features elementary and structured data types, expressions and assignments, and an analogon to procedure calls, called *unit assignment.* A unit is essentially an instance of a structured type, which in analogy to object-oriented languages combines local data declarations with statements and expressions defining their values.

The Lola System consists of a part that depends on the device (or technology) by which the circuit is to be implemented, and a part which does not. The latter, also called *front end,* consists of the compiler which translates a Lola text into a data structure representing the program, and of an interpreter which expands (flattens) the compiler's output into another data structure which directly represents the circuit, i.e. where every node stands for a gate or register, and

every edge represents a connection (wire). These data structures are described in Section 4.

The device-dependent part(s) of the system, the *back ends,* generate layouts for particular parts (FPGA, PLD) or for a chosen technology (VLSI). Automatic layout generation is easier for PLDs than for FPGAs due to the formers' generality for function implementation. In our system, back ends have been built for the PLDs GAL22V10 (Lattice Logic) and MACH211 (AMD). They are described in more detail in Section 5. Here we point out that the problem of automatic placement is circumvented by requiring the circuit designer to specify the assignment of outputs to macro cells. For this purpose, a facility to associate *position numbers* with declared signals has been incorporated in the language Lola. As a consequence, automatic routing becomes reasonably straight forward. More recent, more complex PLD designs have the effect of diminishing this advantage of PLDs over FPGAs.

Whereas PLD's essentially represent a one-dimensional array of macrocells and associated product terms, FPGAs are two-dimensional arrays of cells. This fact makes automatic mapping of a circuit onto the cell matrix more difficult by an order of magnitude. An automatic router and placer - the latter allowing for hints provided by the designer interacting with the system - have been developed for the fine-grained Xilinx XC6200 FPGA [2, 6]. This back-end benefits greatly from a structured description of the circuit to be generated, and the fact that structural information is retained by the compiling process.

In a system designed for educational use, however, we wish to avoid the black-box effect resulting from automatic placement and routing. Here, the purpose is to provide understanding of the issues involved in a design and development process rather than merely crude knowledge about how to employ a certain tool. As a consequence, the Lola System is equiped with an interactive, display-oriented *layout editor* briefly presented in Section 5.

At this point, specification of a circuit in terms of a Lola module on the one hand, and configuring a certain FPGA to represent this same circuit on the other hand, appear as disjoint activities. The former may even be considered as superfluous, because the latter alone leads to a functioning circuit. However, an essential bridge between Lola and layout editing is established by a consistency checker, a program that determines whether or not a submitted layout is consistent with a given Lola specification. Naturally, this *Checker* belongs to the device-dependent part of the Lola System. Experience has amply demonstrated the advantage of constructing the layout *after* specifying the circuit in terms of a Lola module, and then using the Checker to verify consistency. The Checker is indeed a powerful aid in quickly locating many trivial mistakes that can hardly be avoided when generating a layout by hand. An important point is that - in conjunction with Lola's facility to specify circuits in a structured, hierarchical way - layouts can also be checked in a hierarchical order: subcircuits first, compositions thereafter. The configuration of the extensible Lola System is shown in Fig. 1.

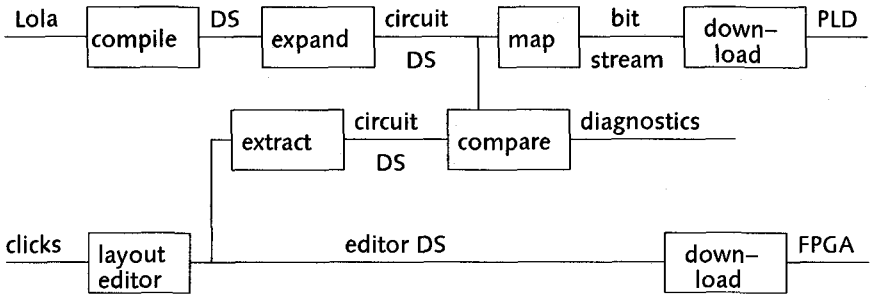

Fig. 1. The Lola System

3 The Language Lola

3.1 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.

3.2 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}.

3.3 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.

3.4 Constant Declarations

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

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

3.5 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}.

3.6 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 (^). In the latter case, the base must be 2.

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

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 3.10) corresponding to a formal input variable.

3.7 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).

3.8 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" .
ForStatement = "FOR" identifier ":=" expression ".." expression "DO"
```

```
        StatementSequence "END" .
UnitAssignment = identifier selector "(" ParameterList ")".
ParameterList = parameter {"," parameter}.
parameter = expression | constructor.
constructor = "[" expression {"," expression} "]" .
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.

3.9 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 3.6).

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.
```

3.10 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 identifier at the end of the declaration must match the one following the symbol TYPE.

3.11 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)
```

3.12 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.

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

4 The Front-End

4.1 The Compiler

The compiler, translating Lola texts into a data structure, is based on the well-known top-down, recursive descent parsing technique. The language syntax is designed such that a lookahead of a single symbol is sufficient [3]. A scanner reads the source text and delivers a symbol sequence, where identifiers and numbers are considered as atomic symbols. The compiler generates the data structure in a single pass over the source text. The resulting data structure is built in the main store, and is not deposited on disk store, i.e. it appears as a global variable accessible to other parts of the system. This way of passing data from one phase of a process to another is easily possible and common practice in the Oberon System [7]. It avoids complicated externalizing and subsequent internalizing operations and thereby contributes significantly to processing efficiency.

While accepting symbols from the scanner and generating elements of the output structure, the parser also performs a modest form of type checking. The translation process is fairly straight forward, and the output is essentially a syntax tree, in which nodes correspond to terminal symbols in the source text. Every node is characterized by a *tag,* essentially reflecting the corresponding terminal symbol, and two (pointers to) descendant nodes (a and b). Some nodes, such as numbers, feature a fourth attribute called *value.* Hence, the syntax tree is a binary tree. Syntactic structures being lists are represented as degenerate trees.

While processing declarations, a compiler typically builds a symbol table, and while processing statements, it generates code. In the case of the Lola compiler, a data structure is constructed in both cases, and nodes (called *items*) of the *syntax tree* may reference nodes (called *objects*) of the *symbol table* (see Fig. 2). In fact, objects are a type extension of items, with the additional attributes of a string representing the *name* and a pointer to a *next* object in the list. The attribute *class* indicates whether the object denotes an input, output, or local variable.

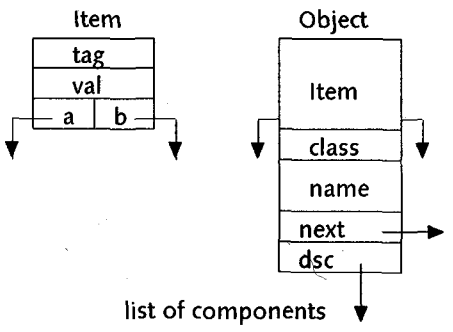

Fig. 2. Elements of the data structure generated by the compiler

4.2 The Expander

Expansion may be considered as the execution of the output of the Lola compiler, during which a direct representation of the formulated circuit is generated. The term *expansion* stems from the process of "unrolling" repetitive constructs, such as array declarations and for statements. The new, expanded data structure is again a binary tree with nodes representing anonymous *signals* and extensions thereof, representing named signals called *variables* (see Fig. 3). Operators with

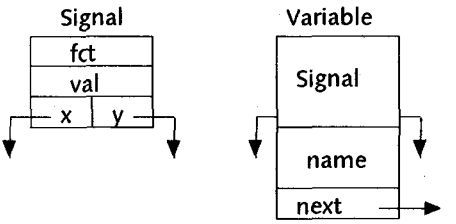

Fig. 3. Elements of the data structure generated by the expander

more than two operands are represented by a fixed sequence of binary nodes:

```
MUX(s: x, y)     ->    [mux, s, [mux', x, y]]
REG(ck: enb, d)  ->    [reg, ck, [reg', enb, d]]
```

The expander consists essentially of four recursive procedures for evaluating numeric expressions, signal expressions, assignments (statements), and declarations respectively. Lola programs feature variable declarations textually preceding the definitions of their values by statements, and likewise the evaluation of declarations precedes that of statements. The former generates part of the tree which represents names signals, the latter associates subtrees corresponding to expressions with them. Numeric expressions are evaluated into numbers. This implies that the lengths of arrays and the number of repetitions in FOR statements become manifest, thus allowing the specified number of replicas to be generated and evaluated. In the case of a unit assignment, the data structure representing the type to be instantiated is copied, the actual parameters are evaluated and linked as inputs into the cloned instantiated structure. This process may occur recursively in case of type declarations containing further instantiations (unit assignments).

After the expanded data structure has been generated, a simplification pass is performed. Although the most obvious simplifications of Boolean expressions had been handled by the compiler, instantiation of structured types may generate new possibilities for simplification when parameters with constant values are supplied. The cases considered are the following:

```
~0 = 1          ~1 = 0          ~~x = x
x + 1 = 1       1 + x = 1       x * 0 = 0      0 * x = 0
```

x + 0 = x	0 + x = x	x * 1 = x	1 * x = x
x - 0 = x	0 - x = x	x - 1 = ~x	1 - x = ~x
MUX(0: x, y) = x	MUX(1: x, y) = y	~x - 1 = x	1 - ~x = x

After the simplification pass follows a traversal of the data structure to discover the presence of (combinational) loops representing race conditions. This operation is explained in terms of node coloring. Initially, let all nodes be black. Then, for each named signal, its value defining tree is traversed, coloring each visited node grey. The traversal of a branch stops, if a register or a named, white node (visited variable) is encountered; if a grey node is reached, a loop is present. On the return path of a traversal, nodes are colored white. Hence the colors' meaning is *untouched* for black, *in process* for grey, and *visited* for white. Evidently, the data structure is not a tree, but a directed, acyclic graph, provided that registers are considered as cuts. Note that latches are considered as potentially transparent. Hence they cannot act as cuts.

4.3 The Simulator

A simple simulation package is also part of the Lola System. It allows to simulate the functioning of synchronous circuits, i.e. circuits where all registers operate with the same clock. The simulator operates directly on the expanded data structure. It modifies (owns) the node field *val,* recording every signal's value in each clock cycle (simulation step). The data structure is traversed in the same manner as for loop detection. The structure is traversed in depth-first manner, computing a node's value from those of the descendants. Again, registers serve as terminal points in the traversal.

s # NIL -> value(s) = fct(value(s.x), value(s.y))

Registers are the carriers of values from each cycle to the next cycle. Therefore, register nodes inherently must carry two values, namely the value resulting from the last cycle and being used in the current cycle, and that obtained in the current cycle. Since registers have 3 inputs (clock, enable, data), they are represented as two (binary) nodes, which comes in handy at this point. The computed value of a node R is assigned to R.y.val (if the enabling signal value(R.y.x) = 1). At the end of each cycle, this value is moved to its proper place: R.val: = R.y.val. The values s.val of D-latches and SR-latches, which are basic operators of Lola, are defined as follows:

D-latch:
    IF value(s.x) # 0 THEN s.val := value(s.y) END

SR-latch:
    IF value(s.x) = 0 THEN s.val := 1
    ELSIF value(s.y) = 0 THEN s.val := 0
    END

A simulator is necessary if hardware to represents the circuit is unavailable. However, it is handy even when hardware is available, because it may help in locating undefined signals that might give rise to spurious errors. This is achieved by using four-valued signal variables rather than binary ones. The values are: 0, 1, *undefined, contended.* Before a sequence of simulation steps is executed, all signals are given the value *undefined.* Any operation with an undefined operand value yields that value as result. *Contended* is the result of multiple, concurrent assignments to a (tri-state) bus.

5 Backends

5.1 A Layout Editor

Backend components are device-dependent by definition. The display-centered, interactive Layout Editor for the Atmel 6000 FPGA is to serve as an example. We refrain from discussing its internal structure, referring to other publications [4]. Instead we confine our brief description to the editor's use and functionality. An editor is intimately coupled with the specific type of device for which it serves. We therefore precede its description with a cursory presentation of the particular FPGA's architecture, skipping details [5].

The device consist of a matrix of 32x32 cells, grouped in 8x8 cell blocks. Every cell has two outputs available as input in each of the 4 adjacent cells. In addition to this, there exist buses connecting 8 cells in line. A bus is connectable to its corresponding bus in neighbouring blocks by repeater switches.

Considering the cell architecture, we distinguish between its core and its periphery. The core has 2 inputs (a, b), 2 outputs (A, B) and 4 possible *states* (configurations). Two of the states serve for routing only, the remaining ones form a half-adder and a counter cell respectively:

A := a	A := b	A := a-b	A := REG(a-b)
B := b	B := a	B := ~(a*b)	B := a*b

The periphery connects the core with neighbouring cells and with buses, and it assumes 4 *modes* (configurations). Interestingly, a certain combination of state and mode yields a multiplexer. This useful combination is displayed by an extra entry in the mode menu:

A := MUX(L: a, b) B := 1 (L is the bus input)

Available states and modes are displayed by the editor through popup menus. They may be freely combined.

Whereas configurations of cells, of I/0 cells connected to pins, and of bus repeaters are selected via popup menus, routing through cells and via buses is achieved by clicking a mouse button while pointing at the specific, sensitive area where a connection is to occur. This interactive mode of "wiring" allows for very rapid construction of designs, and also for immediate corrections.

Apart from configuration operations, the editor also allows shifting the FPGA picture within a viewer (window), for moving and replication of entire groups of cells, and for presenting different excerpts of a design in multiple viewers. Furthermore, designs can be externalized as files.

5.2 The Checker

As previously mentioned, the *Checker* serves to determine whether or not a circuit's implementation (its layout) is consistent with its specification (its Lola program). For this purpose, the circuit is first "extracted" from the layout, i.e. the layout is mapped into a data structure consisting of the same types of nodes as the one generated by the expander [4]. Only then can the extracted structure representing the implementation be compared with the expanded structure representing the specification. This comparison between structures occurs for all corresponding signals. In order to establish correspondences, signals in the layout must be labelled with the names used in the Lola specification.

Unfortunately, not all elementary Boolean operations can be represented directly by single FPGA cells. For example, the Atmel 6000 FPGA contains no Or-gate. It must be constructed using an And-gate and inverters, where inverters themselves must be formed by Nand-gates with one input set to 1. Other examples where no direct equivalent is available on the FPGA, are the latch and the multiplexer. The extractor therefore is designed to detect the equivalent circuits and translate them into their expected form:

```
q = x*1 - (~x*y)      ->    q = x OR y
q = ~s*x - s*y        ->    q = MUX(s: x, y)
q = MUX(en: q, x)     ->    q = LATCH(en, x)
q = ~(s' * ~(r'*q))   ->    q = SR(s', r')
```

The implementation of functions with restricted resources is called *technology mapping*. The comparator must take restrictions into account by making use of simplification rules, de Morgan's rules, and distributive and associative laws. This is no trivial task, and care must be taken to reasonably confine the search for equivalent expressions in order to maintain efficiency.

The Checker not only discovers inconsistencies, but moreover indicates their origin by showing the concerned signal's expression as specified and as implemented. It also highlights the affected cell in the layout.

5.3 A PLD fuse map generator

PLDs and PLAs, that is, and/or-matrices with and without registers, were the first programmable components available. Compared to FPGAs with their matrix of cells, automatic routing and placement for PLDs is simpler due to the one-dimensionality of their architecture and the generality of the and/or gate matrix.

It is therefore possible to convert a Lola circuit specification automatically into a PLD configuration (fuse map) without overly complex software. The translation is further simplified, if some restrictions are imposed on the source specifications, which are necessary anyway because of the finite facilities offered by the device. In the Lola System, fuse map generators for the devices GAL22V10 (Lattice Logic) and MACH211(AMD) are currently available, both for their In-System-Programming (isp) versions. The restrictions imposed on the Lola texts are:

1. The output signals must be defined in one of the following 4 forms.

   $y_i := F_i(x, y)$ $\quad$ $y_i := REG(F_i(x, y))$
   $y_i := \tilde{}F_i(x, y)$ $\quad$ $y_i := \tilde{}REG(F_i(x, y))$

   where x stands for the m input signals x_i, y for the n output signals y_k,and F_i for a Boolean function specified as an expression.
2. If tri-state (INOUT) signals occur, they must be defined in one of the following forms:

   $y_i := e_i \mid F_i(x, y)$ $\quad$ $y_i := e_i \mid h_i;$ $\quad$ $h_i := REG(F_i(x, h))$
   $y_i := e_i \mid \tilde{}F_i(x, y)$ $\quad$ $y_i := e_i \mid \tilde{}h_i;$ $\quad$ $h_i := REG(F_i(x, h))$

   where e_i is a term and h_i must be a local variable.
3. The numbers of input and output signals (and of product terms) have upper limits which follow from the architectures of the respective PLD design.
4. The Fuse Map Generator does not perform automatic placement. Instead, the placement of output signals to macro cells must be specified in the Lola text. For this purpose, the language features a so-called *position assignment,* allowing to associate a position, i.e. a macro cell number, to a signal. The pin number then follows from the macro cell number.

The fuse map generator program proceeds in the following steps, being controlled by the data structure resulting from the Lola text by compilation and expansion. There are two passes, in which the list of declared output variables is scanned, and the following actions are performed with each variable encountered:

1. For each output signal, the configuration of its macrocell is determined according to the form of the defining statement. The validity of the position number is checked. Multiple cell assignments are detected.
2. For each output signal, its expression is translated into entries in the fuse map. The expression is converted into disjunctive normal form, i.e. into a sequence of product terms. In a first tree traversal, exclusive ORs and multiplexers are converted, and illegal operators such as register and latches are diagnosed as errors. In a second scan, normal form is established by using de Morgan's rules and by copying subexpressions.

6 Summary and Conclusions

The advent of large-scale, programmable devices and gate arrays causes the borderline between hardware and software to disappear, and it lets the similarities between hardware and software design methods become apparent. Textual specifications begin to replace graphical schematics and bring the advantages of structured design and description into the domain of circuit design.

In accordance with this trend, we designed the language Lola. In contrast to the widely used languages VHDL and Verilog, Lola focuses on the essentials and describes circuits instead of programs simulating their behaviour. As a result, the length of defining documentation, the effort for learning and mastering the notation, and the size of its implementation are but a fraction of what is in common use today (see Table 1). These properties make the language Lola highly

module	lines	words	bytes (obj code and data)	
LSB	244	1237	4060	Lola System Base
LSC	591	3219	7800	Compiler
LSS	173	1003	3740	Scanner
Lola	243	1329	4060	Expander
Simulator	251	1210	3110	
GAL22V10d	416	2164	11870	
MACH211	427	2083	6820	
M211	279	1427	21740	Bitstream Generator

Table 1. Size of the Lola System (NS 32000)

suitable for educational use. But it does not replace experiments in the laboratory. The Lola System is therefore augmented by components that automatically compile a Lola program into a fusemap for a PLD and thereby allow the testing of physical hardware. Furthermore, a layout editor for an FPGA is provided together with a program that detects the presence or absence of consistency between specification and implementation (layout).

The use of a language and of structured design techniques familiar from software programming, together with the availability of programmable devices for testing the specified circuits, provides a significant motivation for computer science students to study hardware design. In research and development, these facilities encourage the joint design of hardware and software, and they provide benefits of the synergies resulting from such an approach.

References

1. M. Reiser and N. Wirth. *Programming in Oberon.* Addison-Wesley, 1992, ISBN 0-201-56543-9.
2. S. Gehring and S. Ludwig. The Trianus System and its Application to Custom Computing. *Proc. of the 6th International Workshop on Field-Programmable Logic and Applications.* Springer-Verlag, 1996.
3. N. Wirth. *Compiler Construction.* Addison-Wesley, 1996.
4. S. Gehring and S. Ludwig. A Laboratory for a Digital Design Course using FPGAs. *Proc. of the 4th International Workshop on Field-Programmable Logic and Applications.* Springer-Verlag, 1994.
5. Atmel Corp. Configurable Logic: Design and Application Book, 1995.
6. Xilinx Corp. XC6200 FPGA Family. Advanced Product Description, 1996.
7. M. Reiser. *The Oberon System.* Addison-Wesley, 1991, ISBN 0-201-54422-9.