VHDL Observers for Clock Constraint Checking

Charles André, Frédéric Mallet, Julien Deantoni

Full Text available on the publisher website: http://ieeexplore.ieee.org/document/5551372/
DOI: 10.1109/SIES.2010.5551372

VHDL Observers for Clock Constraint Checking

Charles André, Frédéric Mallet, Julien DeAntoni
Université Nice Sophia Antipolis
I3S - UMR 7271 CNRS
INRIA Sophia Antipolis Méditerranée
06902 Sophia Antipolis cedex, FRANCE
Email: {Firstname.Lastname}@unice.fr

Abstract

Logical time has proved very useful to model heterogeneous and concurrent systems at various abstraction levels. The Clock Constraint Specification Language (CCSL) uses logical clocks as first-class citizens and supports a set of (logical) time patterns to specify the time behavior of systems. We promote here the use of CCSL to express and verify safety properties of VHDL designs. Our proposal relies on an automatic transformation of a CCSL specification into VHDL code that checks the expected properties. Being written in VHDL this code can be integrated in a classical VHDL design and verification flow. Our proposed structural transformation assembles instances of pre-built VHDL components while preserving the polychronous semantics of CCSL. This is not trivial due to major differences between the discrete-time delta cycle based semantics of VHDL and the fixed point semantics of CCSL. This paper describes these differences and proposes solutions to deal with them so as to build VHDL observers for the kernel CCSL constraints. We illustrate the approach by verifying an open-source implementation of the AMBA AHB-to-ABP bridge.

I. Introduction

The UML Profile for Modeling and Analysis of Real-Time and Embedded systems [1] (MARTE), adopted in November 2009, has introduced a *Time model* [2] that extends the informal *Simple Time* of UML2. This time model is general enough to support different forms of time (discrete or dense, chronometric or logical), accessed through model elements called *clocks*. The time model came with a companion language, called *Clock Constraint Specification Language* (CCSL) and defined in an annex of the MARTE specification. Initially devised as a simple language for expressing constraints between clocks of a MARTE model, CCSL has evolved and has been developed independently of the UML. CCSL is now equipped with a formal semantics [3] and is supported by a software environment[1] that allows specification, resolution, and visualization of clock constraints.

In this paper, we exploit the capability of CCSL to represent *multiform logical time*. In logical time, only the ordering of instants matters, not the actual "physical" duration of time between them. For modeling purpose, a *logical clock* can be associated with any event. This (logical) clock ticks whenever the event occurs, thus the event serves as a time reference. Considering several independent (or loosely dependent) events leads to the concept of multiform logical time. Modeling with logical time partial ordering was advocated in [4]. The notion of multiform (or polychronous) logical time has been exploited extensively in the theory of Synchronous languages [5], and in HDLs (Hardware Description Languages).

The design process for complex electronics systems makes use of numerous models different in their abstraction level and their nature (underlying model of computation) [6]. Usually, the most abstract models are *untimed* or causal. Timing information, regarded as non-functional properties, is introduced in later stages and has the form of "real-time constraints". Since time information may also carry functional intent, some time constraints, expressed as logical time constraints, should be part of the functional models, even at high abstraction levels. Logical time is flexible enough to unify causal and temporal relations under a unique concept. Therefore CCSL is well-adapted to capture these specifications for all these levels. Once the implementation realized, it is advisable to verify that it is correct with regard to the specification. *Observers* are one of the possible techniques of verification. As its name indicates, an observer continuously observes executions of a system to detect some specific, often undesirable, behaviors. Usually, the observer is written in the same language as the model (*e.g.,*Esterel, SystemC, VHDL).

Starting with the structural operational semantics of CCSL, the condition to violate a clock constraint can be derived, and then the implementation of the corresponding observer in a target programming language is possible. This has been successfully done for CCSL with the synchronous language Esterel [7]. A research report [8] details a library of Esterel modules that allows the automated construction of observers for any CCSL specification. The key of the success resides in the closeness of the semantics of CCSL and Esterel that are both instant-based fixed point semantics. We propose here to adapt and detail this approach for VHDL implementations. The adaptation is not trivial because of the semantic gap between the CCSL semantics and the simulation semantics of VHDL based on microsteps with delta cycles. The comparison between the two semantics and the construction of a library of VHDL components for CCSL specification checking is one of the contributions of this paper.

[1] TimeSquare, available at http://www-sop.inria.fr/aoste/dev/time_square

The approach is illustrated on a AHB to APB Bridge that is part of a larger design (a LeonII-based embedded system, whose VHDL model is available in open source[2]).

Section II introduces the main features of CCSL and the principle of using observers for the verification. Then, Section III briefly describes the AMBA bridge. The general method of building VHDL observers for CCSL constraints along with some precise examples are given in Section IV. A discussion on verification results and related works follows in Section V. Finally, we draw conclusions and open perspectives.

II. CCSL

This section briefly introduces the logical time model of MARTE [1] and the Clock Constraint Specification Language (CCSL).

A. Logical time model

A *clock* is a totally ordered set of *instants*. A *time structure* is a set of clocks C and relations on instants. The basic relations are *precedence* ($\prec$), *coincidence* ($\equiv$), and *exclusion* ($\#$). For any instants i and j in a time structure, $i \prec j$ means that the only acceptable execution traces are those where i occurs strictly before (precedes) j. $i \equiv j$ imposes instants i and j to be coincident, whereas $i \# j$ forbids the coincidence of the two instants. In this paper, we consider discrete sets of instants only, so that the instants of a clock can be indexed by natural numbers. For a clock c, $c[k]$ denotes its k^{th} instant.

Specifying a full time structure using only instant relations is not realistic since clocks are usually infinite sets of instants. Thus an enumerative specification of instant relations is forbidden. Hence the idea to extend relations to clocks. The Clock Constraint Specification Language (CCSL) has been defined to specify such relations between clocks. As an example, consider the clock relation *precedence* (denoted $\boxed{\prec}$). $a \boxed{\prec} b$, read 'a precedes b' or also 'a is *faster than* b', specifies that for all instants of clock a, its n^{th} instant *precedes* the n^{th} instant of clock b. More formally:

$$a \boxed{\prec} b \text{ means } \forall k \in \mathbb{N}^\star, a[k] \prec b[k].$$

A CCSL specification consists of clock declarations and conjunctions of *clock relations* between *clock expressions*. A clock expression is a declared clock or a new clock defined from existing ones. An example of clock expression is *delay* (denoted $\$$). $a \$ n$ specifies that a new clock is created and is the exact image of a, delayed for n instants of a. Note that this expression is a simplified version of the expression *defer*, which specifies that a clock can be delayed for a number of instants counted on another clock. For simplicity, we give only the semantics of the *delay*:

$a \$ n$ defines a clock c such that $\forall k \in \mathbb{N}^\star, c[k] \equiv a[k+n]$

A technical report [3] describes the syntax and the semantics of a kernel set of CCSL constraints.

B. CCSL in modeling

In modeling, CCSL clocks are used to represent events of the system, or more precisely the instants of a clock represent the occurrences of an event. Of course, all events of the system need not be clocks; only relevant events, regarding the desired specification, are used. Note that these events can be conceptual and later on associated with a model element or combination of model elements. This is illustrated in subsection III-B.

An operational view of CCSL relations and expressions is given by a Structural Operational Semantics (SOS). This semantics gives operational rules to build an execution that conforms to a CCSL specification. For each statement (relation or expression), initialization, enabling condition, and internal state evolution rules are provided. All the initialization rules of all statements should be computed at the beginning of the execution. Then, all the enabling conditions are evaluated to deduce the set $\mathscr{F} \subset \wp(C)$, whose elements $F \in \mathscr{F}$ are valid sets of clocks (*i.e.*,each clock in F satisfies all the enabling conditions). Since CCSL specifications are allowed to be non-deterministic, $\mathscr{F}$ is usually not a singleton. Any F from $\mathscr{F}$ can be chosen as the set of the clocks that actually tick in the reaction. In a simulation, F is imposed by the system under observation. The verification then consists in checking that the observed set of ticking clocks F is in $\mathscr{F}$. Giving F, the system evolves by processing the internal state evolution rule of each statement.

As an introduction to the CCSL SOS semantics, we give the rules for one relation and one expression. For the relation *precedence* $a \boxed{\prec} b$, the SOS rules are as follows:

initialization: $\delta : \mathbb{N} = 0$

enabling condition:

$$(\delta = 0) \Rightarrow \neg b \tag{1}$$

internal state evolution:

$$\delta \leftarrow \begin{cases} \delta + 1 & \text{if } a \wedge \neg b \\ \delta - 1 & \text{if } \neg a \wedge b \\ \delta & \text{otherwise.} \end{cases} \tag{2}$$

[2]http://www.gaisler.com

The internal state of this relation is δ, a natural number. δ is initialized to 0. When $\delta = 0$, b cannot tick, and a is free to tick or not. For a given set of ticking clocks, if a ticks and b does not then δ is incremented by 1. If a does not tick and b ticks then δ is decremented by 1. In any other cases, δ is unchanged.

For expression *delay*(n), the internal state is bv, an array of bits of length n. Let c be the clock defined by a \$ n, the corresponding SOS rules are:

initialization: $bv : Bit[n] = 0^n$

enabling condition:

$$\begin{aligned} (bv[0] = 0) &\Rightarrow \neg c \\ (bv[0] = 1) &\Rightarrow (c = a) \end{aligned} \tag{3}$$

internal state evolution:

$$\begin{aligned} \forall i \in [0..n-2], \quad bv[i] &\leftarrow \begin{cases} bv[i+1] & \text{if } a \\ bv[i] & \text{otherwise} \end{cases} \\ bv[n-1] &\leftarrow \begin{cases} 1 & \text{if } a \\ bv[n-1] & \text{otherwise} \end{cases} \end{aligned} \tag{4}$$

Initially, bv is filled up with 0. When the first bit of bv is 1, the constraint is $c = a$, which means that either a and c do not tick or they both tick. If $bv[0]$ is 0, then c cannot tick. For a given set of ticking clocks, if a ticks then bv is left-shifted and the entering value ($bv[n-1]$) is 1. If a does not tick, bv is unchanged.

These rules are used in subsection IV-D for the construction of the observers.

C. Property checking with observers

1) Observers: Verification by observers is a technique widely applied to property analysis / checking [9], [10], [11], [12]. Their goal is to observe a program to check if some given properties hold. Often the observers are used at runtime or in simulation. As represented in figure 1, an observer can see input, output and internal events or values of the program. If the observed evolution does not satisfy one expected property, the observer enters a failure state and reports a *violation*.

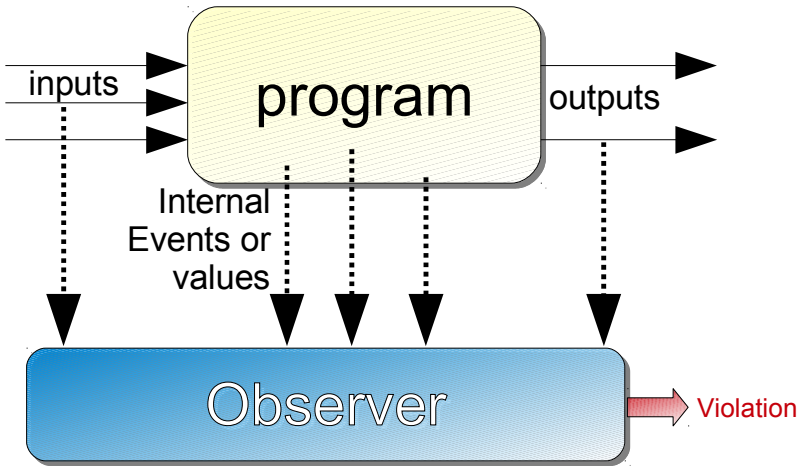

Fig. 1. Property checking of reactive programs with an observer.

CCSL specifies acceptable behaviors by a set of constraints. To verify that a specific program has an acceptable behavior, we build an observer for each constraint. It can be seen as an observer of a *safety property*. The generation of these observers is detailed in the next subsection.

2) Principle of the generation of observers: A CCSL specification is a set of possibly inter-related constraints. Our goal is to generate an observer from the CCSL specification. We propose to create a library of "components", for each CCSL constraint (relations and expressions) and perform a structural generation. Relations and expressions are of a different nature. An expression defines a new clock. For each expression we build a component called a *generator*. A relation constrains two clocks. Since we use non-intrusive observers, we cannot force anything to happen and we can only observe violations. For each relation, we build a component called an *observer*, which has two inputs (one for each clock) and one ouput: the violation signal. A violation of the specification will occur if any violation output of any relation *observer* is asserted.

A *generator* has to create a new clock depending on its inputs. Those inputs are given by the program under test or by another generator, so that the resulting clock is fully deterministic. For this reason, the specification of a generator is exactly the SOS rule of the corresponding expression.

An *observer* must verify that something bad never happens. However, the SOS rule corresponding to a relation specifies what should happen. Consequently, a violation occurs when the incoming clocks falsify the enabling condition. Therefore, the *violation condition*—checked by the observer—is just the logical negation of the enabling condition. The initial conditions and the internal state evolution rules remain unchanged.

From the SOS rules, it is possible to obtain the specification for the generators and the observers. Both of them consider logical clocks as inputs. However, the targeted implementation provides valued signals, not logical clocks. Thus, a conversion between signals (or possibly a combination of signals) into logical clocks is required. Such an adaptation is done by specific hand-made components called *adaptors*. Adaptors are also very useful to use the same specification for different implementation of the same system at different abstraction levels. Adaptors are sometimes called *transactors* when they play such a role. Examples of such adaptors are given in Section III.

According to the CCSL syntax, each clock relation can be represented as a tree. The root of this tree is the clock relation, the leaves are clocks, and intermediate nodes are clock expressions. Thus, the components used to implement a clock relation checker are assembled as a tree. An observer component is the root, adaptor components are the leaves, and generator components are the intermediate nodes. It is important to note that this structure is acyclic: information starting from the leaves eventually arrives at the root. For optimization reasons, some composants may be shared. So, the actual structure can be a DAG (Directed Acyclic Graph) of components, whose maximal elements are observer components, and minimal elements are adaptor components (see figure 7). Be it a tree or a DAG, we call an assembly of components used to check a clock relation an *observation network*.

Once the specification of adaptors, generators and observers is done, it must be realized in the implementation language. In the remainder of this paper, we detail this possibly complex task, when the simulation semantics of the implementation differs from the semantics of the specification language. This is highlighted by the creation of a VHDL component library for CCSL.

III. Example: an AMBA Bridge

A. AHB to APB Bridge

The *Advanced Microcontroller Bus Architecture* (AMBA) specification defines an onchip communications standard for designing high-performance embedded microcontrollers. We consider two buses defined with the AMBA specification:

- The *Advance High-performance Bus* (AHB) for high-performance, high clock frequency system modules;
- The *Advanced Peripheral Bus* (APB) optimized for minimal power consumption and reduced interface complexity to support peripheral functions.

In a typical AMBA architecture, which contains both types of bus, an AHB to APB *bridge* is necessary. The APB bridge interfaces the AHB to the APB and converts system bus transfers into APB transfers. It buffers address, control, and data from the AHB, drives the APB peripherals and returns data or response signals to the AHB. On a data transfer request, it decodes the address using an internal address map and generates a peripheral select, PSELx. Only one select signal can be active during a transfer. The bridge drives the data onto the APB for write transfers or, in case of read transfers, it drives the APB data onto the system bus.

B. Bridge specification

Figure 2 illustrates a write transfer on the APB bridge. The transfer starts when the destination address is written in HADDR. A central address decoder is used to provide a select signal, HSELx, for each slave on the AHB bus. The select signal is a combinatorial decode of the high-order address signals. Let $\mathsf{HSEL_B}$ be the select signal for the bridge. When HADDR is set to a value within a given address range, $\mathsf{HSEL_B}$ is set to high and the bridge should initiate a transfer (at T2). A write transfer is initiated when HWRITE is set to high, a read transfer is initiated otherwise.

For write transfers, the data must be given in HWDATA and must be available at the next cycle (at T3). Each transfer takes exactly two cycles to complete on the APB. In a first step (T3-T4), the address is further decoded by the bridge to select the appropriate APB slave. The address is set in PADDR, the date is set in PWDATA and the appropriate PSEL signal is asserted. In a second step (T4-T5), PENABLE is asserted and the write transaction is completed.

From this specification we attempt to extract a *higher view* of the transaction and identify the logical events that can be modeled as logical clocks. We identify two logical clocks here: tb_s (transfer bridge start), whose instants characterize the initiation of the transfer; tb_f (transfer bridge finish), which characterizes the completion of the transfer. A basic property that should be valid for any kind of transfer is that the initiation should always precede the completion. Such a property can be expressed in CCSL using the relation *precedence* (see Eq. 5).

$$tb_s \prec tb_f \tag{5}$$

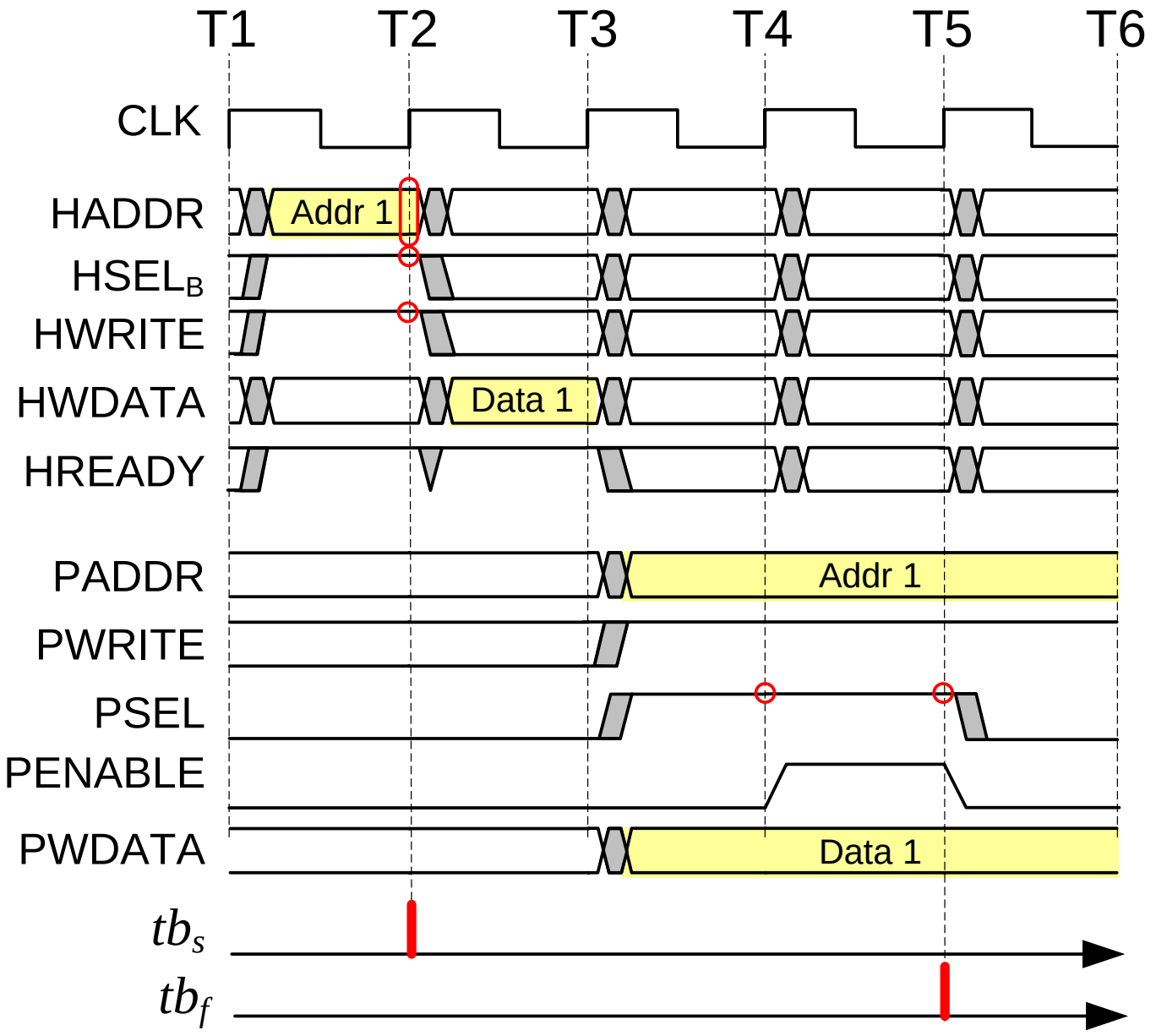

Fig. 2. A typical write transfer through the bridge.

To bridge the abstraction gap between the high-level logical view of the specification and the specific RTL implementation, we need an *adaptor*. To build such an adaptor, we have to decide what exactly is considered as the initiation of the transfer and what is considered as the end. Focusing on the initiation (tb_s), several solutions are possible. An asynchronous view (ignoring the bus clock CLK) could consider the rising edge of signal $\mathsf{HSEL_B}$ (*i.e.,*some time between T1 and T2) as the actual initiation. A synchronous view would rather consider the rising edge of signal CLK when $\mathsf{HSEL_B}$ is high (at T2). Both solutions are acceptable. The latter one has been chosen and depicted in Figure 2.

Considering now the case of the transfer completion (tb_f), the adaption appears a bit more complex. A synchronous interpretation of the transfer dictates that the actual completion occurs on the second raising edge of CLK when PSEL has been continuously high during two cycles (at T5). An asynchronous interpretation could consider the transfer completion on the falling edge of PENABLE. Section IV-D gives a VHDL implementation of the synchronous version.

C. Bounded transfers

The simple specification provided in Eq. 5 is general to any request/response or producer/consumer system. In the APB bridge, the request is the transfer initiation (tb_s) and the response is the transfer completion (tb_f). Such a transfer is unbounded, it

only specifies that the response must come at some point but it can be arbitrarily far from the request. In most cases, this is not suitable and transfers need to be bounded. This is actually the case for the AHB to APB bridge, whose specification explicitly mention a bound of 2. That is to say that at most two (but no more) consecutive requests can be performed even though the response to the first request has not been given yet.

For a buffer of size n, the specification would be that the k^{th} response always precedes the $(k+n)^{\text{th}}$ request (see Eq. 6).

$$(\forall k \in \mathbb{N}^{\star})\ tb_f[k] \prec tb_s[k+n] \tag{6}$$

This can easily be expressed in CCSL by combining the relation *precedence* with a *delay* as in Eq. 7.

$$tb_f \boxed{\prec} tb_s\ \$\ n \tag{7}$$

IV. Simulation semantics

A. VHDL Simulation Semantics

We briefly describe the *event-driven* simulation semantics of VHDL. Elaboration and execution of a VHDL model are specified in the VHDL Language Reference Manual [13, chapter 12]. *Elaboration* is the process by which declarations become effective. The elaboration of a VHDL design hierarchy results in a collection of processes interconnected by nets, named by the standard as *model*. This model can then be *executed* to simulate the design. A simulation consists of executions of interacting user-defined processes. These executions are coordinated by an event-driven *simulation kernel*, also known as the VHDL simulator.

The simulation comprises a sequence of *simulation cycles*. A global clock holds the *current simulation time*. This time is not decreasing and is incremented by discrete steps. Usually, several simulation cycles, called *delta cycles*, are executed at the same simulation time. The delta delay, which separates the successive delta cycles, is considered as an infinitesimally small interval of time. The issue is to determine when the simulation time has to (effectively) progress and when a delta cycle has to be performed.

A simulation cycle consists of two separate phases: active signal updating and processes executions.

1) Each active signal is updated. If this results in a change of value, then an event occurs on this signal.
2) For each process P, if P is currently sensitive to a signal on which an event has occurred in this simulation cycle, then P resumes and executes until it suspends. Actually only a *non postponed* process executes.

If any signal changes its value during the simulation cycle, time is not passing and a delta cycle is executed instead. In fact, the current time changes only after a "steady-state" is reached.

Because of the neat separation between updating and processing phases during a simulation cycle, the result of the simulation is deterministic: it does not depend on the order in which processes are executed.

B. Glitches in Design

Even if in VHDL delta cycles allow deterministic simulations, they may cause *glitches* (*i.e.,*a false or spurious transient signal variations). This is usual in combinatorial circuits and is generally harmless in synchronous circuits where only snapshots of signals are considered on the rising edge of a clock signal. This is not the case when the glitch triggers some visible effects. With a simple example we explain the relationship between delta cycles and glitches in VHDL. We then justify why the same circuit modeled in Esterel is free of glitch.

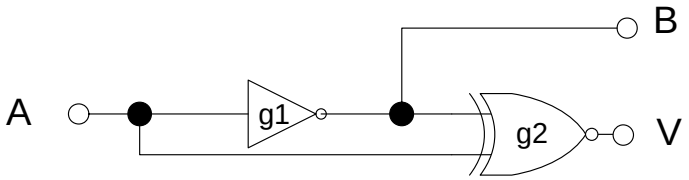

Fig. 3. Simple circuit with transient signal.

The circuit (figure 3) has one input signal A and two output signals B and V. The logical equations are:

$$B = \neg A$$
$$V = (A \Leftrightarrow B)$$

Therefore, $V = \big(A \Leftrightarrow (\neg A)\big) = false$, that is, V should always be '0'. This can be stated in VHDL with an *assertion*:

```
assert V = '0' report "Violation";
```

A straight modeling of the circuit in VHDL is:

```
1 entity CheckExclusion is port(
2    A:in Bit; B:inout Bit; V:out Bit);
3 end entity CheckExclusion;
4 architecture CExc of CheckExclusion is
5 begin
6   g1: B <= not A;
7   g2: V <= A xnor B;
8 end architecture CExc;
```

A simulation of this program detects a violation on a change of signal A, shown as a "0-width glitch" on V in the simulation trace. The expression "0-width glitch" means that zooming in the waveform will not increase the width of the glitch: its duration is 0, or more precisely one or several consecutive delta cycles.

Assume that just before instant t the circuit is in a steady state such that A='0', B='1', V='0'. At instant t, A changes to '1'. Table I explains the evolutions of the circuit according to the VHDL simulation semantics. Bold face figures indicate a change in the value. It appears that V has a transient '1' (the glitch).

	t^-	t	$t+\delta$	$t+2\delta$	$t+3\delta$
A	0	**1**	1	1	1
B	1	1	**0**	0	0
V	0	0	**1**	**0**	0

TABLE I
MICROSTEPS IN A SIMULATION INSTANT

0-width glitches could be avoided in VHDL if only steady-states were considered. VHDL'93 has provided a new facility that is useful in models with delta delays. The keyword **postponed** allows deferred executions of a process during delta cycles. While in an execution phase, a postponed process is not executed, even if it is currently sensitive to a signal on which an event has occurred. Instead, it waits for the end of the *last* delta cycle of the current simulation time to execute. Of course, a postponed process must not cause a new delta cycle. Postponing the concurrent equation $g2$ is enough to avoid the glitch. However, this can only be done because V is not in the sensitivity list of any other process, what might produce another delta cycle.

C. Microstep vs. Fixed point semantics

Now, consider an Esterel program for the same circuit:

```
1 module CheckExclusion:
2    input A;
3    output B, V;
4    sustain {
5      B if not A,
6      V if not (A xor B)
7    }
8 end module
```

The statement **sustain** { ... } (lines 4 to 7) is an infinite loop that executes its inner statements at each instant. The order in which the inner statements are written is irrelevant. The statements are conditional. B is emitted whenever A is absent. V is emitted whenever A and B are both either present or absent. Execution traces show no glitches.

The difference of behavior between the VHDL and the Esterel simulations is due to the different underlying semantics. Esterel does not rely on microsteps. It considers that the signal presence status is either unknown ($\perp$) or defined, which in turn can be either `present` or `absent`. The status of any signal during a reaction respects two *coherence rules*: in any reaction,

1) any signal is either `present` or `absent`; never both;
2) for any output or local signal, all the emit actions (*i.e.,*an action setting the presence status to `present`) must precede any test action on this signal.

As a consequence, it is forbidden for a signal (say V in our example) to have different status during a reaction. The compiler determines an execution ordering that respects the *coherence rules*. Here it computes B first, then V, so that no microsteps with delta cycles are needed.

Implementing observer modules in Esterel is easier than in VHDL, because of the absence of "transient" signal status. A library of adaptor, generator, and observer modules is available for Esterel [8].

D. Implementation in VHDL

In subsection II-C, we gave the principle of a component-based implementation of CCSL constraint observers. Here, we apply this principle to VHDL. Information (clock ticks) has to propage through the observation network before reaching the terminal node (an observer component). In this network, different paths with different lengths can cause glitches because of the microstep semantics. So, a naive implementation might detect *false* violations. The challenge was to devise *delta-delay insensitive* VHDL observers. This section describes our solution to this problem.

1) Adaptors: As explained in subsection II-C, an adaptor takes signals from the program-under-test and generates a new VHDL signal which represents a logical clock. This kind of signal is called a c_clock, and is a 'pulsed' signal whose pulses represent the clock ticks. The width of a pulse is ε (EPSILON). ε is strictly positive but 'as small as possible', *i.e.,*far smaller than the minimal duration ($\Delta_{\min}$) between application events. The standard allows a 1fs resolution, but the actual value, which depends on the simulator, is usually bigger. $\varepsilon > 0$ ensures that the rising and the falling edges of the pulse occur at different simulation time, *i.e.,*not within another delta cycle at the same simulation time. $\varepsilon \ll \Delta_{\min}$ makes that the pulse falling-edge is the simulation instant immediately following the pulse rising-edge. A pulse is easily generated in VHDL by assigning waveforms to a signal. Execution of

```
c_out <= '1', '0' after EPSILON;
```

produces a pulse whose width is EPSILON, a given constant typed Time.

In its simplest form, an adaptor takes a single input signal and generates a pulse on a particular (VHDL) event on this signal (*e.g.,*a rising-edge). The library provides adaptors for rising-edge and falling-edge. Sometimes, specific adaptors must be written. This was the case for the tranfer completion specified in figure 2. It implied a sequential behavior on two signals. The following VHDL adaptor code considers two logical input signals (level and clk) and uses a local counter (line 9). The pulse is generated (line 15), when level is maintained HIGH in two consecutive rising edges of signal clk.

```
1  entity Ccsl_A_TF is
2    port (level,clk: in std_logic;
3          c_out: out bit:= '0');
4  end entity Ccsl_A_TF;
5  architecture Ccsl_A_TF_arch of
6               Ccsl_A_TF is
7  begin
8    process(clk)
9      var cnt: Natural:=0;
10   begin
11     if (clk'event) and (clk = '1') then
12       if level = '1' then cnt := cnt+1;
13       else cnt := 0; end if;
14       if cnt = 2 then
15         c_out <= '1', '0' after EPSILON;
16         cnt := 0;
17       end if;
18     end if;
19   end process;
20 end Ccsl_A_TF_arch;
```

An instantiation of this adaptor has to specify the bindings between formal and actual ports. For instance, the c_clock `c_tbf` associated with the logical clock tb_f (figure 2) is driven by the instantiation:

```
a1: Ccsl_A_TF port map(
      level => PSEL, clk => CLK,
      c_out => c_tbf);
```

2) Observers: To avoid false violations due to glitches, we use postponed processes. Since a relation observer is always at the end of the observation network, the code of an observer can be executed at the very end of a simulation instant. This is illustrated on the (strict) precedence relation.

```
1  entity Ccsl_R_s_precedes is
2    port (c_a,c_b: in bit;
3          v: out bit:= '0');
4  end entity Ccsl_R_s_precedes;
5  architecture Ccsl_R_s_precedes_arch of
6    Ccsl_R_s_precedes is
7  begin
8    postponed process(c_a,c_b)
9      variable delta:integer := 0;
10   begin
```

```
11      if (delta = 0) and (c_b = '1') then
12        v <= '1'; -- violation
13      else v <= '0';
14      end if;
15      if c_a = '1' then
16        delta := delta + 1;
17      end if;
18      if c_b = '1' then
19        delta := delta - 1;
20      end if;
21    end process;
22  end architecture Ccsl_R_s_precedes_arch;
```

The specifications given in subsection II-B are implemented in the above program. The internal state δ is represented by variable *delta*, initialized to 0. The negation of the enabling condition (*i.e.,*the violation condition) is $(\delta = 0) \wedge b$. Lines 11 to 14 check this condition and set `v` accordlingly. Lines 15 to 20 maintain the internal state. The whole process is postponed (line 8), so that its code is executed when all signals are stable.

3) Generators: Since a generator is not a maximal element in the observation network, it cannot be implemented as a postponed process. The idea is to realize it as two separate processes. The first process, named `surface`, deals with the combinatorial behavior. This process drives the generator output c_clock and may introduce glitches. The second process, named `depth`, is sequential and manages the internal state. `depth` is a postponed process, thus it works only when the observation network has stabilized. The names 'surface' and 'depth' come from the synchronous language compilers that also separate combinatoiral and sequential evolutions. Figure 4 shows the internal structure of a generator. This is illustrated with the simple delay generator, which has one input c_clock `c_a`, a natural number input parameter `n`, and one output c_clock `c_c`.

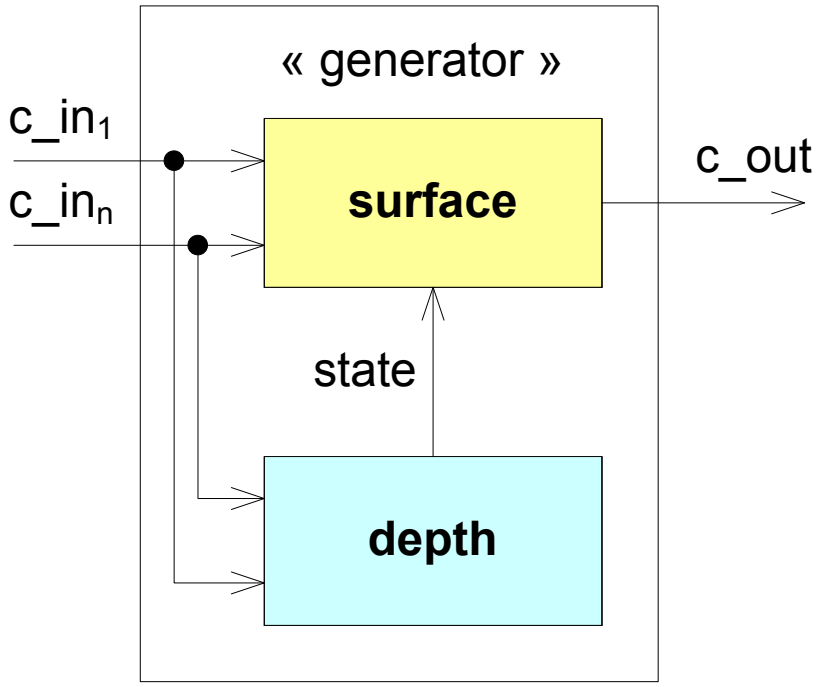

Fig. 4. Generator internal structure.

```
1  entity Ccsl_E_delay is
2    generic (n: POSITIVE := 1);
3    port (c_a: in  bit;
4          c_c: out bit:= '0');
5  end Ccsl_E_delay;
6  architecture  Ccsl_E_delay_arch of
7                Ccsl_E_delay is
8    signal bv:bit_vector((n-1) downto 0)
9                  := (others=>'0');
10 begin
```

```
11    surface: process (c_a)
12    begin
13      if bv(0) = '1' then
14        if c_a = '1' then
15          c_c <= '1', '0' after EPSILON;
16        else
17          c_c <= '0';
18        end if;
19      else
20        c_c <= '0';
21      end if;
22    end process;
23    depth: postponed process (c_a)
24    begin
25      if c_a = '1' then
26        bv <= '1' & bv((n-1) downto 1);
27      end if;
28    end process;
29  end Ccsl_E_delay_arch;
```

The local signal `bv` is declared and initialized at line 8. This signal is accessible by the two processes: `surface` for reading, and `depth` for reading and writing. Lines 13 to 21 implement the enabling condition (Eq. 3) checked in the `surface` process. The body of the depth process updates `bv` when `c_a` ticks as prescribed by Eq. 4.

V. Verification

A. Verification of CCSL Constraints in VHDL

In the presentation of the APB bridge (Section III-B), we specified two high-level properties about the data transfers through the bridge:

- *P1*: any APB bridge transaction is always as a result of a transaction initiation from the AHB bus. A causality relation expressed as a precedence CCSL constraint: $tb_s \boxed{\prec} tb_f$.
- *P2*: before the current bridge transaction is completed, at most one new request for bridge transaction can be sent by the AHB bus master. This is expressed in CCSL as $tb_f \boxed{\prec} tb_s \$ 2$.

We add a third property concerning the control flow through the bridge.

- *P3*: the APB bridge forbids access when its buffer is full. In CCSL this is represented by a logical clock ($full$) that ticks whenever the buffer gets full. At the RTL level, this results in setting the HREADY signal to low.

These properties are checked against the available VHDL model of a LeonII-based architecture, already mentioned in the introduction. Recall that the observation network directly reflects the abstract syntax of the clock constraint. As a consequence the implementations of the three properties are of increasing complexity.

1) Property 1: The observation network is very simple. It consists of three components: two adaptors and a precedence observer. It is easily programmed in VHDL. It assembles instantiations of components from our library (the `Ccsl_A_RisingEdge` adaptor, the `Ccsl_R_s_precedes` observer), and the dedicated adaptor `Ccsl_A_TF` proposed in subsection IV-D. The simulation of the LeonII system raises no violation.

```
1  entity P1_Obs is
2    port(CLK, HSEL_B, PSEL: in std_logic;
3         v1: out bit:= '0')
4  end entity;
5  architecture P1_ObsNet of P1_Obs is
6  component Ccsl_A_RisingEdge is
7    port (s: in std_logic;
8          c_out: out bit:= '0');
9  end component;
10 component Ccsl_A_TF is
11   port (level,clk: in std_logic;
12         c_out: out bit:= '0');
13 end component;
14 component Ccsl_R_s_precedes
15   port (
16     c_a,c_b: in bit;
17     v: out bit:= '0');
18 end component;
19 signal c_tb_s: bit := '0';
```

```
20  signal c_tb_f: bit := '0';
21  begin
22    a1:Ccsl_A_RisingEdge port map(
23          s=>HSEL_B, c_out=>c_tb_s);
24    a2:Ccsl_A_TF port map(
25          level=>PSEL, clk=>CLK,
26          c_out=>c_tb_f);
27    o1: Ccsl_R_s_precedes port map(
28           c_a=>c_tb_s, c_b=>c_tb_f,
29           v=>v1);
30  end P1_ObsNet;
```

2) Property 2: The clock constraints P1 and P2 are similar in their structure, except that P2 also requires a `Ccsl_E_delay` generator. The observation network has the same overall structure with an additional local signal `tb_s_d2` that represents the clock $tb_s \$ 2$, and an instantiation of the CCSL delay component:

```
g1: Ccsl_E_delay
  generic map(n=>2)
  port map(
    c_a=>c_tb_s,
    c_c=>c_tb_s_d2);
```

The detailed code is omitted. P2 also successfully passed the simulation.

3) Property 3: This property demands a subtler CCSL specification. Figure 5 shows a possible (correct) evolution of the system. $\#s$ stands for the number of occurrences of s in the run. We can see that the buffer gets full whenever $tb_s[j+1] \prec tb_f[j]$. This is the case at point F where the index of tb_s is 5, and the 'still to occur' index of tb_f is (or more exactly, will be) 4. Hence, we can reformulate the saturation of buffer in terms of logical clocks as:

$$\begin{array}{l}(\forall j \in \mathbb{N}^\star)(\exists k \in \mathbb{N}^\star) \\ (tb_s[j+1] \prec tb_f[j]) \Leftrightarrow (full[k] \equiv tb_s[j+1])\end{array} \tag{8}$$

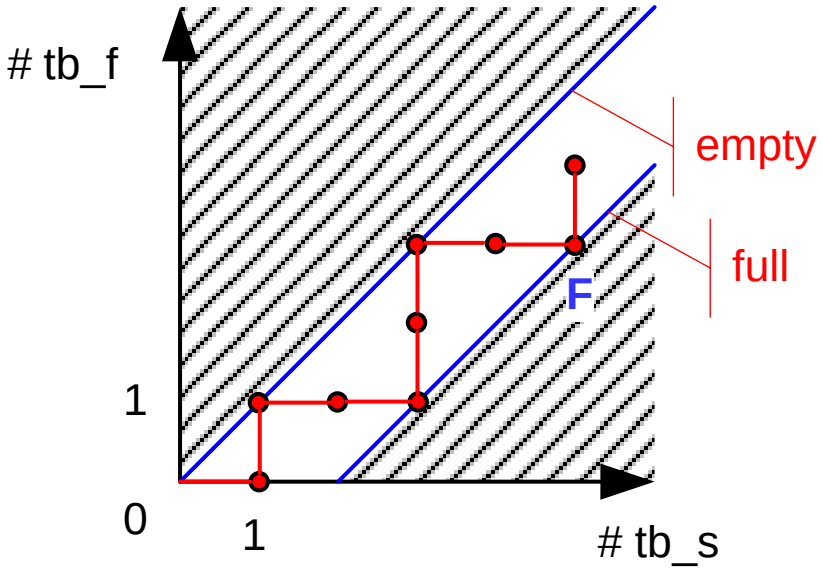

Fig. 5. A possible correct evolution.

From this specification, one can derive a CCSL specification (see [14]):

$$full \boxed{=} \left((tb_s \$ 1) \wedge tb_f\right) - tb_f \tag{9}$$

The expression $a \wedge b$, where $\wedge$ denotes the CCSL *inf* operator, defines the slowest clock among all the clocks faster than a and b. The expression $a - b$, where $-$ denotes the CCSL *minus* operator, defines a clock that ticks in coincidence with a whenever b is not coincident with a. For convenience, we can define two auxiliary clocks: $tb_s\_d1 \triangleq tb_s \$ 1$ and $first \triangleq tb_s\_d1 \wedge td_f$, so that Eq. 9 can be rewritten as $full \boxed{=} first - td_f$. Figure 6 shows an example of execution that respects P3.

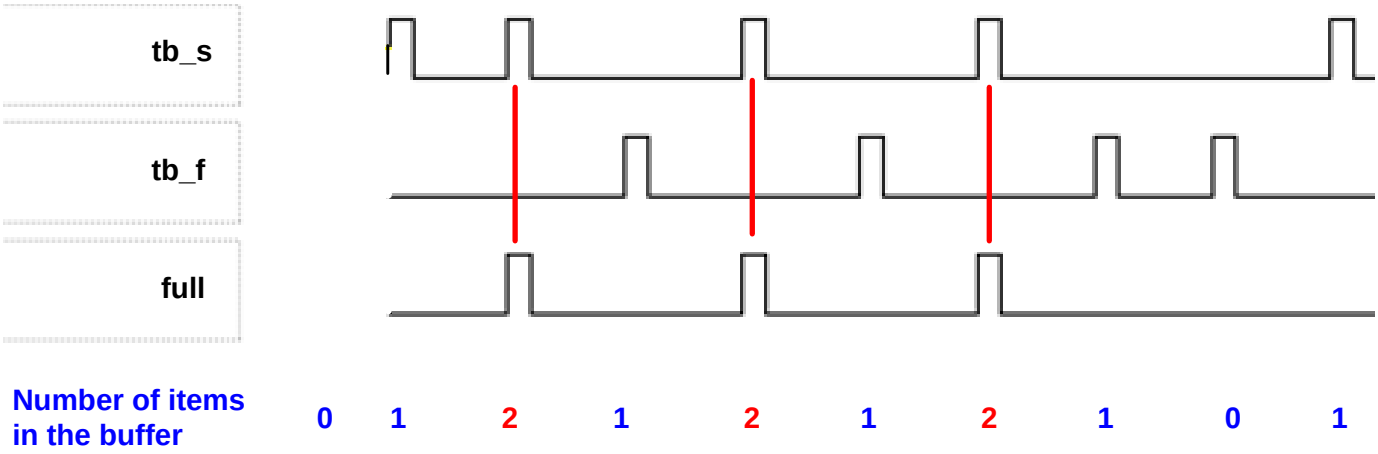

Fig. 6. Sample Execution of Constraint 3 on CCSL Simulator

The question is now how to relate this observation to the APB bridge behavior. An AMBA slave (*e.g.,*the APB bridge) indicates to its master that it is ready to accept transfers by asserting the signal HREADY. So, when the bridge buffer gets full, the bridge drives signal HREADY to low on the *next* bus cycle. Hence, in the observation network (figure 7), we had to delay `c_full` for 1 instant of `c_clk`. This is done by a *defer* generator. Now, since the saturation is manifested by a low level on signal HREADY, we used a *fallingEdge* adaptor to sense HREADY. `c_full_d1` and `c_invhready` are then observed to be coincident. The simulation of the LeonII-based architecture detected a violation of P3. We then inspected the VHDL code of the APB bridge component. It appeared that this implementation did not take account of the buffer size.

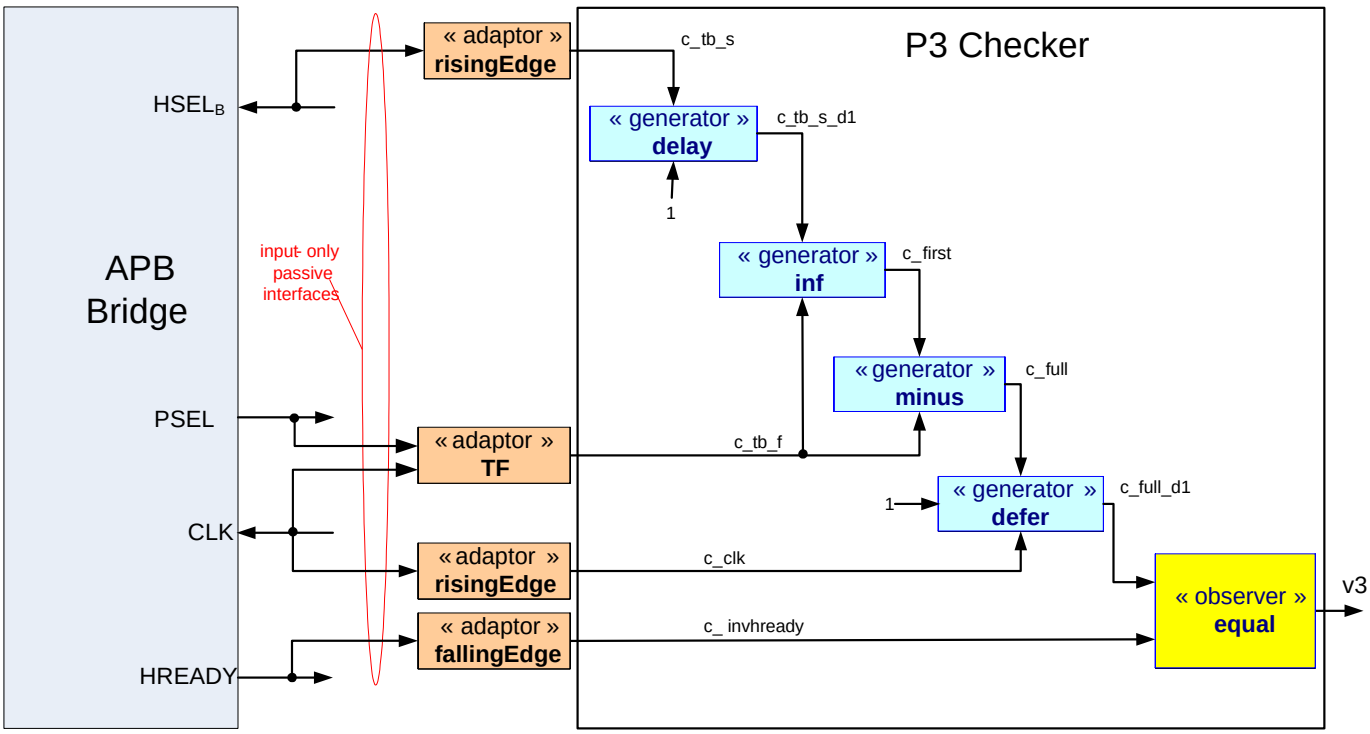

Fig. 7. Observation of property 3.

VI. Discussion and Related Works

A. Semantic issues

The microstep semantics of VHDL may generate glitches (subsection IV-B). The main issue for us has been to develop *delta-delay insensitive* VHDL components. Delay insensitive or speed independent circuits have been intensively studied in

the 80's. In these circuits, the delay on a signal path does not affect circuit behavior. More recently, the concept of latency insensitive design [15] has been introduced for electronic system with a global clock. To our knowledge, these studies have not been applied to VHDL code writing.

VHDL was originally designed for discrete-time discrete-event simulation of digital circuits. The underlying semantics is the VHDL *simulation semantics* used in this paper. A restriction to the language, known as *synthesizable subset* of VHDL [16] was introduced for use in digital logic synthesis. This subset relies on the *synthesis semantics*. Unfortunately mismatches exist between the simulation and the synthesis semantics. Formal hardware verification tools consider the latter. Our solution contains non synthesizable statements like the **postponed** (to wait for stability) and the assignment of waveforms (to generate pulses). This confines our approach to simulation and might seem to discard formal analysis. However, Buck and al. [17] have proposed a formal model construction using HDL simulation semantics. They have defined a behavioral equivalence on sample points (called synchronization points). This allows a symbolic simulation of HDL constructs based on the simulation semantics. Their formal model is consistent with simulation at the specified synchronization points. The instants of our logical clocks play a role similar to the synchronization points. However our solution is more abstract: clock constraints are logical constraints that do not check actual data values.

Our goal is to reuse standard VHDL-based design environments. When synthesis is required, our observer-based approach remains valid and still allows formal and exhaustive verification with static analysis [7]. Synchronous languages like Esterel [18], when used to design circuits, have solved the issues of glitch by doing a static (modular) analysis of the code followed by a topological sort of the processes. The discrete-event domain [19] of Ptolemy uses similar techniques too. Such approaches require the knowledge of the whole system, to detect possible causality loops and compute the dependency graph. It also requires to develop a dedicated environment including a compiler, and a simulator. Our case is much simpler since we have trees (or DAGs) and can therefore be modular. However, the modular compilation of Esterel is not a solution here. Indeed, the compiler would only solve the dependency problems within the modules but would not be able to solve inter-module races. A compositional modular synthesis of VHDL entities would require a dedicated simulator based on similar techniques.

B. Usage of CCSL

CCSL appears in this paper as yet another formalism to express logical time safety properties. One could claim that existing languages such as temporal logics or PSL [20] are sufficient for expressing such properties. However, CCSL diverges from PSL on two points: expressiveness and integration with system-level model-based environments.

CCSL can only express safety properties whereas PSL can also express liveness properties. Even though, most CCSL relations can be encoded by LTL formulas, some relations (like the precedence) that introduce unbounded parameters cannot be encoded in LTL or CTL.

PSL is already integrated in some of the existing VHDL development environments and is consequently well adapted for this low abstraction level. Conversely, CCSL has been used several times to specify logical time property of a system since the very first stage of the design (*i.e.,*during modeling). Developed conjointly with the UML profile for MARTE, it directly relies on model information (UML/SysML). In this paper, we have detailed how to use a CCSL specification for the automatic generation of VHDL observers, linking together high level specifications and low level analyses. The observers are then used together with an actual implementation to verify if the implementation satisfies the specification. As highlighted in section V this approach has helped us find a bug in an implementation of the AMBA bridge delivered by the Spirit Consortium, demonstrating the usefulness of this approach.

VII. Conclusion

We have shown how CCSL can be used to capture *safety properties* that can be exploited to verify existing VHDL implementations. We have discussed some general principles that govern the construction of property observers. The main difficulty—compared to our previous contribution with the Esterel language—is to bridge the semantics gap between the fixed-point semantics of CCSL and the microstep simulation semantics of VHDL. We have built a library of *delta-delay insensitive* VHDL components including adaptors, generators and observers, which implement the operational semantics of CCSL relations and expressions. Based on this library, we propose an automated generation of an observation network by a structural transformation. Only adaptors have to be written by hand since they are abstracted away by the logical view of the CCSL specification. A library of very often used adaptors is also available.

Our implementation uses *postponed processes* to avoid glitches that might be avoided by using a three-valued logic as in the constructive semantics of Esterel. Avoiding postponed processes is key to adapt our proposal to SystemC, which does not provide an equivalent mechanism. A plug-in is available on our site (http://www-sop.inria.fr/aoste/dev/time_square/observers/). It currently supports automated generation of observer networks in VHDL and Esterel.

A longer term perspective would be to investigate the construction of possibly intrusive observers—also known as supervisors—based on CCSL. Such a capability would have an application to design fault tolerant systems with dynamic adaptions and monitoring.

REFERENCES

[1] OMG, *UML Profile for MARTE, v1.0*, Object Management Group, Nov. 2009, OMG document number: formal/09-11-02.

[2] C. André, F. Mallet, and R. de Simone, "Modeling time(s)," in *MoDELS*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 559–573.

[3] C. André, "Syntax and semantics of the clock constraint specification language (CCSL)," INRIA, Research Report 6925, 05 2009. [Online]. Available: http://hal.inria.fr/inria-00384077/en/

[4] E. A. Lee and A. L. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, December 1998.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[6] J. Axel and I. Sander, "Models of computation and languages for embedded system design," *IEE Proceedings on Computers and Digital Techniques, Special issue on Electronic System Design*, vol. 152, no. 2, pp. 114–129, March 2005.

[7] C. André and F. Mallet, "Specification and verification of time requirements with CCSL and Esterel," in *LCTES*, C. M. Kirsch and M. T. Kandemir, Eds. ACM, 2009, pp. 167–176.

[8] C. André, "Verification of clock constraints: CCSL observers in Esterel," INRIA, Research Report 7211, 02 2010. [Online]. Available: http://hal.inria.fr/inria-00458847/en/

[9] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*. London, UK: Springer-Verlag, 1994, pp. 83–96.

[10] L. Aceto, A. Burgueño, and K. G. Larsen, "Model checking via reachability testing for timed automata," in *TACAS*, ser. Lecture Notes in Computer Science, B. Steffen, Ed., vol. 1384. Springer, 1998, pp. 263–280.

[11] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis, "Testing conformance of real-time applications by automatic generation of observers," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 23–43, 2005.

[12] B. Ben Hedia, F. Jumel, and J.-P. Babau, "Formal evaluation of quality of service for data acquisition," in *FDL*. ECSI, 2005, pp. 579–589.

[13] IEEE, *IEEE Standard VHDL Language Reference Manual*, IEEE Standard, 2000, IEEE Std 1076a-2000.

[14] A. Mehmood Khan, F. Mallet, C. André, and R. de Simone, "IP-XACT components with abstract time characterization," in *Forum on specification, verification and design languages, FDL 2009*, September 2009.

[15] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *ICCAD*, J. K. White and E. Sentovich, Eds. IEEE, 1999, pp. 309–315.

[16] IEEE, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, IEEE Standard, 2004, IEEE Std 1076.6-2004.

[17] J. Buck, D. Wang, and Y. Zhu, "Formal model construction using hdl simulation semantics," *High-Level Design, Validation, and Test Workshop, IEEE International*, vol. 0, pp. 115–122, 2007.

[18] G. Berry, "The foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, C. S. G. Plotkin and M. Tofte, Eds. MIT Press, 2000.

[19] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Ann. Software Eng.*, vol. 7, pp. 25–45, 1999.

[20] IEEE, *IEEE Standard for Property Specification Language (PSL)*, IEEE Standard, 2005, IEEE Std 1850-2005.