The United Nations University

UNU/IIST

International Institute for
Software Technology

A Verilog Specification of STARI

Pablo Giambiagi and Gerardo Schneider

March 18, 1998

UNU/IIST Report No. ???

$\boxed{\mathcal{R}}$

UNU/IIST and UNU/IIST Reports

UNU/IIST is a Research and Training Center of the United Nations University. It was founded in 1992, and is located in Macau. UNU/IIST is jointly funded by the Governor of Macau and the Governments of China and Portugal through contribution to the UNU Endowment Fund.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. advanced development projects in which software techniques supported by tools are applied,
2. research projects in which new techniques for software development are investigated,
3. curriculum development projects in which courses of software technology for universities in developing countries are developed,
4. courses which typically teach advanced software development techniques,
5. events in which conferences and workshops are organised or supported by UNU/IIST, and
6. dissemination, in which UNU/IIST regularly distributes to developing countries information on in ternational progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively parti cipate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research $\boxed{\mathsf{R}}$, Technical $\boxed{\mathsf{T}}$, Compendia $\boxed{\mathsf{C}}$ or Administrative $\boxed{\mathsf{A}}$. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2001

The United Nations
University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

A Verilog Specification of STARI

Pablo Giambiagi and Gerardo Schneider

Abstract

Verilog is a Hardware Description Language used for the design and description of hardware in a behavioral and structural way. It has some interesting features like concurrency, synchronism, shared variables, non-blocking assignments (scheduled assignments), timing controls, infinite computations, zero-time computations, etc., that makes it an interesting language to study. This report explains some features of Verilog in an informal way through small examples and presents the Verilog code of STARI as a main application.

Copyright © 1998 by UNU/IIST, Pablo Giambiagi and Gerardo Schneider

Contents

1 Introduction

Verilog is a Hardware Description Language used for the design and description of hardware in a behavioral and structural way. Hence, it is possible to write an "abstract" program (that represents the behaviour of the system) and a more "concrete" (implementation, structural) one. The meaning of a Verilog program is usually given by a simulation semantics, which is based on events (changes in the values on wires and registers). As remarked by Gordon [2], Verilog is the most used HDL language (in 1995 Verilog was getting 66 percent of business with 25,000 designers). Even though Verilog is widely used, usual presentation of the simulation semantics where equivalence between an abstract program and its implementation is given by observing the output of a simulation cycle, lack the necessary formality. Clearly it would be desirable to have a logical formal semantics to reason about properties of Verilog programs and to prove, among other things, equivalence between programs (specification and implementation). Some attempts to give formal semantics to Verilog are [**?**, 7, 3].

Verilog has some interesting features like concurrency, synchronism, shared variables, non-blocking assignments (scheduled assignments), timing controls, infinite computations, zero-time computations, etc., that makes it an interesting language to study. Some of these features are presented in this work through small examples and used in the Verilog specification of a STARI circuit.

STARI (Self-Timed At Receiver's Input) is a self-timed FIFO that interfaces a transmitter and a receiver operating at the same clock frequency but with a possible skew between their clock signals (see Figure 3). STARI was developed by Greenstreet [4] in 1993 and it has been used as a case study in compositional verification [**?**, **?**].

This report is organised as follows: Section 2 provides an overview of Verilog as well as many small illustrating examples whereas Section 3 presents STARI and its Verilog description (specification).

2 Overview of Verilog

Verilog is a Hardware Description Language (HDL) widely used to model the structure and behaviour of digital systems ranging from simple hardware building blocks to complete systems. In this section we will introduce Verilog briefly, discussing some semantical problems through examples. This section is based mainly on [2] (see also [7], [3], [5], [6] and [1] as well as http://www.cl.cam.ac.uk:80/users/mjcg/Verilog/ for some references about Verilog).

A specification in Verilog consists of a *main module*, which specifies a closed system containing both test data and hardware models, consisting of zero or more modules (or blocks). The specification can be *behavioural* or *structural* (or a combination of both). A behavioural specification defines the behaviour of a module using programming language constructs while the structural one expresses a module as a hierarchical interconnection of submodules. The modules have input and output *ports*, and events on the input ports cause events on the output ports. Verilog is the most used HDL (25,000 designers (1995), 5,000 each year in addition and 66% of business as pointed by [2, page 2]) that shows its importance to the industry. Its relevance is not only practical but also theoretical as it has some interesting features like for example:

- Normal programming language constructs (procedural assignments, loops, etc)
- Shared variables
- Synchronisation (Timing control: @(v), etc)
- Concurrency (modules, initial-statements, etc)
- Real-time constructs (Delay, Timing controls)
- Zero-duration computations
- Zero delays
- Non-blocking assignment ($v <= x$)
- Continuous assignments (wires)

The non-blocking assignment, in contrast with the procedural(or blocking) assignment, does not "block" the execution of the process when the statement is reached, i.e. the control of the program continues with the next enabled statement and the non-blocking assignment is done after all the active events have been executed (see Example 2.1.4 and Section 2.2 for an explanation of the usual semantics of Verilog).

2.1 Some simple examples

We present here a few simple examples to show some features of Verilog. The first three examples can be found in [2] while the others where written by one of the authors just to test the behaviour of some programming constructs. All the examples were tested using a free-ware version of the Veriwell/SPARC Simulator (Copyright (c) 1994 Wellspring Solutions, Inc.).

2.1.1 Example

The following is a very simple example showing a behavioural specification of a `NAND` gate.

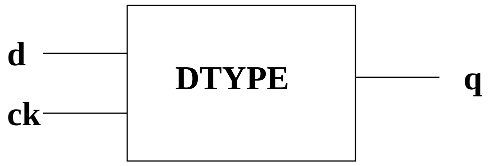

Figure 1: An edge-triggered Dtype register.

```
module NAND (i1,i2,o);
  input i1,i2; output o;

  assign o = ~(i1 & i2);
endmodule
```

The ports `i1,i2` and `o` are *wires*, as opposed to *procedural variables*, have no storage capacity: they can be continuously driven or left unconnected. The symbols `~` and `&` denote negation and conjunction respectively. The *continuous assignment* `assign o = ~(i1 & i2)` is activated whenever a change on `i1` or `i2` occurs, and in this case the expression `~(i1 & i2)` is evaluated and the result is immediately propagated to the left hand side variable.

2.1.2 Example

The following example shows an edge-triggered Dtype register (see Figure 1).

Whenever there is a positive edge on the input `ck` (i.e. a change from 0 to 1) the value of the input `d` is copied to the output `q`. A behavioural specification is

```
module DTYPE (ck,d,q);
  input ck,d; output q; reg q;

  always @(posedge ck) q = d;

endmodule
```

Here `ck` and `d` are wires and `q` is a *register*. Registers are variables that stores values (just like variables in imperative programming languages). The statement `always @(posedge ck) q =d` makes the timing control `@(posedge ck) q =d` be repeated forever. This timing control waits for a positive edge on `ck` and in this case the procedural assignment `q = d` is executed. The circuit shown in Figure 2 is an implementation of a Dtype register and it can easily be written in Verilog:

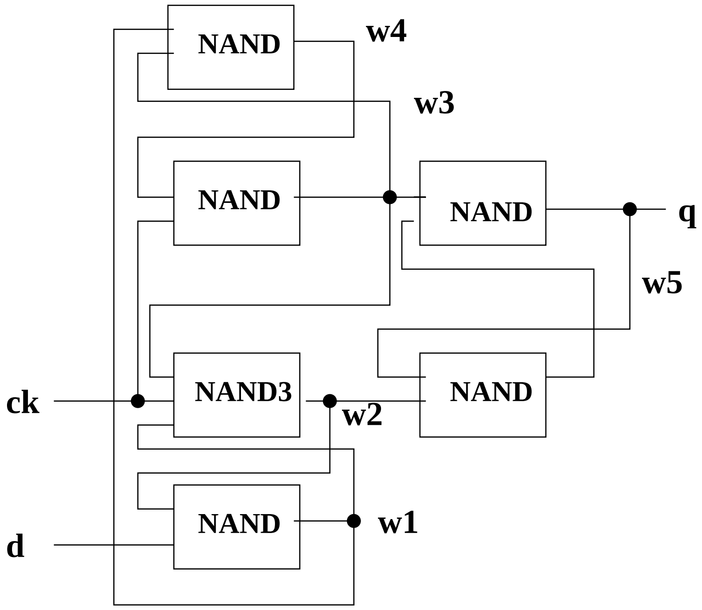

Figure 2: An implementation of a Dtype.

```
module DTYPE_IMP (ck,d,q);
  input ck,d; output q; wire w1,w2,w3,w4,w5;

  NAND  M1(w2,d,w1);
  NAND3 M2(w3,ck,w1,w2);
  NAND  M3(w4,ck,w3);
  NAND  M4(w1,w3,w4);
  NAND  M5(w3,w5,q);
  NAND  M6(q,w2,w5);

endmodule
```

where `NAND3` is the following three-input NAND-gate:

```
module NAND3 (i1,i2,i3,o);
  input i1,i2,i3; output o;

  assign o = ~(i1 & i2 & i3);

endmodule
```

The module `DTYPE_IMP` shows two important characteristics of Verilog:

- Concurrency. All the `NAND` and the `NAND3` modules are executed in parallel in a separate thread of computation, where shared variables can lead to non-determinism (see Example 2.1.3)
- Instantiation of modules: the module `NAND` is instantiated five times here (distinguished by the name Mi) and in this case each instantiation acts as a different module

Verilog is used not only to specify hardware components and behaviours but also to specify test data. The module `DTYPE_TEST_DATA` specifies some test signals:

```
module DTYPE_TEST_DATA (ck,d);
  output ck,d; reg ck,d;

  initial begin
        ck = 0; // time 0
    #5  d  = 1; // time 5
    #5  ck = 1; // posedge ck at time 10
    #10 ck = 0; // negedge ck at time 20
    #5  d  = 0; // time 25
    #5  ck = 1; // posedge ck at time 30
    #5  d  = 1; // time 35
  end

endmodule
```

where `#5` is a `delay` statement, and for instance `#5 d = 1` means that the procedural assignment `d = 1` will be done after 5 simulation time units. Everything following \\ on a line is a comment. The `initial` block means that the process is executed exactly once at the beginning of the simulation time.

The following module compares the behavioural and structural specifications of the DTYPE

```
module DTYPE_TEST ();
  wire ck,d,q1,q2;

  DTYPE_TEST_DATA M1 (ck,d);
  DTYPE           M2 (ck,d,q1);
  DTYPE_IMP       M3 (ck,d,q2);

  initial
    $monitor ("Time = %0d, ck = %b, d = %b, q1 = %b, q2 = %b",
                       $time, ck, d, q1, q2);
```

```
endmodule
```

where `$monitor` is a directive to the simulator to print out the values on the wires whenever they change and `$time` is a system variable that prints out the simulation time (advanced by the `delay` (#) statement.

The output of the simulation is

```
Time = 0,  ck = 0, d = x, q1 = x, q2 = x
Time = 5,  ck = 0, d = 1, q1 = x, q2 = x
Time = 10, ck = 1, d = 1, q1 = 1, q2 = 1
Time = 20, ck = 0, d = 1, q1 = 1, q2 = 1
Time = 25, ck = 0, d = 0, q1 = 1, q2 = 1
Time = 30, ck = 1, d = 0, q1 = 0, q2 = 0
Time = 35, ck = 1, d = 1, q1 = 0, q2 = 0
```

This test shows that `DTYPE` and `DTYPE_IMP` are equivalent (at least with the test data provided) and also illustrates a theoretical (more specifically, semantical) challenge posed by Verilog: it would be desirable that this equivalence were formally proved, and for that a formal semantics is needed.

2.1.3 Example

This example shows that Verilog programs (having shared variables) can lead to a *race condition*: the semantics does not uniquely determine the result. In Verilog there is a main module which might have *items* (or *blocks*) running in parallel in separate threads of computation. The items can be continuous assignments (see Example 2.1.1), instances of other modules (see module `DTYPE_IMP` in Example 2.1.2), `initial`-statements or `always`-statements (see module `DTYPE` in Example 2.1.2).

```
module INTERLEAVING ();
  integer x;

  initial
    begin
      x = 0;
      x = x+1;
    end
```

```
  initial x = 2;

endmodule
```

We have here two initial blocks running in parallel whose result can be `x=1` or `x=2` depending on which `initial` block is first executed, or even `x=3` if the second initial block is executed after the `x = 0` assignment of the first block, but before the execution of `x = x+1`.

2.1.4 Example

We present here an example of *non-blocking assignment* and blocking assignment with *zero delay*. Consider the following Verilog program:

```
module control;

reg [7:0]  x, y, z;

initial
  $display (" Time      x  Statement");

initial
begin
  x = #0 1;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      1");

  x <= #0 x + 1;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      2");

  x = 5;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      3");

  #1;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      4");

end

endmodule
```

We have two `initial` blocks running in parallel. The first one just writes the first line of the output[1] and in the second block we have four statements: a blocking assignment with zero delay, a non-blocking assignment with zero delay, a simple blocking assignment without delay and a `delay` statement. The `$write` command just output a string (between quotations marks) and the value of some variables (separated by commas) in the screen. The `$display` command is like `$write` but produce a new line. In the second block, the expression on the right hand side of the assignment `x= #0 1` is evaluated (just gives "1") and "blocks" the statement for 0 time units. After that, the second statement is executed: the expression `x + 1` is evaluated (the result is 2) but the result is not copied at this moment, it is enqueued for 0 time units and the assignment is done just at the end of the current slot time, i.e. after all the active and inactive events were executed (see the simulation semantics in Section 2.2). As the process is not "blocked" it can continue the execution with the next instruction (`x = 5`) and finally with the `delay` statement that advances the time in one time unit and the final value of `x` is displayed. The output is the following:

```
Time       x  Statement
   0       1      1
   0       1      2
   0       5      3
   1       2      4
```

It is worth mentioning that a blocking assignment with delays can have the following two forms: $\#n\ v = e$ and $v = \#n\ e$. In the first case the process is blocked for n time units, then the expression e is evaluated and its value is copied to v whereas in the second case the expression e is evaluated in the current slot time and the value is copied to v after n time units. In this example we use zero delay in the non-blocking assignment but the general form of the sentence is $v <= \#n\ e$ with $n \geq 0$. The important aspect to remember is that the expression (in the right hand side) of a non-blocking assignment is evaluated in the current simulation time, but the assignment is done after all the active and inactive events scheduled for time slot n.

2.2 Scheduling Semantics

The "normal" semantics of Verilog is based on the scheduling of events (changes in the values of wires and registers), that is, it is based on a simulation algorithm. This simulation (event or scheduling) semantics is explained in the Verilog IEEE Standard Report 1364 [5, Section 5]. According to this report, the simulation algorithm is based on a stratified queue of events segmented in the following five regions:

[1] Even though the execution of the `initial` blocks is non-deterministic, the simulator first chooses one sequence of execution and will act deterministically after that: it will execute statements in the whole process always in the same order, unless we change their order in the source code.

1. *Active events*: which occur at the current simulation time and can be processed in any order
2. *Inactive events*: which occur at the current simulation time but after all the active events are processed
3. *Non-blocking assignment update events*: which have been evaluated during some previous (or the same) simulation time, but must be assigned at this simulation time but after all the active and inactive events
4. *Monitor events*: which must be processed after all the above events
5. *Future inactive events* and *future non-blocking assignment update events*: which occur at some future simulation time.

Maybe one of the most important theoretical challenges posed by Verilog is its lack of a formal semantics that would allow, among other things, to prove some properties of programs or to develop verification conditions to ensure that two specifications are equivalent (as for example the behavioural and structural version of the DTYPE in Example 2.1.2). For other semantical challenges of Verilog refer to [2]).

There are not so many published papers attempting to give formal semantics to Verilog (as far as we know) and some of them are [**?**], [3] and [7].

2.3 Example

The following example shows the order between the blocking and non-blocking assignments with and without delays.

```
module control;

reg      [7:0]  x, y, z;

initial
  $display (" time      x       y    Statement");

always @(z)
begin

  x = x + 3;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       1");
```

```
  x = #1 20;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       2");

  x = #0 x + 1;
   $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
   $display ("       3");

  x = #0 x + 2;
   $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
   $display ("       5");

   x <= #0 x + 1;
   $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
   $display ("       6");

  x = x + 1;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       8");

  x = x+ 2;
   $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       10");

end

initial
begin
  z = 0; y = 0; x = 0;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       11");
  #1;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       12");

  z = 1;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       13");
  #1;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       14");

  z = 2;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       15");
```

```
  #1;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       16");

  z = 3;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       17");
  #1;
  $write ("%5.0f  %5.0f   %5.0f", $time, x, y);
  $display ("       18");

end

endmodule // control
```

There are three processes running in parallel. The first one is just to write the head of the output whereas the second one is an `always` block that is activated whenever a change on `z` is detected. The third block is an `initial` block that is activated just at the beginning of the simulation time and interacts with the other blocks until the last sentence is executed.

The output of the simulation is

```
 time      x       y    Statement

    0      0       0       11
    0      3       0      1
    1      3       0      12
    1      3       0      13
    1     20       0      2
    1     21       0      3
    1     23       0      5
    1     23       0      6
    1     24       0      8
    1     26       0      10
    2     24       0      14
    2     24       0      15
    2     27       0      1
    3     27       0      16
    3     27       0      17
    3     20       0      2
    3     21       0      3
    3     23       0      5
    3     23       0      6
    3     24       0      8
```

```
  3     26        0        10
  4     24        0        18
```

Notice that the interaction between the modules is done just when a timing control statement ($\#n$ or $@(z)$) is reached. This is not a restriction of the language but of the simulators. Even though the scheduling semantics presented in [5, Section 5] allows interleaving at any time between threads, some simulators restrict the interleaving to the cases mentioned above.

2.4 Example

The following is a very simple example of an infinite computation: two processes running in parallel activating each other (3 and 4). The other processes are just meant to print the head of the output (1), the values of the variables `CP1` and `CP2` (2) and to give initial values to these variables (5).

```
module loop;

reg      [7:0]   CP1, CP2;

initial                                     // 1
  $display ("  Time    CP1     CP2");

always @(CP1 or CP2)                        // 2
begin
  $write ("%5.0f  %5.0f   %5.0f",
  $time, CP1, CP2);
  $display (" ");
end

always @(CP2) CP1 = CP1 + 1;                // 3

always @(CP1) CP2 = CP2 + 1;                // 4

initial                                     // 5
  begin
    #1;
    CP1 = 1;
    CP2 = 0;
    #1;
    CP1 = 2;
    #1;
  end
```

```
endmodule
```

The output is

```
  Time    CP1     CP2
     1      1       0
     1      2       1
     1      3       2
     1      4       3
     1      5       4
     1      6       5
     1      7       6
     1      8       7
     1      9       8
     1     10       9
     1     11      10
     1     12      11
     1     13      12
     1     14      13
     1     15      14
   ...    ...     ...
```

2.5 Example

```
module control;

reg      [7:0]  x;

initial
  $display ("  Time     X   Statement");

initial
begin
  x = 0;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      6");
end

initial
begin
  #2;
```

```
  x = x + 5;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      1");

  x <= #1 x + 5;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      2");

  @(x) x = x + 1;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      3");

  x = x + 2;
  $write ("%5.0f  %5.0f", $time, x);
  $display ("      4");

  #1;

  $write ("%5.0f  %5.0f", $time, x);
  $display ("      5");

end

endmodule
```

This example shows that a non-blocking assignment (`x <= #1 x + 5`) can activate a timing control (`@(x) x = x + 1`) waiting for changes on a variable. In particular, the assignment of the non-blocking assignment is done at the end of the third time unit (there is a delay of two times units and the non-blocking assignment has a delay of one time unit). Hence, the output is:

```
  Time      X   Statement
    0       0      6
    2       5      1
    2       5      2
    3      11      3
    3      13      4
    4      13      5
```

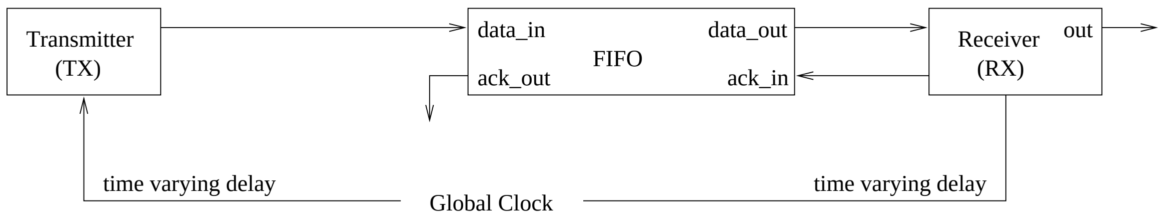

Figure 3: A STARI interface.

3 STARI

3.1 A Description of STARI

Many digital circuits are designed to work in a synchronous way, which means that they rely on a global common timing mechanism that is shared by all components of the circuit. Synchronous circuits like these are simpler to design but they suffer from clock skew problems. Clock skews are differences in the phase of the clock signal received by each component caused by the different lenghts and properties of the clock signal distribution lines. To counteract the clock skew problems, components can be made to communicate through self-timed interfaces, where essentially the data transmitted regulates the operations of the receiver, and acknowledgement signals from the receiver regulate the operations of the transmitter. Thus, no synchronisation is required but, instead, round-trip latency problems limit severely the bandwidth of the communication channel.

STARI is a communications technique that combines synchronous and self-timed design methods, and which can tolerate clock skews greater than the clock period while being free from the bandwidth restrictions of self-timed interfaces.

In STARI, as depicted in Figure 3 (from [**?**]), the transmitter sends a new data value each time it receives a varyingly-delayed clock signal. In other words, the transmitter needs no flow control from the FIFO. The FIFO then transmits the data value to the other end where it is read by the receiver each time it receives its own varyingly-delayed clock signal. The receiver is also responsible for producing the appropriate acknowdedgement after it has copied the value to its output (*out*).

STARI is designed to tolerate skews in the clock signals which are several times greater than the clock period. When this occurs, it can appear as either the tranmitter or the receiver's clock signal is working faster than the other. Because of the storing capacity of STARI, it accumulates data when the transmitter is working faster, and delivers stored data when the it is the receiver's turn to work faster.

In what follows, we present some important details on the implementation of the STARI circuit.

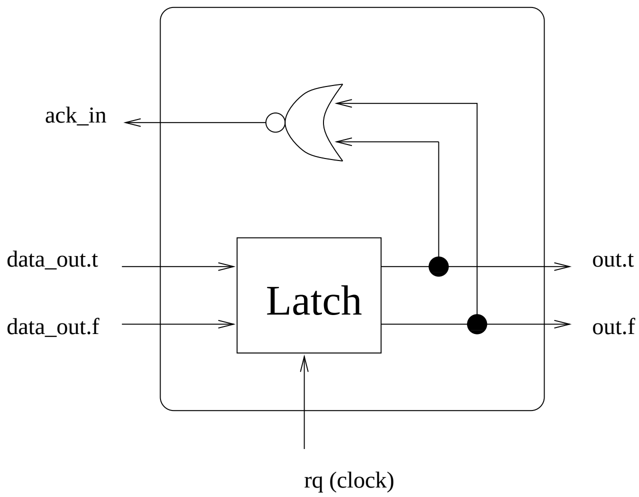

Figure 4: The receiver.

3.1.1 Dual-rail coding

The STARI circuit uses dual-rail coding [**?**] to represent boolean values and to allow successive values to be distinguished. Each boolean signal `x` is encoded with two signals, `x.t` and `x.f`, as illustrated below:

x.t	x.f	x
0	0	E (empty)
0	1	F (false)
1	0	T (true)
1	1	illegal

Table 1: Dual-rail coding.

We will say that **T** and **F** are "valid" values, while **E**, the empty value, is a "separator".

Also notice that STARI does not use dual-raid coding for its acknowledgement lines.

3.1.2 The Transmitter and the Receiver

The transmitter produces a new data value to the *data_in* line whenever it receives a clock tick. The trasmitter may or may not be responsible for the generation of these values, but it must always be able to produce its output on time. Moreover, in operation, it alternates between sending empty and valid values.

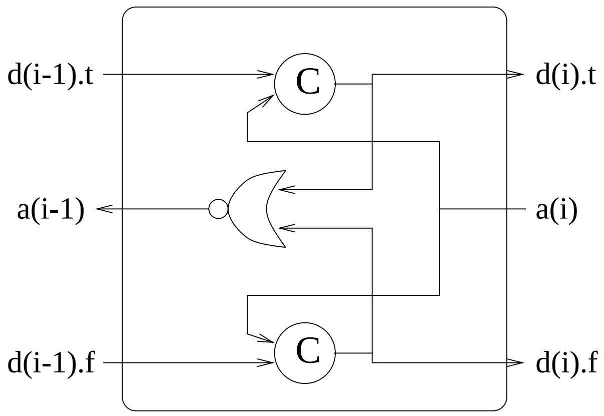

Figure 5: STARI's i-th stage.

The receiver, shown in Figure 4, copies the value output by STARI (*data_out*) into its own output (*out*) whenever it receives a clock tick. Then, it sends an acknowledgement to STARI (notice that this does not occur immediately as there is a delay associated with the NOR gate).

3.1.3 The Stages

The FIFO in STARI is composed by a succession of aligned identical stages. The *i-th* stage is shown in Figure 5. Refering back to Figure 3 and assuming a STARI with n stages, we intend d_0, d_n, a_o and a_n to be alternative names for `data_in`, `data_out`, `ack_out` and `ack_in` respectively.

Each stage contains two Muller C-elements and an NOR gate. Each Muller C-element manages one rail of the dual-rail coding, copying the input to the output whenever the previously output value has been acknowledged. Table 2 describes the behavior of this component in each possible situation.

Input 1	Input 2	Output
0	0	0
0	1	unchanged
1	0	unchanged
1	1	1

Table 2: Muller C-element functionality.

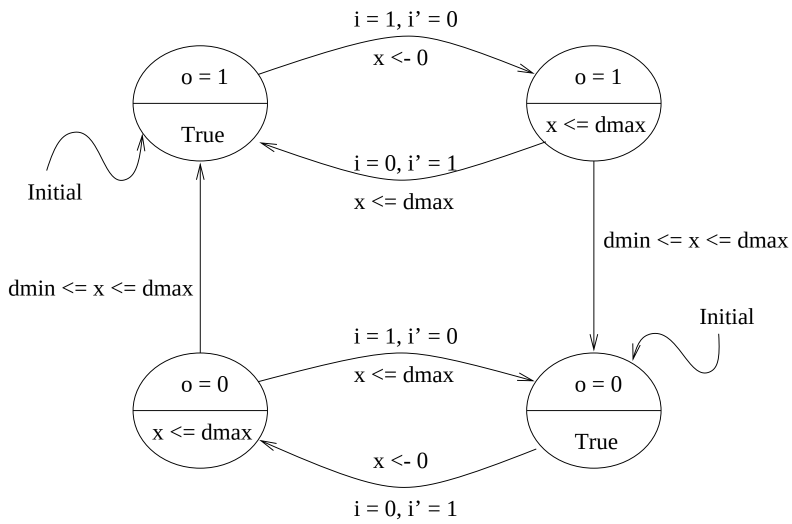

Figure 6: Timed process for an inertial delay.

3.1.4 Component Delays

Each component used in either the stages or the receiver has, inevitably, a varying minimum response time. Any change in its input must last at least some time to have any effect on the component's output. Moreover, there is usually a diffuse zone in which the component may or may not react.

Basically, there are only two components whose associated response times are of importance to the correct operation of STARI: the Muller C-elements and the NOR gates.

Consider a NOR gate. In our model, every NOR gate is associated with two constants: $\delta_{nor,min}$ and $\delta_{nor,max}$. Any change in any of the gate's inputs that lasts less than $\delta_{nor,min}$ time units, will not appear reflected in its output. Simetrically, any change lasting at least $\delta_{nor,max}$ will definitely be accompanied by a change in the output. For any input change lasting between $\delta_{nor,min}$ and $\delta_{nor,max}$, the output may or may not changed.

Analogously, our model associates every Muller C-element with constants $\delta_{c,min}$ and $\delta_{c,max}$.

Figure 6 (from [**?**]) illustrates a *timed process* that reflects the kind of behavior expected from an inertial delay buffer with non-deterministic delay between δ_{min} and δ_{max}. In the figure, x is a clock that is reset every time a transition labelled $x \leftarrow 0$ is taken. i is the input variable, whose changes are copied to o after some delay.

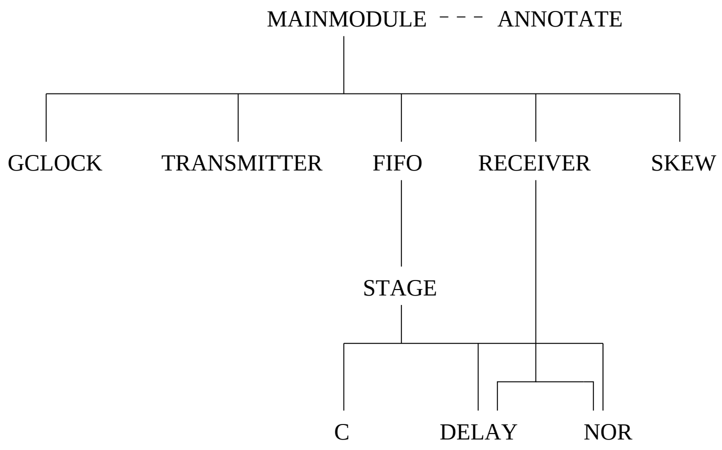

Figure 7: Module Structure of STARI.

3.1.5 Clock Skews

We model clock skews quite differently from the way we model component delays. First, we assume that two clock ticks can only arrive at a timed component (e.g. the transmitter) in the same order at which they were issued by the global clock. This means that if a clock tick is delayed, subsequent ticks can only arrive later. Second, we assume that every tick eventually reaches the end of the transmission line, meaning that the line never loses a tick. Third, we recognise the importance of being able to model skews several times larger than the clock period. And finally, we demand from the distribution line the possibility of controlling the minimum separation between consecutive ticks (the reason for this will be given later).

In [**?**], STARI is analysed using the verification tool Cospan [**?**, **?**]. There, clock skews are modelled just like component delays, thus failing to take into account some of these considerations and making it impossible to analyse skews larger than half a clock period. Certainly, in the presence of such small skews, there is no real justification for the utilization of a STARI circuit.

3.2 An Structural Specification of STARI

We present here a structural specification of STARI in Verilog. As explained in Section 2, a structural specification expresses a module as a hierarchical interconnection of submodules. Figure 7 shows the module structure of the Verilog specification of STARI.

Below we present the Verilog code of each module together with an explanation. In Appendix A we present the whole Verilog code of the STARI.

3.2.1 Module ANNOTATE

This module sets the parameters of the program. The definition of the parameters is done in each module and this module just instantiates each parameter with the following syntax:

$$\langle name - of - the - module\rangle . \langle name - of - the - instantiation\rangle . \langle parameter\rangle$$

It is worth mentioning that changes to the minimum and maximum delays as well as to the minimum distance between consecutive receiver clock ticks will change the behaviour of the STARI. In particular is it possible to check some properties and the "tolerance" of the circuit by modifying these values. The `init_load` parameter indicate the number of initial loaded values on the FIFO. As pointed out by Greenstreet [4] the STARI should begin working with half of the FIFO loaded with input values.

```
module annotate;
defparam
        mainmodule.sTr.smin      = 5,   // Minimum delay for the transmitter
        mainmodule.sTr.smax      = 100, // Maximum delay for the transmitter
        mainmodule.sTr.mdist     = 4,   // Minimum distance between consec.
                                        // transmitter clock ticks.
        mainmodule.sRc.smin      = 0,   // Minimum delay for the receiver
        mainmodule.sRc.smax      = 0,   // Maximum delay for the receiver
        mainmodule.sRc.mdist     = 6,   // Minimum distance between consec.
                                        // receiver clock ticks.

        mainmodule.Rc.Dmin_nor   = 3,   // Minimum delay for NOR component.
        mainmodule.Rc.Dmax_nor   = 3,   // Maximum delay for NOR component.

        mainmodule.GQ.half_pi    = 50,  // half clock period
        mainmodule.GQ.sim_pi     = 20,  // number of simulations periods

        mainmodule.Tr.init_load = 6;    // no. of initially loaded values

endmodule
```

3.2.2 Module MAINMODULE

The main module represents the whole system with its components: Global clock, FIFO, Transmitter and Receiver with their correspondent Skews. It is easy to see that the parameters of each module match the inputs and outputs of the corresponding component as explained in the description of STARI.

```
module mainmodule;

wire         TRclock,            // Transmitter clock
             RCclock,            // Receiver clock
             Clock;              // Global clock (wire)

wire   [1:0] data_in,            // input data to the FIFO
             data_out;           // output data from the FIFO
wire         ack_in,             // input acknowledgement (FIFO)
             ack_out;            // output acknowledgement (FIFO)


GCLOCK       GQ  (Clock);               // Global clock

SKEW         sTr (Clock, TRclock);      //skew for the transmitter
TRANSMITTER  Tr (TRclock, ack_out, data_in);

FIFO         FF (data_in, ack_in, data_out, ack_out);

SKEW         sRc (Clock, RCclock);      //skew for the receiver
RECEIVER     Rc (RCclock, data_out, ack_in);

endmodule //mainmodule
```

3.2.3 Module GCLOCK

This module specifies the global clock of the system. It has no input and it just outputs the value of the clock (0 or 1) every half period.

```
module GCLOCK (q);
output       q;

reg          q;

parameter    half_pi = 50,   // half clock period
             sim_pi  = 20;   // number of simulations periods

integer      i;

initial
  begin
  q = 1;
```

```
  for (i=0; i<sim_pi; i=i+1)
    begin
      # half_pi;
      q = 0;
      # half_pi;
      q = 1;
    end
  $finish;
end

endmodule // GCLOCK
```

3.2.4 Module TRANSMITTER

The purpose of this module is twofold. First, during the initialisation stage, it feeds values into the FIFO so as to leave it half full. And then, it is responsible for the insertion of alternating values into the FIFO each time it receives a clock tick.

```
module TRANSMITTER (TRclock, ack_in, data_out);
input         TRclock, ack_in;
output        data_out;

reg [1:0]     data_out;

parameter     [1:0] Empty     = 2'b00, // Empty value: 00
                    False     = 2'b01, // False value: 01
                    True      = 2'b10; // True  value: 10

parameter     init_load = 0;           // How many values are loaded
                                       // initially into the STARI

reg           flag;
integer       i;

// Initially, the STARI is loaded as fast as possible, but waiting for
// the acknowledgement from the first STARI stage.

initial begin
  flag = 0;
  data_out = Empty;
  for (i=0; i<=init_load; i=i+1)
    begin
      @(ack_in)
```

```
        case (ack_in && data_out == Empty)
          1 : begin
                if (flag == 0) data_out = True;       //  generate new data
                else  data_out = False;
                flag = ~flag;
              end
          0 : if (~ack_in && (data_out != Empty))
                data_out = Empty;         // sends an Emtpy value after a
                                          // True or False value
              else $display("Error");
        endcase
    end
end

always @(negedge TRclock)   // the transmitter is allowed to send data
 begin
   case (ack_in && data_out == Empty)
      1 : begin
            if (flag == 0) data_out = True;       //  generate new data
            else  data_out = False;
            flag = ~flag;
          end
      0 : if (~ack_in && (data_out != Empty))
            data_out = Empty;         //sends an Empty after a True|False
          else $display("time = %0d, ** STARI overflows **", $time);
   endcase
 end

endmodule //TRANSMITTER
```

The Transmitter module has an `initial` and an `always` submodule. The `initial` submodule is just to guarantee that the FIFO will have some data before the receiver can try to take data from it. That is, the Transmitter sends as many data as indicated by the `init_load` parameter. Ii is common practice to initialise the FIFO to be half full.

The `always` submodule specifies the behaviour of the transmitter: It sends data just when a negedge of the clock is detected (a change from 1 to 0). In our implementation we send sequences of False (01), Empty (00), True (10), Empty (00) data. The `flag` is used just to alternate valid values.

3.2.5 Module RECEIVER

At every clock tick, the receiver copies the value present at the last stage of STARI into its `sample` variable and then, after some delay, issues an acknowledgement to the same stage.

```
module RECEIVER (RCclock, data_in, ack_out);
input         RCclock;
input  [1:0]  data_in;
output        ack_out;

parameter     [1:0] Empty = 0;         // Empty value: 00
parameter     Dmin_nor = 0,            // Minimum delay for NOR component
              Dmax_nor = 0;            // Maximum delay for NOR component


reg [1:0]     sample;                  // where the input data is stored
wire          o;                       // output of the NOR component

initial begin
  sample = Empty;
end

always @(negedge RCclock)   // the receiver is allowed to receive data
  begin
    if (sample == data_in)
      $display ("time = %0d, ** STARI underflows **", $time);
    sample = data_in;
  end

NOR   Nrc (sample[1], sample[0], o);         // to produce the
DELAY Drc (o, ack_out, Dmin_nor, Dmax_nor);  // acknowledgement

endmodule //RECEIVER
```

The `initial` block of the Receiver module just initialise the variable `sample` that is the output of the STARI whereas the `always` block just copy the received data (sent by the transmitter through the FIFO) output, control the underflow and produce an acknowledgement indicating that has received the data.

Notice that in section 3.1, variable `sample` was called `out`.

3.2.6 Module FIFO

This module specifies a FIFO with ten stages interconnected. Verilog does not provide any support for parameterised structural specifications of the kind required to implement a FIFO with n stages.

```
module FIFO (d_in, ack_in, d_out, ack_out);
input  [1:0]  d_in;
input         ack_in;
output [1:0]  d_out;
output        ack_out;


wire [0: 10]  t, f, ack;        // wire arrays to interconnect
                                // 10 STARI stages


STAGE S1  (d_in[1], d_in[0], ack[1], t[1],     f[1],     ack_out);
STAGE S2  (t[1],    f[1],    ack[2], t[2],     f[2],     ack[1]);
STAGE S3  (t[2],    f[2],    ack[3], t[3],     f[3],     ack[2]);
STAGE S4  (t[3],    f[3],    ack[4], t[4],     f[4],     ack[3]);
STAGE S5  (t[4],    f[4],    ack[5], t[5],     f[5],     ack[4]);
STAGE S6  (t[5],    f[5],    ack[6], t[6],     f[6],     ack[5]);
STAGE S7  (t[6],    f[6],    ack[7], t[7],     f[7],     ack[6]);
STAGE S8  (t[7],    f[7],    ack[8], t[8],     f[8],     ack[7]);
STAGE S9  (t[8],    f[8],    ack[9], t[9],     f[9],     ack[8]);
STAGE S10 (t[9],    f[9],    ack_in, d_out[1], d_out[0], ack[9]);

endmodule //FIFO
```

3.2.7 Module STAGE

Each stage in a STARI is an instance of this module. It just represents the structure as defined in section 3.1.3.

```
module STAGE (t_i, f_i, ack_in, t_o, f_o, ack_out);
input         t_i, f_i, ack_in;
output        t_o, f_o, ack_out;

wire          t_o1, t_o2, t_o3;     // Internal wires
```

```
parameter     Dmin_nor = 03,        // Minimum delay for NOR component
              Dmax_nor = 03,        // Maximum delay for NOR component
              Dmin_c   = 03,        // Minimum delay for C   component
              Dmax_c   = 03;        // Maximum delay for C   component


C             Cs1(t_i, ack_in, t_o1);
DELAY         Ds1(t_o1, t_o, Dmin_c, Dmax_c);

NOR           Ns1(t_o, f_o, t_o2);
DELAY         Ds2(t_o2, ack_out, Dmin_nor, Dmax_nor);

C             Cs2(f_i, ack_in, t_o3);
DELAY         Ds3(t_o3, f_o, Dmin_c, Dmax_c);

endmodule //STAGE
```

3.2.8 Module C

```
module C (i1, i2, o);
input         i1, i2;
output        o;

reg o;

initial o = 0;

always @(i1 or i2) if (i1 == i2) o = i1;

endmodule // C
```

The C module specifies the following behaviour of the Muller C-element: If its two inputs are equal, then their value is copied to the output. Otherwise, the output remains unchanged.

3.2.9 Module NOR

```
module NOR (i1, i2, o);
input         i1, i2;
output        o;

assign o = ~(i1 | i2);
```

```
endmodule // NOR
```

This module just specifies the NOR gate between the inputs (that are wires). Notice that the NOR gate is implemented with a continuous assignment.

3.2.10 Module DELAY

This module implements the timed process of Figure 6 (page 18). It copies to output wire `o` every value that remains present in wire `i` long enough. If a value remains in `i` less than `dmin` time units, it will not be copied. If it lasts more than `dmax` time units, it will be copied. Any value lasting more than `dmin` and less than `dmax` will or will not be copied, non-deterministically. This non-determinism is simulated with a very simple pseudo-random number generator.

Each time input `i` changes, a local clock is reset in this module. This clock runs at the same frequency as the simulation clock. If its value reaches `dmin`, and the random variable holds an even number, then the input is copied to the output. If the clock reaches `dmax`, then the input is certainly copied to the output.

```
module DELAY (i, o, dmin, dmax);
input          i;                       // input
input [15:0]   dmin,                    // minimum delay
               dmax;                    // maximum delay

output         o;                       // output
reg            o;

parameter      a = 13,                  // parameters of
               b = 7,                   // the
               n = 1013;                // random function

integer        random;
integer        local_clock;             // counts the time a value has
                                        // remained at the input.

initial begin
  local_clock = 0;
  random = 0;
end


// Description of DELAY's behavior:
```

```
// 1. If a value remains at the input wire i for less than "dmin" time units,
//    then it is not copied to the output wire o.
// 2. If a value remains at the input wire i between "dmin" and "dmax"
//    time units, then it might be copied to the output wire o.
// 3. If a value remains at the input wire i for at least dmax time units,
//    then it is copied to the output wire o.

// The nondeterminism in (2) above is simulated using a simple
// pseudo-random number generator.

always begin
  random = (a * random + b) % n; // for the non-determinism
  if (local_clock < dmax)
    if ((local_clock >= dmin) && (random % 2 == 0)) o =i;
  else
    o = i;
  #1;
  local_clock = local_clock + 1;
end

always @(i) local_clock = 0;  // reset the clock every time the
                              // input changes
endmodule  //DELAY
```

3.2.11 Module SKEW

Module SKEW applies a random variable delay (between `smin` and `smax`) to a signal while preserving the order of changes and guaranteeing a minimum distance (`mdist`) between changes.

The implementation relies on the nonblocking assignment feature of Verilog. Every time a change is detected in wire `i`, a random delay, falling between `smin` and `smax`, is calculated and the resulting absolute time is compared against the previous change deadline so as to guarantee a minimum separation between them.

The actual Veriwell implementation required a duplicate definition of this module (under different names) since two instances where sharing local variables, thus preventing the correct operation of each module.

```
module SKEW (i, o);
input         i;
output        o;
reg           o;
```

```
parameter     smin  = 0,   // Minimum skew
              smax  = 0,   // Maximum skew
              mdist = 0,   // Minimum distance between consec. ticks

              a = 101,       // parameters of
              b = 7,         //     the
              n = 1013;      // random function

integer       random,        // random number
              old_deadline,  // scheduled last (old) change of the output
              delay,         // delay
              time1;         // used to convert $time into an integer value

initial begin
  random = 0;         // Initial value of the random variable
  old_deadline = -1;  // Initial value of the last (old) deadline
end

always @(i) begin
   random = (a * random + b) % n;       // to determine variable skew
   time1 = $time;                       // store the current sim. time

// compute the new delay (dmin <= delay <= dmax)
   delay = smin + ((random * (smax - smin))/(n-1));

// verify that it preserves minimum separation between changes to o.
   if (delay < old_deadline-time1+mdist)
     delay = old_deadline - time1 + mdist;

// schedule the output change to occur delay time units later
   o <= #delay i;

// remember this last deadline
   old_deadline = time1 + delay;
end

endmodule  // skew
```

4 Discussion

We have given an overview of the Verilog hardware description language together with some examples. Verilog has some interesting features like concurrency, synchronism, shared variables, non-blocking assignments (scheduled assignments), timing controls, infinite computations, zero-time computations, etc., that makes it an interesting language to study. Some of these features were presented in this work through small examples and used in the Verilog specification of a STARI circuit.

STARI (Self-Timed At Receiver's Input) is a self-timed FIFO that interfaces a transmitter and a receiver operating at the same clock frequency but with a possible skew between their clock signals (see Figure 3). STARI was developed by Greenstreet [4] in 1993 and it has been used as a case study in compositional verification [**?**, **?**]. The formal semantics of Verilog introduced in [**?**] can be used in the Verilog specification of the STARI with the aim of facilitate the definition and proofs of some properties, complementing the work in [**?**].

The aim of this work was to gain experience in the use of Verilog and in the basic properties of the STARI. The former led to the formalisation of the semantics of the language [**?**], while the latter was used in the verification of the circuit using assumption-commitment proof rules [**?**].

References

[1] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In *Hybrid Systems III*, number 1066 in LNCS, pages 220–231. Springer-Verlag, 1996.

[2] P. Giambiagi and Q. Xu. Assumption-Commitment Verification of STARI. To appear as Technical Report at UNU/IIST in February, 1998.

[3] Ulrich Golze. *VLSI Chip Design with the Hardware Description Language VERILOG: An Introduction Based on a Large RISC Processor Design*. Springer Verlag, 1996.

[4] Mike Gordon. The Semantic Challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, pages 136–145, June 1995. Revised version (April 11, 1996) of the paper published in the proceeding.

[5] Mike Gordon. Synthesizable Verilog: sintax and semantics. Available at http://www.cl.cam.ac.uk/users/mjcg/Verilog/V/V.html, June 1997.

[6] Mark Greenstreet. *STARI: A Technique for High-bandwidth Communication*. PhD thesis, Princeton, 1993. Available at http://ncstrl.cs.princeton.edu/Dienst/Repository/2.0/Body/ncstrl.princetonfTR-394-92/postscript.

[7] IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE std 1364-1995*, 1995.

[8] C. Leung and M. R. Greenstreet. A Simple Proof Checker for Timing Verification. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 294–305, November 1995.

[9] Rajeev Madhavan. Quick Reference for Verilog HDL. Automata Publishing Company, 1995. Available at ftp://ftp.siliconlogic.com:/pub/comp.lang.verilog/ref.tar.Z.

[10] Gordon Pace. Ph.D. thesis. Chapter 3 and 4, 1997.

[11] Gerardo Schneider and Xu Qiwen. Towards a Formal Semantics of Verilog using Duration Calculus. Technical report, UNU/IIST, P.O.Box 3058, Macau, February 1998.

[12] Ch. L. Seitz. *Introduction to VLSI Systems*, chapter System Timing. Addison Wesley, 1979.

[13] S. Tasiran and R. K. Brayton. STARI: A Case Study in Compositional and Hierarchical Timing Verification. In O. Grumberg, editor, *CAV'97: Computer-aided Verification*, number 1254 in LNCS. Springer-Verlag, 1997.

[14] Serdar Tasiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. Verifying Abstractions of Timed Systems. In *Proc. of CONCUR'96: 7th International Conference on Concurrency Theory*, number 1119 in Lecture Notes in Computer Science, pages 546–562. Springer-Verlag, August 1996.

A A Verilog specification of the STARI

```
//---------------------------------------------------------------------
//
// STARI.EXM
//
// Model of the STARI
//   with non-determinism and two delays: one for the gates and the
//   other one for the skew
//
// In this model we use different delay modules for modelling the delay
// of the gates and the skew of the global clock. This is to allow
// skews greater than a half of the clock period. We use two SKEW modules
// because Veriwell cannot distinguish the "delay" variable in the module
// (they should be different)
//
//
// Date: 19/1/97
//
//---------------------------------------------------------------------


//---------------------------------------------------------------------
//
// Parameter definition
//
//---------------------------------------------------------------------
module annotate;
```

```
defparam
        mainmodule.sTr.smin     = 5,   // Minimum delay for the transmitter
        mainmodule.sTr.smax     = 350, // Maximum delay for the transmitter
        mainmodule.sTr.mdist    = 4,   // Minimum distance between consec.
                                       // transmitter clock ticks.
        mainmodule.sRc.smin     = 0,   // Minimum delay for the receiver
        mainmodule.sRc.smax     = 56,  // Maximum delay for the receiver
        mainmodule.sRc.mdist    = 6,   // Minimum distance between consec.
                                       // receiver clock ticks.

        mainmodule.Rc.Dmin_nor  = 3,   // Minimum delay for NOR component.
        mainmodule.Rc.Dmax_nor  = 3,   // Maximum delay for NOR component.

        mainmodule.GQ.half_pi   = 50,  // half clock period
        mainmodule.GQ.sim_pi    = 20,  // number of simulations periods

        mainmodule.Tr.init_load = 6;   // no. of initially loaded values

// Delay parameters for every stage component should be modified from
// the definition of module STAGE

endmodule


//---------------------------------------------------------------------
//
// Main Module
//
//---------------------------------------------------------------------
module mainmodule;


wire          TRclock,             // Transmitter clock
              RCclock,             // Receiver clock
              Clock;               // Global clock (wire)

wire    [1:0] data_in,             // input data to the FIFO
              data_out;            // output data from the FIFO
wire          ack_in,              // input acknowledgement (FIFO)
              ack_out;             // output acknowledgement (FIFO)


GCLOCK        GQ  (Clock);                // Global clock

SKEW          sTr (Clock, TRclock);       //skew for the transmitter
```

```
TRANSMITTER   Tr (TRclock, ack_out, data_in);

FIFO          FF (data_in, ack_in, data_out, ack_out);

SKEW2         sRc (Clock, RCclock);       //skew for the receiver
RECEIVER      Rc (RCclock, data_out, ack_in);

endmodule //mainmodule

//-----------------------------------------------------------------
//
//  module GCLOCK (Global clock)
//
//-----------------------------------------------------------------
module GCLOCK (q);
output        q;

reg           q;

parameter     half_pi = 50,   // half clock period
              sim_pi  = 20;   // number of simulations periods

integer       i;

initial
  begin
  q = 1;
  for (i=0; i<sim_pi; i=i+1)
    begin
      # half_pi;
      q = 0;
      # half_pi;
      q = 1;
    end
  $finish;
end

endmodule // GCLOCK


//-----------------------------------------------------------------
//
// module TRANSMITTER
//
//-----------------------------------------------------------------
```

```
module TRANSMITTER (TRclock, ack_in, data_out);
input         TRclock, ack_in;
output        data_out;

reg [1:0]     data_out;

parameter     [1:0] Empty     = 2'b00, // Empty value: 00
                    False     = 2'b01, // False value: 01
                    True      = 2'b10; // True  value: 10

parameter     init_load = 0;           // How many values are loaded
                                       // initially into the STARI

reg           flag;
integer       i;

// Initially, the STARI is loaded as fast as possible, but waiting for
// the acknowledgement from the first STARI stage.

initial begin
  flag = 0;
  data_out = Empty;
  for (i=0; i<=init_load; i=i+1)
    begin
      @(ack_in)
        case (ack_in && data_out == Empty)
          1 : begin
                if (flag == 0) data_out = True;       //  generate new data
                else  data_out = False;
                flag = ~flag;
              end
          0 : if (~ack_in && (data_out != Empty))
                data_out = Empty;         // sends an Emtpy value after a
                                          // True or False value
              else $display("Error");
        endcase
    end
end

always @(negedge TRclock)   // the transmitter is allowed to send data
 begin
   case (ack_in && data_out == Empty)
      1 : begin
            if (flag == 0) data_out = True;       //  generate new data
            else  data_out = False;
```

```
          flag = ~flag;
        end
    0 : if (~ack_in && (data_out != Empty))
          data_out = Empty;         //sends an Empty after a True|False
        else $display("time = %0d, ** STARI overflows **", $time);
   endcase
 end

endmodule //TRANSMITTER


//--------------------------------------------------------------
//
// module RECEIVER
//
//--------------------------------------------------------------
module RECEIVER (RCclock, data_in, ack_out);
input         RCclock;
input  [1:0]  data_in;
output        ack_out;

parameter     [1:0] Empty = 0;         // Empty value: 00
parameter     Dmin_nor = 0,            // Minimum delay for NOR component
              Dmax_nor = 0;            // Maximum delay for NOR component


reg [1:0]     sample;                  // where the input data is stored
wire          o;                       // output of the NOR component

initial begin
  sample = Empty;
end

always @(negedge RCclock)   // the receiver is allowed to receive data
  begin
    if (sample == data_in)
      $display ("time = %0d, ** STARI underflows **", $time);
    sample = data_in;
  end

NOR   Nrc (sample[1], sample[0], o);         // to produce the
DELAY Drc (o, ack_out, Dmin_nor, Dmax_nor);  // acknowledgement

endmodule //RECEIVER
```

```
//-----------------------------------------------------------------
//
// module FIFO  //FIFO with 10 stages
//
//-----------------------------------------------------------------
module FIFO (d_in, ack_in, d_out, ack_out);
input  [1:0]  d_in;
input         ack_in;
output [1:0]  d_out;
output        ack_out;


wire [0: 10]  t, f, ack;        // wire arrays to interconnect
                                // 10 STARI stages


STAGE S1  (d_in[1], d_in[0], ack[1], t[1],     f[1],     ack_out);
STAGE S2  (t[1],    f[1],    ack[2], t[2],     f[2],     ack[1]);
STAGE S3  (t[2],    f[2],    ack[3], t[3],     f[3],     ack[2]);
STAGE S4  (t[3],    f[3],    ack[4], t[4],     f[4],     ack[3]);
STAGE S5  (t[4],    f[4],    ack[5], t[5],     f[5],     ack[4]);
STAGE S6  (t[5],    f[5],    ack[6], t[6],     f[6],     ack[5]);
STAGE S7  (t[6],    f[6],    ack[7], t[7],     f[7],     ack[6]);
STAGE S8  (t[7],    f[7],    ack[8], t[8],     f[8],     ack[7]);
STAGE S9  (t[8],    f[8],    ack[9], t[9],     f[9],     ack[8]);
STAGE S10 (t[9],    f[9],    ack_in, d_out[1], d_out[0], ack[9]);

endmodule //FIFO


//-----------------------------------------------------------------
//
// module STAGE // stage of the FIFO
//
//-----------------------------------------------------------------
module STAGE (t_i, f_i, ack_in, t_o, f_o, ack_out);
input         t_i, f_i, ack_in;
output        t_o, f_o, ack_out;

wire          t_o1, t_o2, t_o3;     // Internal wires

parameter     Dmin_nor = 03,        // Minimum delay for NOR component
              Dmax_nor = 03,        // Maximum delay for NOR component
              Dmin_c   = 03,        // Minimum delay for C   component
```

```
              Dmax_c   = 03;        // Maximum delay for C   component


C             Cs1(t_i, ack_in, t_o1);
DELAY         Ds1(t_o1, t_o, Dmin_c, Dmax_c);

NOR           Ns1(t_o, f_o, t_o2);
DELAY         Ds2(t_o2, ack_out, Dmin_nor, Dmax_nor);

C             Cs2(f_i, ack_in, t_o3);
DELAY         Ds3(t_o3, f_o, Dmin_c, Dmax_c);

endmodule //STAGE


//-------------------------------------------------------------
//
// module C //C component
//
//-------------------------------------------------------------
module C (i1, i2, o);
input         i1, i2;
output        o;

reg o;

initial o = 0;

always @(i1 or i2) if (i1 == i2) o = i1;

endmodule // C


//-------------------------------------------------------------
//
// module NOR  // NOR component
//
//--------------------------------------------------------------
module NOR (i1, i2, o);
input         i1, i2;
output        o;

assign o = ~(i1 | i2);

endmodule // NOR
```

```
//-------------------------------------------------------------
//
//  module DELAY (delay for NOR and C components)
//
//-------------------------------------------------------------
module DELAY (i, o, dmin, dmax);
input           i;                      // input
input [15:0]    dmin,                   // minimum delay
                dmax;                   // maximum delay

output          o;                      // output
reg             o;

parameter       a = 13,                 // parameters of
                b = 7,                  // the
                n = 1013;               // random function

integer         random;
integer         local_clock;            // counts the time a value has
                                        // remained at the input.

initial begin
  local_clock = 0;
  random = 0;
end


// Description of DELAY's behavior:
// 1. If a value remains at the input wire i for less than "dmin" time units,
//    then it is not copied to the output wire o.
// 2. If a value remains at the input wire i between "dmin" and "dmax"
//    time units, then it might be copied to the output wire o.
// 3. If a value remains at the input wire i for at least dmax time units,
//    then it is copied to the output wire o.

// The nondeterminism in (2) above is simulated using a simple
// pseudo-random number generator.

always begin
  random = (a * random + b) % n; // for the non-determinism
  if (local_clock < dmax)
    if ((local_clock >= dmin) && (random % 2 == 0)) o =i;
  else
    o = i;
```

```
  #1;
  local_clock = local_clock + 1;
end

always @(i) local_clock = 0;  // reset the clock every time the
                              // input changes

endmodule  //DELAY


//-----------------------------------------------------------------
//
//  module SKEW (delay for the transmitter components)
//
//-----------------------------------------------------------------
module SKEW (i, o);
input         i;
output        o;
reg           o;

parameter     smin  = 0,   // Minimum skew
              smax  = 0,   // Maximum skew
              mdist = 0,   // Minimum distance between consec. ticks

              a = 101,       // parameters of
              b = 7,         //     the
              n = 1013;      // random function

integer       random,        // random number
              old_deadline,  // scheduled last (old) change of the output
              delay,         // delay
              time1;         // used to convert $time into an integer value

initial begin
  random = 0;          // Initial value of the random variable
  old_deadline = -1;  // Initial value of the last (old) deadline
end

always @(i) begin
   random = (a * random + b) % n;       // to determine variable skew
   time1 = $time;                       // store the current sim. time

// compute the new delay (dmin <= delay <= dmax)
   delay = smin + ((random * (smax - smin))/(n-1));
```

```
// verify that it preserves minimum separation between changes to o.
   if (delay < old_deadline-time1+mdist)
     delay = old_deadline - time1 + mdist;

// schedule the output change to occur delay time units later
   o <= #delay i;

// remember this last deadline
   old_deadline = time1 + delay;
end

endmodule  // skew
```