ATChecker User Manual

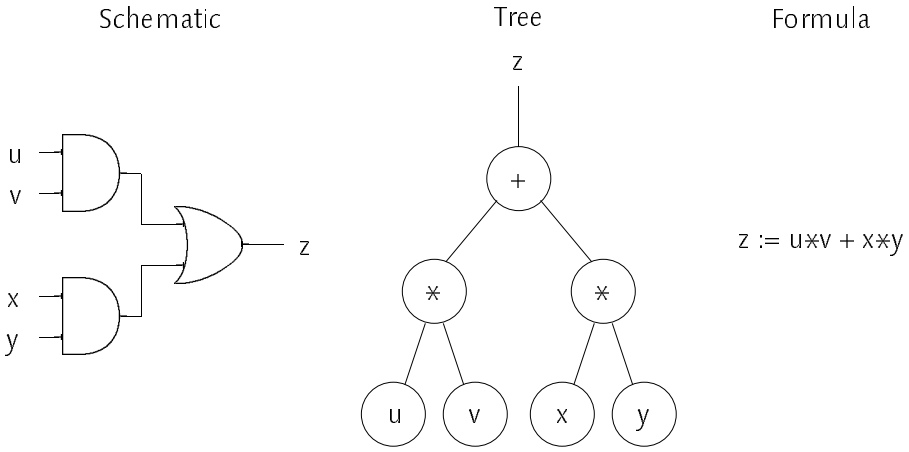
Stephan W. Gehring

Abstract

The ATChecker program tests a layout implemented with the aid of the AT–Editor for conformance with a specification in the form of a Lola program. As a base for comparing the two representations of a digital circuit, the checker uses a data structure based on a set of binary trees generated by the Lola compiler. To simplify the conformance check, the internal representation of the layout is first transformed into an equivalent data structure. Then the two sets of trees are matched to detect inconsistencies. Inconsistencies found are displayed textually and also marked in the layout to facilitate error locating.

How it works

A digital circuit is characterized by its inputs, outputs, and a set of Boolean functions combining the inputs. Each circuit output is associated with the result of such a function. It can be represented as a *binary tree* with nodes consisting of Boolean constants, operators, variables, or units composed of several operators (e.g. multiplexors, registers). Variables, in this context, designate signals associated with a name, such as an input or an output. Each output forms the root of such a binary tree. A complete circuit can thus be represented as a set of trees, one for each output. Inner tree nodes consist of operators with edges pointing towards the node's inputs, while leaf nodes consist of Boolean constants and input variables.
The following example illustrates the equivalence between a Boolean function represented as a set of interconnected gates, a binary tree, and a Lola statement.

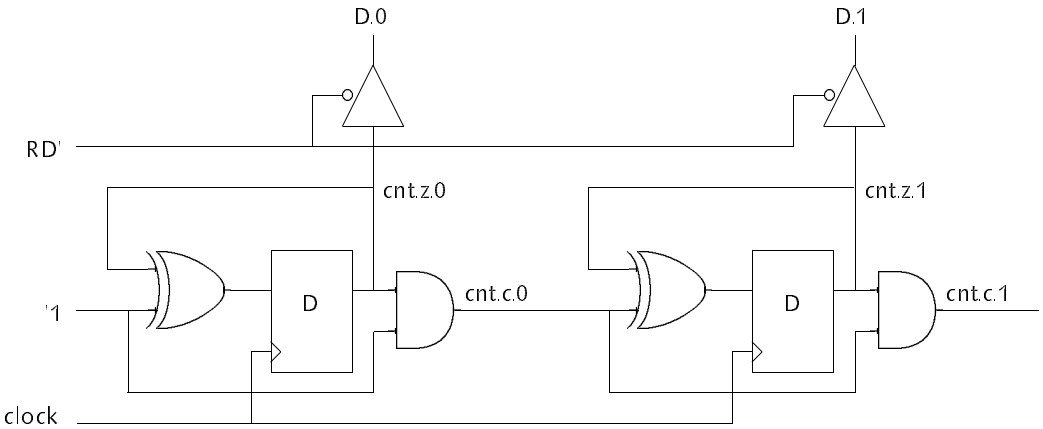


As shown above, statements of a Lola program can be represented by trees as well: trees serve as a common representation for layouts and Lola programs and hence provide the base for comparison.
By transforming both a layout and a corresponding Lola program into a set of trees the two representations can be matched. Under the assumption that the Lola program describes the circuit correctly, i.e. it depicts a *circuit specification*, inconsistencies between corresponding pairs of trees are interpreted as errors in the layout, the *circuit implementation*. Two trees correspond if the names of their root variables match.

Transformation of Lola programs into a set of trees is done by the Lola compiler already, so the compiler's output can be directly used for comparison. The layout entered with the AT–Editor, however, is stored in an internal format and must be transformed into an equivalent set of binary trees before matching. This is done by a circuit extractor, which recursively traverses signals in the layout starting at each output signal and terminating at input signals or Boolean constants. After processing a gate's inputs, a node equivalent to the gate's operation is generated and linked accordingly. The resulting tree is then checked for combinational loops (other than SR flip–flops and latches implemented with multiplexors). After a simplification step the tree is ready to be checked for consistency with the Lola specification.

Example

The following example of a two–bit binary counter illustrates the steps necessary to implement and verify the correctness of circuits entered with the AT–Editor. The schematic below contains two counter elements whose outputs (cnt.z.0, cnt.z.1) are fed to a bus D through tri–state gates.



Step 1: The Lola description

In a first step, a Lola program, which will serve as a reference ("specification") for the checker, is developed. It consists of a single module with a parameterized type *Counter* which defines an N–bit binary counter. The module body instantiates a 2–bit counter and assigns its outputs to the bus. Note that the circuit is synchronous and the register clock hence implicit.

```
MODULE Example;

TYPE Counter (N);                       (* N–bit binary counter *)
IN incr : BIT;                          (* counter enable *)
OUT z : [N] BIT;                        (* output bits *)
VAR c : [N] BIT;                        (* carry bits *)
BEGIN
    z[0] := REG (z[0] – incr);
    c[0] := z[0] * incr;
    FOR i := 1..N–1 DO
        z[i] := REG (z[i] – c[i–1]);
        c[i] := z[i] * c[i–1]
    END;
END Counter;

CONST N := 2;
IN RD' : BIT;                           (* bus read control *)
INOUT D : [N] TS;                       (* output bus *)
VAR cnt : Counter (N);                  (* counter *)
BEGIN
    cnt ('1);                           (* build counter *)
    FOR i := 0 .. N–1 DO
        D[i] := ~RD' | cnt.z[i]         (* assign counter output to bus D*)
    END
END Example.
```

The program can now be compiled with the Lola compiler (*Lola.Compile*). The compiler generates a set of trees which are displayed as a list of Boolean expressions (*Lola.Show*):

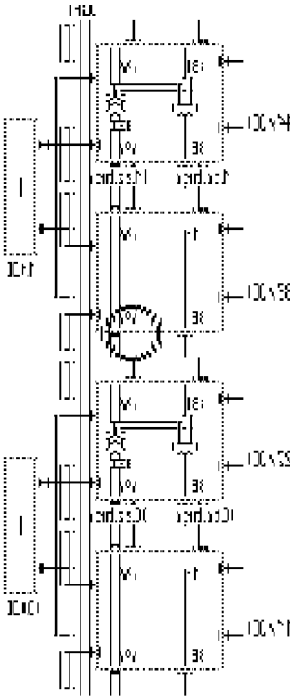
```
expanding Example
cnt.incr := '1
cnt.z.0 := (CK↑('1:(~cnt.z.0)))          (* ↑ = register, ~ = negation *)
cnt.z.1 := (CK↑('1:(cnt.z.1-cnt.c.0)))   (* - = XOR *)
cnt.c.0 := cnt.z.0
cnt.c.1 := (cnt.z.1*cnt.c.0)             (* * = and operator *)
D.0* := (((~RD')|cnt.z.0),)              (* | = tri-state gate *)
D.1* := (((~RD')|cnt.z.1),)
RD'*
```

Step 2: Design Entry

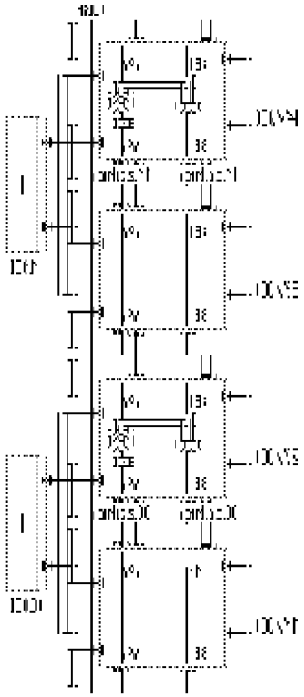
With the Lola compiler output at hand, the design can now be entered with the AT–Editor. The figure below (left) displays the layout of the two counter elements, the higher order bit on top of the lower order bit. For each variable listed in the Lola compiler output a corresponding label must be placed in the layout.

Step 3: Consistency Check

After specifying the circuit with a Lola program and entering it with the AT–Editor, the two circuit representations are ready to be checked for consistency.



Faulty first implementation: connection between lower and upper counter elements missing (circle)



Corrected implementation: carry (cnt.c.0) is fed to the upper counter element

Assume that the (faulty) layout on the left has been entered in a first attempt. To check the implementation of variable cnt.z.0 we mark ("*") the viewer that displays the layout and execute the command

```
ATChecker.Check cnt.z.0
```

From the entered circuit, the checker first extracts the binary tree for variable cnt.z.0 and then matches the tree against the corresponding tree generated by the Lola compiler. The trees match and the checker issues the message

```
cnt.z.0 –> matches
```

Next we check variable cnt.z.1:

```
ATChecker.Check cnt.z.1

cnt.z.1 –> error
spec: cnt.z.1 := (CK↑('1:(cnt.z.1–cnt.c.0)))
impl: cnt.z.1 := (CK↑('1:(~cnt.z.1)))
      cnt.z.1 := (CK↑('1:(~cnt.z.1 <– XOR expected at   0  4  AOut
```

As seen from the resulting error message, the checker detects an inconsistency between cnt.z.1 as defined by the Lola program (*spec:*) and its actual implementation in the layout (*impl:*) which are both displayed in the error message. The checker then indicates the cause for the mismatch and also selects the mismatch location in the layout to aid the user in locating the error more quickly.
Before checking for consistency, the checker extracts the binary tree for output variable cnt.z.1. It follows the signals in the layout recursively beginning at the output and terminating at Boolean constants and input variables. During this process, the checker passes the register and the XOR gate in cell 0/4 and generates the appropriate nodes in the tree. The inputs to the XOR node are cnt.z.1 from cell 0/4 and the Boolean constant '1 from cell 0/3, hence cnt.z.1 = REG (cnt.z.1 – '1). When the expression is simplified, (cnt.z.1 – '1) is reduced to ~cnt.z.1. When the trees are matched, the Lola specification indicates an XOR gate which has no counterpart in the layout since it merely contains a negation at that node in the tree. Therefore, the error message

```
cnt.z.1 := (CK↑('1:(~cnt.z.1 <– XOR expected at   0  4  AOut
```

is issued. The error in the layout is the result of not feeding the carry output of the lower counter element to the upper one. After connecting cnt.c.0 to the upper counter element (figure above right) cnt.z.1 can be checked anew and is now found to be correct:

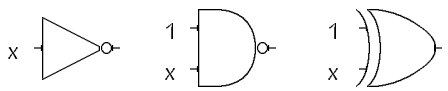
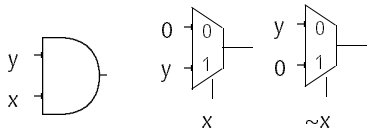
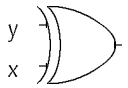
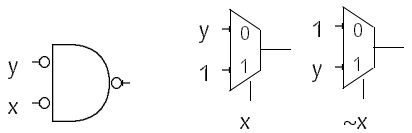
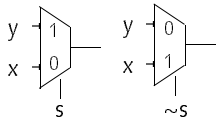
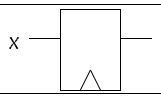
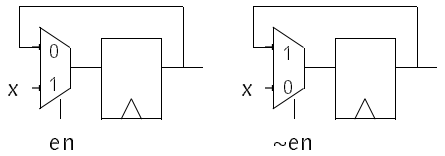
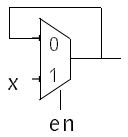
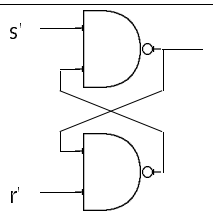
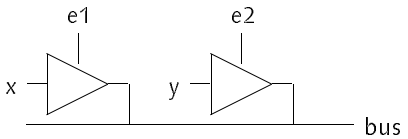
```
ATChecker.Check cnt.z.1
cnt.z.1 –> matches
```

The command *ATChecker.CheckAll* checks all variables defined in the Lola program:

```
ATChecker.CheckAll
cnt.z.1 –> matches
cnt.z.0 –> matches
cnt.z.1 –> matches
cnt.c.0 –> matches
cnt.c.1 –> matches
D.0 –> matches
D.1 –> matches
all variables match
```

Representation of Lola Constructs

The following table explains the possible representations of Lola constructs in the AT–Editor. Between two constructs, any even number of inverters is allowed. Gates and attached inverters (e.g. NAND gates) need not lie in the same cell in the layout. Note that the AT6002 FPGA does not feature an OR gate. Since the checker supports the laws of DeMorgan, a specified OR gate can be substituted by a NAND gate with inverted inputs in the layout.

Function	Lola Representation	AT–Editor Representation
Constants	'0, '1	0, 1
Variables	counter.out	counter.out (Label)
Not	~ x	
And	x * y	
Xor	x – y	
Or	x + y	
Multiplexer	MUX (s: x, y)	
Register	REG (x)	
Register	REG (en, x)	
Latch	LATCH (en, x)	
SR Flip–flop	SR (s', r')	
Tri–state	bus := e1 | x; bus := e2 | y	

Features and Restrictions

Inverters	successive pairs of inverters are ignored
DeMorgan	the laws of DeMorgan are applied: a * b = ~(~a + ~b), a + b = ~(~a * ~b)
Commutativity	commutativity is supported: a * b = b * a, a + b = b + a, a – b = b – a
Associativity	In favor of a simple and efficient implementation of the ATChecker, associativity of Boolean operators is not fully supported. Only a subset of equivalent expressions is therefore considered equal, such as

(a ∗ (b ∗ c)) ((a ∗ b) ∗ c)
((x – y) – z) (x – (y – z))
(a + (b + (c + d))) (((a + b) + c) + d)

but not

((a ∗ b) ∗ (c ∗ d)) ((a ∗ c) ∗ (b ∗ d)).

A simple workaround for problems related to associativity is to parenthesize expressions in the Lola program according to the actual implementation.

MUX selectors — inverted multiplexor selectors are allowed: MUX (s: a, b) = MUX (~s: b, a)

OR/AND with MUX — OR and AND gates can be implemented with MUX:

a + b = MUX (a: b, '1)
a ∗ b = MUX (a: '0, b)

Hints

1. Write the Lola program before implementing the circuit with the AT–Editor. For one, the Lola program makes regularities in the circuit more clearly visible. Often, these regularities can later be exploited when laying out the circuit with the editor. Second, the names of the variables and hence labels in the layout are found in the Lola compiler output displayed in the log viewer.
2. Use a reasonable number of variables. Searching for implementation errors in variables which implement complex functions can be very cumbersome. It can therefore be advisable to break complex expressions into simpler ones by inserting variables.
3. Check variables with simple functions before checking variables with complex functions. Variables which implement complex functions often share subexpressions with variables implementing simpler functions. After verifying the correct implementation of simpler variables, verifying complex ones can be tackled more easily.

List of Commands

ATChecker.Check (name | "↑") checks the implementation of the given variable based on its specification in the output of the Lola compiler. The parameter designates the name of the variable to check. The marked ("∗") viewer designates the AT–Editor viewer which contains the implementation. If no viewer is marked and the command is not in the menu frame, the viewer containing the most recent selection is used.

ATChecker.CheckAll checks the implementation of all variables listed in the output of the Lola compiler. The command stops when all variables have been successfully checked or when an implementation error is detected.

ATChecker.Show (name | "↑") displays the function associated with the implementation of a variable as an expression. As its parameter Show takes the name of the variable to show. The marked ("∗") viewer designates the AT–Editor viewer which contains the implementation of the variable. The user is notified of combinational loops other than SR flip–flops and latches.

Necessary Modules

The checker consists of the following files which are available on the Pluto server:

Checker	matching algorithm
ATChecker	tree extraction from AT–Editor layouts
LolaGen	tree construction operations