2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems

Tiempo Asynchronous Circuits System Verilog Modeling Language

Marc Renaudin, Alain Fonkoua
TIEMPO - SAS
Montbonnot, France
marc.renaudin@tiempo-ic.com
alain.fonkoua@tiempo-ic.com

***Abstract*—This paper describes the SystemVerilog modeling language developed by Tiempo to design asynchronous circuits. The language enables designers to model, verify and debug asynchronous circuits using standard simulators, viewers and debuggers. The paper first highlights how the concept of communication channel is supported and how SystemVerilog is used to declare channels and ports, reading and writing ports and testing port's activity. The different memorization semantics associated with channels are addressed. Modeling and designing asynchronous circuit architectures using channels is then presented taking advantage of SystemVerilog modules and processes. The modeling of distributed and concurrent asynchronous circuits using the concepts defined in Tiempo SystemVerilog language is then described. An illustrative example shows the efficiency of the language as well as its ease-of-use.**

Keywords- SystemVerilog; Asynchronous Circuits; Channel; Handshake.

I. INTRODUCTION

This paper describes the language defined by Tiempo to model asynchronous circuits using System Verilog. One of the key goals of this work is to provide a complete and powerful language for the design and simulation of asynchronous circuits that enables the use of standard CAD tools available on the market. No specific simulator is required to simulate asynchronous modules. Any simulator compliant with System Verilog may be used provided that Tiempo predefined header files are supported.

Most of the HDLs (hardware description languages) have been considered in the past to model and synthesize asynchronous circuits. Among the possible candidates there are Verilog [1][3][8][10], SystemVerilog [2], VHDL [4][7] and SystemC [5][6]. Although widely used by designers, Verilog and VHDL were discarded because they lacked the interface concepts which provide a nice hook to implement the asynchronous semantics. The final choice of SystemVerilog is motivated by the need to provide synchronous designers with a language closer to hardware [9]. However, SystemC can perfectly be augmented with the same capabilities to support asynchronous design [5][6].

The language presented in this paper was designed to allow the development of a synthesis engine capable of targeting any asynchronous circuit styles, using any kind of handshaking protocols and templates. Synthesis is out of the scope of the paper, but note that Tiempo Asynchronous Circuit Compiler named "ACC" is supporting the language described in this paper. To our knowledge this is the first time a commercial procedural HDL language is used to support such a broad set of features to model asynchronous circuits, including parallelism and sequential composition, different communication and synchronization mechanisms, as well as different memorization semantics.

The paper organization follows a gradual presentation of the key concepts required and offered by the language to model asynchronous circuits and systems as communicating concurrent circuits. Section II describes the modeling of channels and the semantics associated to channels for communication, memorization and synchronization between concurrent modules. Section III deals with asynchronous circuit modeled by means of SystemVerilog modules and processes. To conform to Tiempo asynchronous semantics, some rules must be satisfied by SystemVerilog modules. These rules are exposed in Section III as well as typical examples illustrating the modeling style. Finally, the use of the language with a complete example is reported in section IV. Section V concludes the paper.

II. CHANNELS

Channels represent the basic medium for communication between asynchronous design entities and processes. A channel allows point-to-point communication between two processes. Each communication through a channel involves a token exchange between the two processes. The process initiating the communication is the active process. The other is the passive process. The channel end used by the active process is the active port of the channel. The other is the passive port. At any given time, the passive process may sense the channel to test if the active process has initiated a communication action. A channel is said ready when the active process has initiated a communication. Passive processes check the Ready property of a channel when they don't want to be blocked in a communication but just want to check if a communication is pending. A channel is characterized by:

- A type which determines the set of values that may be included in the token exchanged.

1522-8681/12 $26.00 © 2012 IEEE
DOI 10.1109/ASYNC.2012.22

- A protocol which specifies the handshake policy applicable.

Two protocols are defined: push and pull.

- In the push protocol, the active process writes a token to the channel and the passive process reads a token from the channel.
- In the pull protocol, the active process reads the token from the channel and the passive process writes the token to the channel.

Provided the Ready property of the channel is not evaluated, there is no need to specify a protocol of the channel: either one (push or pull) may be used.

No assumption is made about the data encoding used. One goal of the modeling was to allow all styles of channel encoding. In SystemVerilog channels are modeled with Tiempo predefined interfaces as follows:

- Internal channels are represented by interface instances.
- Channel ports are represented by interface modport port declarations.

A. Channel declaration

1) channel declaration

```
interface channel_TData
task automatic BeginRead(output TData data_out);
task automatic EndRead();
task automatic Read(output TData data_out);
task automatic Write(input TData data_in);
modport in ( import BeginRead, import EndRead,
             import Read, input Ready);
modport out (import Write);
endinterface
interface push_channel_TData #(parameter FIFO_DEPTH=0)
function automatic bit Ready;
task automatic BeginRead(output TData data_out);
task automatic EndRead();
task automatic Read(output TData data_out);
task automatic Write(input TData data_in);
modport in ( import BeginRead, import EndRead,
             import Read, input Ready);
modport out (import Write);
endinterface
interface pull_channel_TData
function automatic bit Ready;
task automatic BeginRead(output TData data_out);
task automatic EndRead();
task automatic Read(output TData data_out);
task automatic Write(input TData data_in);
modport in ( import BeginRead, import EndRead, import Read);
modport out (import Write, input Ready);
endinterface
```

Figure 1. SystemVerilog implementation of Channels

Internal channels are instances of SystemVerilog interfaces. Channel ports are **modport** of such interfaces. Figure 1. Sketches the various SystemVerilog interfaces used to implement channels and channel port declarations. These interfaces are declared by the following macro call:

```
`DEF_CHANNEL(TData)
```

The channels interfaces are used as follows:

- channel_TData: for the declaration of channels without specifying the protocol (push or pull) to be used. For these channels the read and write operations are defined. The active and passive processes are not specified, so the channel cannot be probed (Ready). This kind of declaration is recommended when the Ready property of the channel is not sensed by the passive process using the channel.
- push_channel_TData: channel declaration with push protocol. The active process is the writer and the passive the reader. The Ready property may be sensed by the passive process (the receiver).
- pull_channel_TData: channel declaration with pull protocol. The active process is the reader and the passive the writer. The Ready property may be sensed by the passive process (the transmitter).
- The optional FIFO_DEPTH parameter represents a positive integer count characterizing the FIFO buffering capacity associated with the channel. It is allowed only for push channels. The FIFO_DEPTH/2 quantity represents the maximum number of tokens that can be written on the channel without the intervention of any read operation. A channel with a FIFO_DEPTH of 1 is a half-buffer. A full-buffer is a channel with a FIFO_DEPTH of 2. The default value of the FIFO_DEPTH parameter is 0. Channels with a FIFO_DEPTH of 0 are unbuffered. Pull channels are always unbuffered. Channel ports are always unbuffered. Buffering capacity can only be specified for internal push channels.

Type **event_type** denotes a predefined enumeration type. It is defined as follows:

```
typedef enum {SREVENT} event_type;
```

It is used to define channels carrying no value. These are pure event channels. In fact any enumeration type with a single value may be used to define a pure event channel (cf. next section to see how to define channels of user-defined types).

Channel arrays can be defined by specifying a range after the channel identifier. The left hand index and right hand index specify the range of the array. When a range is specified, the interface instance is an array of channels. Each individual channel can be accessed by specifying an index value in the specified range.

For arrays of channels, a set of components are also defined by the DEF_MACRO which implement common dataflow channel operations:

- **split**: component used to split an input channel with vector type into a set of scalar sub elements. Thus byte-channel may be split into a set of 8 bit-channel other channels.
- **join**: component used to build a single composite channel out of a set of scalar channels.
- **merge**: component used to build a channel from a set of input channels assumed to receive token in a mutually exclusive way.

Examples:

```
push_channel_byte C[3:0]();      // channel array of bit[7:0], C[2] denotes a
                                 // channel of bit[7:0]
pull_channel_bit A[7:0]();       // channel array of bit
push_channel_byte Input();
push_channel_bit output[7:0]();
split_push_byte #(8) MySplit (Input,Output);// split component
```

2) template type channel

```
interface channel #( parameter type TData)
task automatic BeginRead(output TData data_out);
task automatic EndRead();
task automatic Read(output TData data_out);
task automatic Write(input TData data_in);
modport in ( import BeginRead, import EndRead,
             import Read, input Ready);
modport out (import Write);
endinterface
interface push_channel#(parameter type TData, parameter FIFO_DEPTH=0)
function automatic bit Ready;
task automatic BeginRead(output TData data_out);
task automatic EndRead();
task automatic Read(output TData data_out);
task automatic Write(input TData data_in);
modport in ( import BeginRead, import EndRead,
             import Read, input Ready);
modport out (import Write);
endinterface
interface pull_channel #(parameter type TData)
function automatic bit Ready;
task automatic BeginRead(output TData data_out);
task automatic EndRead();
task automatic Read(output TData data_out);
task automatic Write(input TData data_in);
modport in ( import BeginRead, import EndRead, import Read);
modport out (import Write, input Ready);
endinterface
```

Figure 2 template type interface channel Type template channel declaration with a template type parameter, **channel**, **push_channel** and **pull_channel** are interfaces defined with a type parameter denoting the type of values that can be carried by the channel instances. A channel can be declared with any integer type, enumeration type, packed structure or packed union. If no value is provided for the type parameter, bit type is assumed.

Examples:

```
push_channel #(int) int_ch();      // Declares a push channel of type int
pull_channel #(bit[7:0]) B();      // declares a pull channel to carry
                                   // 8-bits vectors
```

B. Channel port declaration

A channel port declaration is a SystemVerilog interface port declaration. The mode (**in** or **out**) of a channel is in fact a **modport** defined in the corresponding interface definition. Three categories of channel port are defined:

- channel_TData.mode: declaration of a channel port with the protocol specification left open.
- push_channel_type.mode, push_channel.mode: declaration of a push channel port.
- pull_channel_type.mode, pull_channel.mode: declaration of a pull channel port.

Two modports (in, out) specify the mode of the channel port with the following implications:

- Mode **in**: the corresponding port may be read within the module, but not written. If the port has push protocol, it can be probed (Ready function may be called on the port).
- Mode **Out**: the corresponding port may be written within the module, but not read. If the port has pull protocol, it can be probed.

C. Channel operations

Channels are used for point-to-point communications between two components. One of the components initiates the communication; it is the active component. The other is the passive component. Communication may involve data transfer between the components. Communication through channels follows a well-defined handshaking protocol. The active component initiates the communication and must wait for an acknowledgement from the passive component. To avoid deadlock or unintended non-determinism, some rules are enforced:

- A channel port must ultimately be used in exactly one process.
- An internal channel must be used in exactly two processes, one acting as the active component and the other as passive component.

Failing to observe the above rules may lead to deadlock and doesn't follow the asynchronous standards.

The following operations are defined in the channel interfaces:

- Read Channel Operation. The Read operation is refined into two sub operations BeginRead and EndRead operations that can be used to specify unbuffered channel communication as will be explained later.
- Write Channel Operation. Similarly, the Write operation can be refined into two sub operations BeginWrite and EndWrite operations that are not exposed to the user because they are always called in sequence.
- Channel Ready Property.

1) Read channel operation

Read channel operation allows a process to get data from the channel. The call suspends the process until the complete handshake protocol is done. After the reading, the data obtained may be used in the process without blocking the channel writer. In case many channels must be read, it may be useful to use the SystemVerilog fork statement to execute the read operations in parallel as illustrated in Figure 2. The Read channel operation may imply a memorization of the read data if it is used in later channel operations: the channel value is read and then stored so that the channel is immediately released.

2) BeginRead and EndRead suboperations on unbuffered channels

Another form of channel reading operation is provided so that the channel writer holds on until the channel reader has completed to process the data read. No memorization is required in that case, since the data are kept into the channel until the reader explicitly releases the channel. This particular reading operation thus allows the designer to control the availability of the data from the channel. To achieve this goal, the handshake protocol is separated into two procedures:

- The BeginRead operation which acquires the data without acknowledging the channel
- The EndRead operation which releases the channel.

Between the BeginRead and the EndRead, the data obtained may be used. Meanwhile, the channel writer is suspended. Note that usage of these operations minimizes the memorization. However, using a channel datum after the EndRead operation is not forbidden, but may imply memorization.

The A.Read(x) operation is equivalent to:

```
A.BeginRead(x);
A.EndRead();
```

See Figure 3. for an example with BeginRead/EndRead operations.

3) BeginRead and EndRead operations on buffered channels

If a channel is buffered, the read and write operations are not necessarily synchronized. A read operation requires two sub operations. Similarly a write requires two sub operations. The FIFO_DEPTH of a channel represents the maximum number of write sub operations that can be performed without any read sub operations. The dynamic FIFO depth of a channel is the difference between the number of write sub operations performed so far with respect to the number of read sub operations. If the dynamic FIFO depth is zero, BeginRead and EndRead sub operations behave the same way as for unbuffered channels. If the dynamic FIFO size is greater than zero, the BeginRead or EndRead operation completes immediately reducing by one the dynamic FIFO size. In the face of BeginRead, the active value of the channel is returned as the channel value. Adding buffers into channels enables concurrency between the reader and the writer.

4) Write Channel operation on unbuffered channel

Channel write operation allows a process to write a data on a channel following a handshake protocol. The writer is suspended until the handshake completes. When writing on different channels, it is recommended to use a fork statement for a parallel write.

Example: adder with implicit memorization

```
`include "std_async_defs.sv"
module adder (push_channel_bit.in A,   push_channel_bit.in B,
          push_channel_bit.in Cin, push_channel_bit.out Cout,
          push_channel_bit.out S);
  always begin:Add
    bit x,y,z;
    fork          // parallel read of channel input ports
      A.Read(x);
      B.Read(y);
      Cin.Read(z);
    join
    fork          // parallel write of channel output ports
      S.Write(x^y^z);
      Cout.Write(x&y|x&z|y&z);
    join
  end
endmodule
```

Figure 2. Adder with implicit memorization

In the code of Figure 2. the operands are first locally stored which releases the input channels, the addition is performed and the results written into the output channels. In the code of Figure 3. the input channels are only released after the results are computed and written into the output channels. In this case the input channels are acknowledged by the output channels, making the input and output handshaking protocol phases synchronized.

Example: adder minimizing memorization

```
module adder (
   push_channel_bit.in A, push_channel_bit.in B, push_channel_bit.in Cin,
   push_channel_bit.out Cout, push_channel_bit.out S);
  always begin:Add
    bit x,y,z;
    fork              // parallel read of channel input ports
      A.BeginRead(x);
      B.BeginRead(y);
      Cin.BeginRead(z);
    join
    fork              // parallel write of channel output ports
      S.Write(x^y^z);
      Cout.Write(x&y|x&z|y&z);
    join
    fork              // parallel release of channel input ports
      A.EndRead();
      B.EndRead();
      Cin.EndRead();
    join
  end
endmodule
```

Figure 3. Adder with no implicit memorization

5) Write Channel operation on buffered channel

When a write operation is performed on a buffered channel, the write operation returns immediately if the capacity of the channel (FIFO_DEPTH parameter) exceeds the –current size- of the channel by a value greater or equal than two.

```
module adder
#(parameter N=2)
 (
   push_channel_bit.in A, push_channel_bit.in B, push_channel_bit.in Cin,
   push_channel_bit.out Cout, push_channel_bit.out S);
push_channel_bit #(N) iA(),iB(),iCin();//internal channel with Full-buffer
tie_push_bit tA (A,iA); //tie component connecting channel A to iA
tie_push_bit tB (B,iB); //tie component connecting channel B to iB
tie_push_bit tCin (Cin,iCin); //tie component connecting channel Cin to iCin
  always begin:Add
    bit x,y,z;
    fork              // parallel read of channel input ports
      iA.BeginRead(x);
      iB.BeginRead(y);
      iCin.BeginRead(z);
    join
    fork              // parallel write of channel output ports
      S.Write(x^y^z);
      Cout.Write(x&y|x&z|y&z);
    join
    fork              // parallel release of channel input ports
      iA.EndRead();
      iB.EndRead();
      iCin.EndRead();
    join
  end
endmodule
```

Figure 4. Adder with Fine-grain controlled input buffers

If the dynamic FIFO size is equal to the channel capacity, the write operation behaves the same way as for unbuffered channels except that the intervening BeginRead sub-operation will consume the first channel FIFO value and the current

write operation value will be stored as the last value in the FIFO. If the capacity of the channel exceeds the dynamic FIFO-size by a value of one, the first sub-operation of the channel write operation completes. The second sub-operation of the channel write operation will complete as soon as the first BeginRead sub-operation will be performed on the channel.

The code in Figure 4. illustrates one of the means to include buffer into channels thus increasing the concurrency between computations.

It is important to notice here that the Read and BeginRead/EndRead statements, in addition to the ability to control the buffering capacity of channels, provides the designer with the necessary tools to model asynchronous circuits with memorization and concurrency.

6) Channel Ready property

For each channel, a Ready property is defined as a bit signal that is constantly updated so as to reflect the status of the channel. The Ready property is available to passive port of channels to sense if communication has been initiated over the channel by the corresponding active port. It returns a bit value. A '1' is returned if the active component has initiated communication over the channel and a '0' otherwise.

For any type of push channel (push_channel*), the reader process may invoke the Ready property of the channel.

For any type of pull channel (pull_channel*), the writer process may sense the Ready property of the channel.

There is no Ready property defined for channel with open protocol specification (channel*).

The wait channel operation is a SystemVerilog wait statement with an expression formed out of channel Ready function calls combined with logical operators. For the following wait statement:

```
wait(A.Ready || B.Ready)
unique if(A.Ready) A.Read(x);
else  if(B.Ready) B.Read(y);
```

the process is suspended until channel A or channel B is ready. The wait channel statement allows guarded command style modeling. In this mode, the behavior is expressed as a set of guarded commands.

```
module merge (push_channel_bit.in A, push_channel_bit.in B,
        push_channel_bit.out S);
  always begin:Merge
    bit x;
    wait(A.Ready | B.Ready)
    unique if (A.Ready) begin  A.BeginRead(x); S.Write(x); A.EndRead();
    end else if (B.Ready) begin  B.BeginRead(x);  S.Write(x);  B.EndRead();
    end
  end
endmodule
```

Figure 5. Deterministic Merge

The SystemVerilog wait statement is used in Figure 5. to suspend the process until one of the channels becomes ready. It is an explicit form of synchronization allowed to model asynchronous processes.

Note the use of SystemVerilog "**unique if**" instruction which expresses the user intent that exactly one of the cascaded if condition must be true. A warning is generated by the simulator if the condition is violated.

```
module Fork (pull_channel_bit.in E, pull_channel_bit.out A,
        pull_channel_bit.out B);
  always begin:Fork_p
    bit x;
    E.BeginRead(x);
    wait(A.Ready | B.Ready)
    unique case(1)
      A.Ready : A.Write(x);
      B.Ready : B.Write(x);
    endcase
    E.EndRead();
  end
endmodule
```

Figure 6. Deterministic Fork

In Figure 6. the wait channel Ready is sensing the output pull-channel ports A and B. Remember that pull channel output ports are passive. Note the use of the "**unique case**" statement which specifies that the case is full (all cases are covered) and exactly one alternative is valid.

III. ASYNCHRONOUS CIRCUITS

A. Asynchronous module definition

An asynchronous circuit is described by an asynchronous module. An asynchronous module is a SystemVerilog module which communicates with its environment through channel ports following a handshaking protocol.

The ports of an asynchronous module can be interface channel ports. Channel ports and internal channels are addressed in section II.

An asynchronous module specifies a set of concurrent dataflow computations interacting through channels following a well-defined handshaking protocol. To comply with the handshaking protocol, some restrictions must be set on the use of SystemVerilog to model asynchronous modules.

In SystemVerilog, the statements of an asynchronous module are constrained to be one of:

- interface instantiation statement, in particular channel interface instantiations;
- asynchronous process statement: always process statement with restrictions described in section III.B.1;
- generate statements;
- module instances.

The behavior of an asynchronous module may be defined by a set of concurrent asynchronous processes and asynchronous module instantiation statements. Synchronization of the various asynchronous process statements is almost exclusively done by handshake through channels. Internal nets and wire declarations are of no use within an asynchronous module except when used for connection with synchronous world. Communication with synchronous modules is possible but not covered in the paper.

```
module ADDER ( channel_bit.in A, channel_bit.in B, channel_bit.in C,
               channel_bit.out Cout, channel_bit.out S);
  always begin:ADD
    bit x,y,z,vs,vc;
    fork
      A.Read(x);
      B.Read(y);
      C.Read(z);
    join
    vs = x ^ y ^ z;
    vc = (x & y)|(x&z)|(y&z);
    fork
      S.Write(vs);
      Cout.Write(vc);
    join
  end
endmodule
```

Figure 7. Adder Module with a single process

The ports of an asynchronous module can only be channels. Channels are modeled by Tiempo predefined interfaces. In the example of Figure 7. , A, B and C are declared as interface ports channel_bit with **modeport** in. They represent passive channel port of mode in. Cout and S are declared as active channel ports of mode out. It is the meaning of "channel_bit.out". The protocol of the channel ports is not specified in this example.

Parallel read operations are performed on the A, B and C inputs, followed by computation of sum (vs) and carry out (vc) and followed by parallel write of channel ports S, Cout with the values previously computed.

B. Asynchronous process definition

1) Handshakes and synchronization in a process

An asynchronous process specifies a dataflow network relating a set of channels read by the process to another set of channels written by the process. The synchronization is exclusively done through channel handshaking operations. The signal event statements (**posedge**, **negedge**,..) cannot be used. An asynchronous process is implemented by a SystemVerilog always process statement containing no event control or event expression. Because they imply some event control statements, the **always_ff**, **always_latch** and **always_comb** processes shall not be used for asynchronous processes.

```
module slow_half_adder (channel_bit.in A, channel_bit.in B,
                        channel_bit.out C, channel_bit.out S);
  always begin:
    bit x,y,z,c;
    A.Read(x);
    B.Read(y);
    z=x ^ y;
    c=x & y;
    S.Write(z);
    C.Write(c);
  end
endmodule
```

Figure 8. Procedural adder process

In the asynchronous process of Figure 8. , there is no event statement. But the process may suspend while executing channel read-write operation. In the handshake protocol, a read operation on a channel can complete only if a write operation is executed on the same channel. Similarly, a write operation can complete only if a read operation is done on the same channel. The asynchronous process above requires a strict execution order: A read operation must complete on channel A before the read operation on channel B is executed. When the sequence of read operations are completed, the write operation on channel S can be executed using the "exclusive or" of the values obtained from A and B. After that, the write operation is executed on channel C using the conjunction of the values obtained from A and B.

The always process is a procedural way to specify a dataflow network in terms of tokens flowing from input channels to output channels. The half-adder presented in the here-above example is slow because all operations within the process are serialized. In specifying the dataflow network, the fork-join construct of SystemVerilog will prove useful to specify concurrent channel operations as illustrated by the ADDER of Figure 7. .

The recommended way for processes to exchange data is through channels. Shared variable are also available in Tiempo SystemVerilog language but not addressed in this paper.

2) Rules related to the dataflow

Though a process may spawn many different threads, the dataflow implied by the process must be unambiguously and clearly defined. The simulation behavior of the process must be predictable and should not depend on the execution order of the threads. To this aim, the use of variables between these threads must conform to the following rules:

- No variable write-write conflict between threads: A given variable must be assigned by at most a single thread.
- No variable read-write conflict between threads. A given variable may not be written and read by two parallel threads.
- No channel read conflict: the same channel may not be read in two concurrent threads.
- Channel write conflicts are forbidden: two concurrent threads are not allowed to write into the same channel.

The rules for variables apply only to process threads, not to processes. SystemVerilog variables must be used exclusively within a single process. Channels are the sole mean of communication between processes. The next two sections address the modeling of deterministic and non-deterministic behavior.

3) Deterministic and non-deterministic procedural statements

Each guarded command includes two components: a Boolean expression called guard expression or guard, and a set of statements.

The model continually senses its guard expressions. As soon as one becomes true, the associated commands are executed. The wait statement is necessary to suspend the process until or unless at least one of the guarded expressions is true. Removing the wait statement may lead to an infinite loop if all the guard expressions are false.

Using the wait statement, a set of guarded commands can be written as shown in Figure 9. .

```
always begin
   wait (guard1 || guard2)
   unique case(1)
     // guard1: commands1
     // guard2: commands2
   endcase
```

Figure 9. Guarded commands example.

a) Unique-if and unique-case statements and deterministic choices

Unique-if and unique-case statements allow the user to specify that a case or cascaded if statement should be handled as a full and parallel case. For a sequence of "if-else if" statements, it means exactly one condition must be true. For a case statement, it means exactly one alternative will be matched. These conditions are checked during simulation and warnings are reported whenever they are violated. Using unique-if or unique-case statement is equivalent to implying a deterministic guarded command style.

The models displayed in Figure 5. and Figure 6. are correct only if channels A and B cannot be ready concurrently. If this condition is violated, the model is not correct and a non-deterministic style must be considered.

b) SystemVerilog randcase statement and non-deterministic choices

Using the **randcase** SystemVerilog statement, merging two channels into one can be modeled as in Figure 10.

```
module non_deterministic_merge (push_channel_bit.in A,
                                push_channel_bit.in B, push_channel_bit.out S);
  always begin:Merge
    bit x;
    wait(A.Ready | B.Ready)
    randcase
    A.Ready:begin
        A.BeginRead(x);
        S.Write(x);
        A.EndRead();
      end
    B.Ready: begin
        B.BeginRead(x);
        S.Write(x);
        B.EndRead();
     end
    endcase
  end
endmodule
```

Figure 10. Non deterministic choice modeling using the **rand case** statement.

```
module non_deterministic_fork (pull_channel_bit.in E,
                               pull_channel_bit.out A,  pull_channel_bit.out B);
  always begin:Fork
    bit x;
    E.BeginRead(x);
    wait(A.Ready | B.Ready)
    randcase
    A.Ready : A.Write(x);
    B.Ready : B.Write(x);
    endrandcase
    E.EndRead();
  end
endmodule
```

Figure 11. Non deterministic fork modeling using the rand case statement.

An alternative is not selected if the associated condition evaluates to 0. If both evaluate to 1, one is selected at random. The example of Figure 11. provides a model of a non-deterministic fork.

IV. A COMPLETE MODELING EXAMPLE

As an illustrative example, a CRC checker module is considered. This module computes a CRC from a stream of bytes and checks if it is consistent with the CRC provided at the end of the data stream. The number of bytes to process is provided at the beginning of the computation using a separate channel.

The module specification is depicted in Figure 12. It is communicating with the environment using a 4-bit input channel to specify the byte count, an 8-bit input channel carrying the data stream and a 1-bit output channel to tell if the CRC checking is successful or not.

The module is composed of two sub-modules, a finite state machine and a computational block (Figure 12.). The finite state machine is taking as input the byte count and is controlling the other module using the CONTROL channel. The computational block is taking as input the data stream, computes the CRC, checks the final result and finally outputs the status through the CRC_STATUS channel.

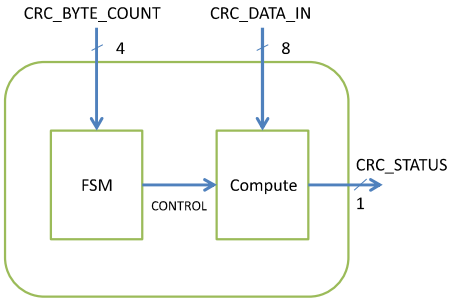

Figure 12. Synoptic of the CRC circuit.

The SystemVerilog model of this CRC checker is given and described in Figure 13.

```
typedef bit unsigned [3:0] bit4_t;
`DEF_CHANNEL(bit4_t)
typedef enum {INIT, COMPUTE, CHECK} command_t;
`DEF_CHANNEL(command_t)
```

```
module  CRC (
        push_channel_byte.in CRC_DATA_IN,
        push_channel_bit4_t.in CRC_BYTE_COUNT,
        push_channel_bit.out CRC_STATUS  );
push_channel_command_t CONTROL ();

FSM  u_fsm (
.CONTROL(CONTROL), .CRC_BYTE_COUNT(CRC_BYTE_COUNT)
);
COMPUTE u_comp (
.CRC_DATA_IN(CRC_DATA_IN),.CONTROL(CONTROL),
.CRC_STATUS(CRC_STATUS));
endmodule
```

Figure 13. CRC module.

The module CRC defines the interface with the environment and instantiates the two sub-modules FSM and COMPUTE described in Figure 14. and Figure 15.

```
module FSM (
push_channel_bit4_t.in  CRC_BYTE_COUNT,
push_channel_command_t.out   CONTROL
 );
always begin : process_orchestra
  bit [3:0] byte_counter;
  begin : sequential_machine
    CONTROL.Write(INIT);                          // 1st state
    CRC_BYTE_COUNT.Read(byte_counter);            // 2nd state
    while (byte_counter!=4'h0)
       begin
         CONTROL.Write(COMPUTE);                  // 3rd state
         byte_counter=byte_counter-4'h1;
       end
    CONTROL.Write(CHECK);                         // 4th state
  end // block: sequential_machine
end // block: process_orchestra
endmodule // fsm
```

Figure 14. FSM module.

This FSM module is modeled as a procedural four-state sequential machine. The first state initializes the COMPUTE module; the second state reads the byte-count out of the environment; the third and fourth states are used to control the computation iterations and output the result. Note the simplicity of the procedural model which is using Read and Write operations over channels and a while loop to control the number of iterations. Modeling the FSM using explicit states would also be possible but more complex in this case. Mixing both styles very often leads to efficient solutions.

```
module COMPUTE (
                        push_channel_command_t.in CONTROL,
                        push_channel_byte.in  CRC_DATA_IN,
                        push_channel_bit.out CRC_STATUS
                        );
parameter POLYNOM=8'hd5;// Std CRC8 polynom
  always begin
    byte numerator, input_crc, prev_crc, new_crc;
    state_t command;
    bit temp;
    bit [3:0] i;
    CONTROL.BeginRead(current_state);
    unique case (command)
       INIT: prev_crc = 8'h00; // Initialize crc value
       COMPUTE:              // Compute the checksum
        begin : serialize
          CRC_DATA_IN.Read(numerator);
          for (i=4'h0;i<4'h8;i=i+4'h1)
            begin : for_loop  // CRC calculation algo
                    temp = prev_crc[7] ^ numerator[i];
                    if (temp)
                      new_crc={prev_crc[6:0],1'b0}^POLYNOM;
                    else
                      new_crc={prev_crc[6:0],1'b0};
                    prev_crc=new_crc;
            end
        end // block: serialize
       CHECK:
        // Verify calculated CRC value equal concatenated CRC data value
        begin : check_CRC
          CRC_DATA_IN.Read(input_crc);
          unique if ( input_crc == prev_crc)
            CRC_STATUS.Write(1'b1);
          else
            CRC_STATUS.Write(1'b0);
        end // block : check_CRC
    endcase // unique case (command)
    CONTROL.EndRead();
  end // always begin
endmodule
```

Figure 15. Compute module.

The COMPUTE module is a combinational module which starts reading the command and decodes it using a **case** statement. Avoiding memorization of the command value is achieved using the BeginRead and EndRead operations. On the contrary, the Read statement is used to access the input data stream. It locally stores the data to process, releasing the CRC_DATA_IN channel as quickly as possible and therefore avoids stalling the environment. This example illustrates the typical use of the language and how the specification can easily be structured into asynchronous modules communicating through channels. Despite the fact that the considered example is purely sequential, concurrency is very easily modeled as well. Pipelined or parallel computations can be modeled combining the features described in sections II and III.

V. CONCLUSION

In this paper, we have shown how the language defined by Tiempo based on SystemVerilog provides designers the means to model asynchronous circuits. A full set of communication and synchronization mechanisms between concurrent circuits through channels are provided. Different memorization semantics are supported in order to design sequential as well as concurrent and pipelined circuits. All these features provide the mandatory ingredients that are required to design efficient asynchronous circuits.

The choice of a standard language such as SystemVerilog, not only enables designers to use their favorite simulators and debuggers, but also enables them to mix asynchronous and synchronous models and to use standard verification methodologies.

REFERENCES

[1] A. Saifhashemi, H. Pedram, "Verilog HDL, powered by PLI: a suitable framework for describing and modeling asynchronous circuits at all levels of abstraction", in Proceedings DAC'03, pp. 330-333.

[2] A. Saifhashemi, P. Beerel, "SystemVerilogCSP : Modeling Digital asynchronous Cirsuits Using SystemVerilog Interfaces", Communicating Process Architectures, 2011, ISBN 978-1-60750-773-4.

[3] I. Blunno, L. Lavagno, "Automated Synthesis of Micro-Pipelines from Behavioral Verilog HDL", in Proceedings ASYNC'2000, pp. 84-92.

[4] M. Ligthart, et al, "Asynchronous design using commercial HDL synthesis tools", in Proceedings ASYNC'2000, pp. 114 – 125.

[5] C. Koch-Hofer, M. Renaudin, Y. Thonnard, P. Vivet, "ASC, a SystemC extension for Modeling Asynchronous Systems, and its application to an Asynchronous NoC", in Proceedings NOCs'07, Princeton, New Jersey, USA, May 7-9, 2007, pp. 295-306.

[6] C. Koch-Hofer, M. Renaudin, "Timed Asynchronous Circuits Modeling using SystemC", in Proceedings FDL'07, Barcelona, September 18-20, 2007.

[7] M. Renaudin, P. Vivet, F. Robin, "A design framework for asynchronous/synchronous circuits based on CHP to HDL translation", in Proceedings ASYNC'99, Barcelona, April 18-22, 1999, pp. 135-144.

[8] Rong Zhou, Kwen-Siong Chong, Bah-Hwee Gwee, Chang, J.S., "Quasi-delay-insensitive compiler: Automatic synthesis of asynchronous circuits from verilog specifications", in Proceedings MWSCAS, 2011, pp. 1-4.

[9] TIEMPO, "Introduction to SystemVerilog Asynchronous Modeling", White Paper #2, www.tiempo-ic.com

[10] Chong-Fatt Law, Bah-Hwee Gwee, Chang J.S, "Modeling and Synthesis of Asynchronous Pipelines", in IEEE Transactions on VLSI, Vol. 19, N°. 4, pp. 682-695, 2011.