Modelling VHDL in *Multiclock* ESTEREL*

Basant Rajan† and R.K. Shyamasundar
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay 400 005, India
e-mail: {basant,shyam}@tcs.tifr.res.in

Abstract

The introduction of HDLs (*hardware description languages*) have made a significant contribution to VLSI circuit design. While these languages are well suited to describe circuits in great detail, they are found wanting when attempting formal verification of circuits. One of the drawbacks lies in the lack of formal models of semantics. VHDL, with a wide variety of tools available to carry out circuit simulations, is the *de facto* standard used in VLSI design. On the other hand, the family of *synchronous languages* has been well studied for the design and synthesis of provably correct reactive systems. Thus, it behooves one to explore the possibility of interfacing languages such as VHDL with synchronous programming languages to handle higher levels of abstraction in circuit design with assured correctness. In this paper, we show how the paradigm of *Multiclock* ESTEREL provides a framework for the design of multi-clocked systems and asynchronous systems. *Multiclock* ESTEREL can be used for modeling in conjunction with VHDL to enable formal verification of circuit behavior. We shall also show that *Multiclock* ESTEREL captures the VHDL timing model succinctly.

Multiclock ESTEREL can be used either in conjunction with VHDL or as a replacement depending on the requirements at hand. When used in conjunction, *Multiclock* ESTEREL can be used to describe circuits at higher levels of abstraction while VHDL can be used to describe the signal propagation characteristics.

1 Introduction

VLSI circuit design has gained considerably from the introduction of *hardware description languages* (or HDLs). While these languages are well suited to describe circuits in great detail, they are found wanting when attempting formal verification of circuits.

*Partially supported by the Indo-French Centre (CEFIPRA) project 2202-1: *Formal Specification and Verification of Reactive Hybrid Systems*.

†VERITAS Software (India) Pvt. Ltd., Pune 411020 India

Given the wide variety of tools that exist to carry out simulations for circuit specifications in VHDL, it is the *de facto* standard used in the VLSI industry. There has however been significant momentum in the industry to direct efforts at hardware/circuit verification rather than just validation. Keeping this in view, as a first step, there have been several efforts in providing a formal semantics for various subsets of VHDL containing reactive and timing features. The efforts in this direction have been two fold:

- Provide functional models that are suitable for execution in one of the several programming languages, and
- Provide models that enable interfaces to theorem provers and model checking systems.

One of the major issues in the models for VHDL has been underlying timing models and the zero-delay hypothesis. In this paper, we shall explore the semantics of a subset of VHDL in terms of the family of synchronous languages that have been used widely for the specification and design of provably correct reactive systems These languages have been based on the synchrony hypothesis which can be informally captured by the following statement:

> There is no observable time between the inputs and the corresponding outputs.

The above hypothesis closely corresponds to the underlying hypothesis in VHDL. Keeping such a view in mind, we started looking at the semantics of VHDL in terms of *Multiclock* ESTEREL – a synchronous language built on one of the oldest and widely used imperative synchronous languages, ESTEREL. In fact, programs in ESTEREL can be directly translated to hardware circuits [1]. Using *Multiclock* ESTEREL we have derived the additional benefits from the freedom to use multiple clocks and asynchronous designs. In this paper, we show how the distinct features of VHDL can be captured in *Multiclock* ESTEREL. Such a modeling will allow for formal verification using the verification environments of ESTEREL. Furthermore, it provides an interfacing to handle higher levels of abstraction in circuit design[2].

Multiclock ESTEREL could be used either in conjunction with VHDL or as a replacement depending on the requirements at hand. When used in conjunction, *Multiclock* ESTEREL can be used to describe circuits at higher levels of abstraction while VHDL can be used to describe the signal propagation characteristics. On the other hand, it is conceivable that whole circuits can be described entirely in *Multiclock* ESTEREL. Translations of VHDL into *Multiclock* ESTEREL could be used to prove behavioral characteristics of a circuit or of a component thereof.

The rest of the paper is organized as follows: Section 2 presents a very brief overview of formal semantics models for subsets of VHDL found in contemporary works. An informal description of *Multiclock* ESTEREL follows in Section 3. Section 4 discusses the semantics of VHDL in the context of *Multiclock* ESTEREL. Then a few examples are used to illustrate the approach followed by the conclusion.

2 Formal Models for Subsets of VHDL

In this section, we give a brief overview of some of the efforts at providing formal semantic models for VHDL.

Belhadj [3] present an attempt to integrate VHDL with SIGNAL. Given the variation in the scope of the two languages, the attempt involved defining a subset of both SIGNAL and VHDL and providing a translation between them. The approach also sought to permit descriptions in a mixture of the two languages. Time scales in VHDL were simulated in SIGNAL by mapping macro-time and micro-time in VHDL to instants in SIGNAL and filtering to generate appropriate masking.

Fuchs *et al.* [4] presented a functional semantics for delta-delay VHDL using `Focus` based on the concept of streams and stream processing functions. Their approach differs primarily from other attempts in that they use a direct approach as opposed to providing a semantics through translations into other well defined languages. The approach treats VHDL as a specification language that is used at an abstract level. Such a specification is then interpreted in terms of streams and operations on them which are amenable to formal analysis in the `Focus` environment. A significant drawback however is the inability to reference future instants and model transport/inertial delays explicitly

Breuer *et al.* [5] provide a functional semantics for *unit delay* VHDL. The restriction imposed is that the subset of VHDL considered precludes instantaneous signal assignments or *delta-delayed* signals and zero delay waits. The semantics developed is however compositional and though declarative in style and concept also lends itself to execution that closely parallels a VHDL simulator.

Wolinski *et al.* [6] have explored the possibility of designing systems that are globally asynchronous and locally synchronous. Their work concentrated on synthesis of such systems from the high level description language SIGNAL. The method works by generating a net of processes with a synchronous data manipulation component and an asynchronous clock manipulation component that communicate over event synchronized channels.

3 An Introduction to *Multiclock* ESTEREL

Multiclock ESTEREL is based on the synchronous language ESTEREL which in turn is based on the *perfect synchrony hypothesis*. The hypothesis can be interpreted to mean that outputs are produced synchronously with the inputs. In other words, reactions to inputs happen in *zero time* (in the sense that the time is not observable in this context). Further, there is no explicit clock associated with ESTEREL. *Multiclock* ESTEREL is a language designed to support the explicit specification of clocks as is usual in the specification of hardware circuits. The perfect synchrony hypothesis has been adapted to the explicit specification of clocks in the sense that while in ESTEREL the reactions to external events were just controlled by the environment (which indirectly defined the “clock” of the response), in *Multiclock* ESTEREL, reactions are determined by the clocks controlling the current execution by appropriately abstracting the input from the environment and reacting to it in the form of outputs (again controlled by the appropriate clocks). With such an adaptation of linking reactions to the current clock, *Multiclock* ESTEREL can also be considered to satisfy the synchrony hypothesis.

The syntax and form of *Multiclock* ESTEREL closely resembles that of its precursor, ESTEREL. The general structure of a *Multiclock* ESTEREL program consists of a collection of named modules composed using the parallel or sequence operator. Each of these modules has an explicitly specified clock associated with it.

Each module in turn is built of either sub-modules or simple statements composed using either the parallel or sequence operator. *Multiclock* ESTEREL is for the most part block structured but supports explicit access control that *limits* the scope of objects. All variables and signals need to be declared before use as is the case in ESTEREL. Variables cannot be shared across modules composed in parallel.

Multiclock ESTEREL supports a simple complement of data types that includes *integers*, *floating point* numbers and *character strings* in addition to *signals*, *latches*, *clocks* and *sensors*.

While variables of simple types like *integers* are valid only in the modules defining them, signals can be shared across modules. Signals are used as an interface to communicate with the external world. Signals are distributed using an instantaneous broadcast mechanism just as in other synchronous programming languages like ESTEREL, SIGNAL *etc.*. This in turn implies that the statement emitting a sig-

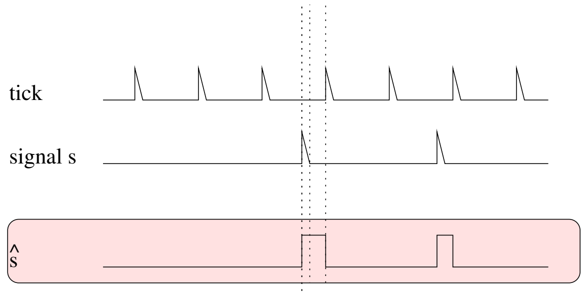

Figure 1. Clock, signal and latches

nal will see the signal in its current execution environment. Sensors are a special kind of signal that can be queried at any time to obtain a value. They do not contribute to the signal environment in as much as they do not generate events.

Since every statement is bound to a clock, there is obviously a need to provide some amount of limited memory *in the form of latches* to interface the asynchronous environment to the reactive core. We shall describe latches in section 3.1, expressions in section 3.2 followed by characteristic features of *Multiclock* ESTEREL in section 3.3.

3.1 Latched Signals

The capability of *Multiclock* ESTEREL to explicitly bind *clocks* to specific signals or signal expressions introduces certain problems. For instance, how does one treat events that occur *out of sync* with the current *tick* signal?

In the following, we present the form of latch used in *Multiclock* ESTEREL(refer to [7] for other possible notions).

Figure 1 shows the relationship between a clock, a signal and the latched version of the signal as used in *Multiclock* ESTEREL . The waveform labeled *tick* is assumed to be the current clock. It is not necessary that the clock be periodic as shown. The waveform labeled *signal s* represents an arbitrary signal occurring in the environment of a *Multiclock* ESTEREL module. $\hat{s}$ denotes a signal that sustains the value of s until the *next* clock tick. The interval for which $\hat{s}$ sustains the signal *s includes* the next clock *tick*. In particular, $\hat{s}$ generated for any given event will never span *two* clock *ticks*. This in turn implies that if s occurs synchronously with a clock tick, its value will not remain latched until the next instant.

Latching serves to implement *limited* memory in asynchronous environments. In particular, *latching*, as it is used in the sequel serves only to hold the *last* value of a signal until the next clock tick. *Multiclock* ESTEREL uses a technique similar to that used above to deal with signals that occur out of synchronization with the current clock. The use of *latched* signals makes interfacing modules/statements with different clocks easy. Signals may never be accessed using disparate clocks, but the *latched* versions of these signals can be read using any clock. Therefore parameter passing between modules with no common clock can be effectively accomplished using latched signals.

3.2 Expressions

Multiclock ESTEREL supports two kinds of expressions. The first kind ranges over signals and are referred to as *clock* expressions. Clock expressions are boolean functions over signals and cannot contain references to variables or sensors. They are used to specify bindings for the local clock or *tick*. The second type of expression ranges over variables as well as *latched* version of signals and are referred to as signal expressions. These expressions are what *Multiclock* ESTEREL code can test for as part of its execution sequence. Note that clock expressions can only be used to bind clock signals. *Latches* of signals are derived entities. All references to signal values are implicitly made to reference the corresponding latch.

3.3 Characteristics of *Multiclock* ESTEREL

Some of the distinguishing features of *Multiclock* ESTEREL that set it apart from its predecessor ESTEREL are discussed below. *Multiclock* ESTEREL allows the specification of individual clocks for modules. ESTEREL programs in contrast allowed only for a single execution clock. By relaxing this requirement, *Multiclock* ESTEREL can now address a wider class of programs without sacrificing modularity. *Multiclock* ESTEREL is truly modular, even when multiple clocks are used. ESTEREL's support for suspension and tasks was never modular in the true sense. By integrating the concept of asynchronous task execution along with the multiple clocks that *Multiclock* ESTEREL allows, *Multiclock* ESTEREL has managed to preserve modularity. *Multiclock* ESTEREL allows for asynchronous input events. Having allowed multiple clocks to govern the execution of different modules in the program and provided for interaction amongst them, *Multiclock* ESTEREL was faced with no difficulty in dealing with asynchronous external events either. *Multiclock* ESTEREL implements internal latching of all inputs.This feature serves to integrate seamlessly the domain of asynchronous events with the realm of synchronous execution semantics. The net effect of this feature is to introduce independent and minimal delays in all of the input channels to allow for the creation of a synchronous interface for the underlying module.

The kernel statements in *Multiclock* ESTEREL along

Multiclock ESTEREL *Statement*	*Intuitive Semantics*
`nothing`	*Does nothing and terminates instantaneously*
`halt`	*Never terminates*
`emit` s	*emit signal s*
$p;q$	*synchronous sequencing; q starts in the terminating instant of p*
`loop` p **`end`**	*nonterminating loop; p should last at least one instant*
`present` s **`then`** p **`else`** q **`end`**	*Checking presence/absence of s*
$p \parallel q$	*Synchronous parallelism; terminates when both terminate*
`signal` s **`in`** p **`end`**	$p \setminus s$
`abort` p **`when`** s	*Strong preemption in the next instant onwards*
`abort` p **`when immediate`** s	*Strong preemption from current instant onwards*
`trap` t **`in`** p **`end`**	*Weak preemption for exceptions (t)*
`exit` t	*exit on trap or exception t*
`begin` p **`end`**	*Block statement*
`suspend` p **`when immediate`** s	*suspends p on signal s from current instant onwards*
`newtick` s **`in`** p **`end`**	*binding clock s for statement p*

Table 1. Syntax of *Multiclock* ESTEREL

with its intuitive meaning are given in Table 1; some of the constructs are described below (for details refer [7]).

The behavior of the *present* construct is to test for the presence of signal *s* and either execute statement *p* or statement *q* depending on whether signal *s* was present or absent. Both the test and the transfer of control to either statement *p* or statement *q* are done synchronously with the current clock.

The *abort* statement executes *p* synchronously with the initiation of the construct and terminates either when signal *s* occurs or when statement *p* terminates. Note that statement *p* will *always* terminate on the clock of the **`abort`** statement enclosing it. Of particular interest is the case when signal *s* is present at the instant when the construct is initiated. In this case, statement *p* is executed and preemption is carried out only at the next occurrence of signal *s*. Further, if signal *s* occurs during the execution of statement *p*, all processing scheduled for the current instant by statement *p* is abandoned.

The *suspend* construct executes statement *p* synchronously when it starts and terminates only when statement *p* does. However, statement *p* is prevented from executing whenever signal *s* is present. During this period however, changes to the environment are recorded. Execution of statement *p* resumes when signal *s* is no longer present.

The completely new construct newtick is briefed below. This statement starts execution synchronously at the current clock and initiates execution of statement *p* on clock *s*. Statements in *p* do not start executing until clock *s* occurs. All statements in *p* see signals in their environment through values latched on the new clock *s*. The construct terminates synchronously with the first occurrence of the current clock after the termination of statement *p* which in turn terminates synchronously with the new clock *s*. Further, statements in statement block *p* are insulated from any suspensions imposed on this construct. For all practical purposes, statement *p* can be considered an independent process.

4 Semantics of VHDL in *Multiclock* ESTEREL

In this section, we shall consider modeling of the timing, delays, zero-delay assignment, logical and reactive features. We shall not explicitly describe the subset for lack of space.

4.1 The VHDL timing model

VHDL semantic is specified by the language reference manual in terms of an event driven simulator. The two phase simulation cycle starts from a known state of variables and signals. In the first phase, all active processes update all local variables and signals until they terminate yielding new states for the variables and transaction traces for the signals. In the second phase, the transaction traces are translated into a new global signal state using drivers and resolution functions.

This leads to VHDL using two distinct notions of time, real or macro time and delta or micro time. Macro time, measured in units as small as femto seconds define points in time where observations about the state of the simulation can be made by an external observer while micro time deals with the succession of strictly synchronous events that occur internally to the simulator while computing the next observable state from the current one. Intervals between points in delta time have no significance and cannot be measured. Further, there is no *a priori* limit on the number of micro time points that can exist between successive macro time points.

All variable assignments in VHDL are made synchronously with micro time, while signal values change synchronously with macro time. In particular, this implies that while changes to signal values can be *scheduled* for later, variables have to be modified immediately.

In contrast, *Multiclock* ESTEREL admits only one notion of time, that defined by a succession of external events where the interval between any two successive events has no significance. Nevertheless, the *Multiclock* ESTEREL engine does use an implicit notion of *micro time* to execute sequential statements. This makes for a very convenient embedding of micro time in VHDL into a similar notion in *Multiclock* ESTEREL. Another significant difference is that VHDL can specify delays in real time while *Multiclock* ESTEREL cannot make explicit references to elapsed time. Further, *Multiclock* ESTEREL can not express any condition involving the (bounded) future while VHDL can and we will be dealing with this aspect in more detail in the sequel. More specifically, while *Multiclock* ESTEREL can await events in the future, it cannot spontaneously react to the passage of time.

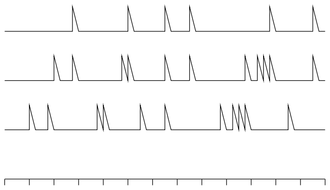

Figure 2. Transport and Inertial Delays

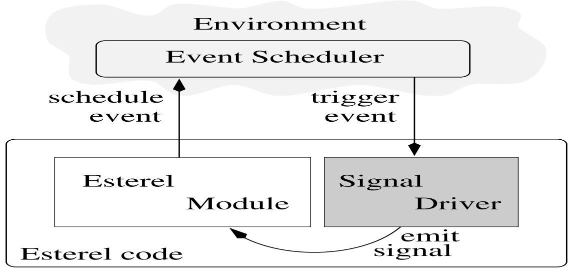

Figure 3. Event Scheduling

4.2 Interfacing VHDL and *Multiclock* ESTEREL

In this section we will present how concepts from VHDL and *Multiclock* ESTEREL can be interfaced for purposes of translation from the former into the latter. We will now refine the problem into how we can implement the various classes of objects in VHDL using *Multiclock* ESTEREL.

Signals form an important part of both VHDL and *Multiclock* ESTEREL. However their treatment in both these languages differ significantly. In VHDL, signals are represented as a flow of values using an explicit time metric. It is considered to hold its value between two events. In *Multiclock* ESTEREL, signals are represented as a stream of values without reference to an explicit clock.

For purposes of translations, this difference presents no difficulties, because both VHDL and *Multiclock* ESTEREL assign to signals synchronously in *micro* time while they are observable only in *macro* time.

While both languages support variables, and their treatment is similar in both environments, there are subtle differences in the support provided. For instance, *Multiclock* ESTEREL does not allow the scope of variables to encompass parallel modules. However both VHDL and *Multiclock* ESTEREL preserve the notion of synchronous updates to variables thus making translation easier. Difficulties in the scoping of variables could be handled by the use of signals that distribute the value of a variable.

VHDL allows the expression of explicit delays in signal assignments while *Multiclock* ESTEREL prohibits any reference to explicit time in keeping with the synchrony hypothesis. In particular, VHDL is capable of modeling two kinds of delays (*c.f.* Figure 2) *viz.*

- *transport delay* - Transport delay models delays involved in signals propagation along wires where changes (of potential) applied at one end are faithfully (subject to attenuation and other related phenomena) transferred to the other end after a small (but non zero and usually fixed) delay. Figure 2 shows a signal subjected to unit transport delay. Note that the waveforms are identical but subject to translation in the time axis.

- *inertial delay* - Inertial delay models the property of certain classes of circuits that require the inputs to be stable for a finite time before any corresponding changes to the outputs become observable. Figure 2 shows a signal subjected to unit inertial delay. Note that the waveforms are no longer identical and that variability of the delayed signals has reduced.

As is obvious, the inability to model either of these would severely hinder the applicability of *Multiclock* ESTEREL in specifying digital circuits.

The way we choose to handle this requirement within the *Multiclock* ESTEREL framework is as follows. To every output signal that will be involved in delayed updates, we associate a driver. Further, we associate an event scheduler with the execution environment of the Multiclock Esterel; the configuration is shown in Figure 3.

The function of the event scheduler is to implement delays outside the *Multiclock* ESTEREL environment. Delayed updates to a signal are posted as event requests by the *Multiclock* ESTEREL module to the event scheduler. After the specified delay period has elapsed, the event scheduler generates a *trigger event* in response to which the signal driver synchronously emits the signal with the updated value.

The event scheduler maintains a separate queue for each signal it is required to handle. Using the same underlying mechanism, we implement both transport delays and inertial delays in the following manner.

- Signals with transport delays associated with them are maintained as ordinary FIFO queues. This has the effect of triggering the event generator at the appropriate times to faithfully generate a delayed stream of values on the corresponding signal.

- Signals with inertial delays associated with them are maintained slightly differently. If conflicting requests are posted to the event scheduler for a given signal for instants separated in time by less than the inertial delay in effect, the earlier request is canceled. This in turn will suppress the generation of trigger event cor-

```
wait on EVENT for TIMEOUT
```

in VHDL becomes

```
abort
    newtick EVENT
        await tick
    end
when TIMEOUT
```

in *Multiclock* ESTEREL.

Figure 4. Implementation of `wait`

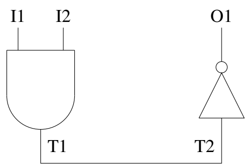

Figure 5. A NAND Gate Implementation

responding to the canceled request, thereby ensuring that no change to the associated signal is effected.

VHDL provides for signal attributes like an *event* which is true whenever the signal it is associated to changes value. In *Multiclock* ESTEREL however, while no special construct is made available to provide the same functionality, a control signal emitted only when the value of a signal changes can be used instead to model VHDL events.

The wait statement in VHDL translates readily into an *await* statement running inside a *abort* construct as shown in Figure 4.

Example 4.1 Consider an implementation of a NAND gate as shown in Figure 5. Let us assume that each of the logic gates introduces an inertial delay of 5ns and that the wire connecting *T1* to *T2* introduces a transport delay of 2ns.

- The implementation in *Multiclock* ESTEREL will assume that the input signals *I1* and *I2* have finite variability between any two instants of time. Without this assumption, the design will run into problems associated to instability.
- The AND gate shown will be clocked on the logical signal $I1 \wedge I2$ while the NOT gate will be clocked on *T2*
- The AND gate can be modeled by the code snippet

```
newtick I1 OR I2 in
  loop
    await tick;
    emit  sched-inertial-5ns-T1(?I1 AND ?I2)
  end
end
```

The above code ensures that whenever either of the inputs *I1* or *I2* changes, a signal is synchronously sent to the event scheduler with the intended value of the signal *T1*.

- Similarly the NOT gate can be modeled by the code snippet

```
newtick T2 in
  loop
    await tick;
    emit sched-inertial-5ns-O1(NOT ?T2)
  end
end
```

Here we ensure that whenever the value of the temporary signal *T2* changes, we schedule a change to the value of the output signal *O1* with the event scheduler.

- The implementation will also have to model the wire connecting the AND gate to the NOT gate *via* the snippet

```
newtick T1 in
  loop
    await tick;
    emit schedule-transport-2ns-T2(?T1)
  end
end
```

The code above ensures that signals *T2* is a delayed version of signal *T1*.

- We will also require three signal drivers for *T1*, *T2* and *O1*, specified like the code snippet below

```
newtick trigger-T1 in
   loop
      await tick;
      emit T1(?trigger-T1)
   end
end
```

The drivers ensure that upon the arrival of the *trigger event* from the event scheduler, the corresponding signal is synchronously emitted.

- The NAND gate can now be modeled by the parallel composition of the above code snippets together with an external event scheduler.
- The event scheduler is responsible for implementing both *transport* and *inertial* delays for the system. We have however left the implementation of the event scheduler itself unspecified. Furthermore, we have assumed that the event scheduler is fast enough to be able to process all the request made to it by the ESTEREL module.

Example 4.2 Micro-pipelines - Let us first review the working of the circuit shown in Figure 6. In Figure 6, the circle labeled ME represents a *Mueller element* whose behavior is to set the state of its output line to that of its input lines whenever all the input lines have the same state and maintain its previous state otherwise. Points marked *a* and *g* are inputs while those marked *b* and *f* are outputs.

To start with let us assume that both inputs (*a*, *g*) are held low and points *d*, *c* and *e* (and hence points *b* and *f*) are also low. The outputs at all the *inverters* shown will be high.

Now consider that input point *a* makes a transition from low to high. The *Mueller* element labeled 1 will then drive the point *c* high too. This in turn will cause the *Mueller* element

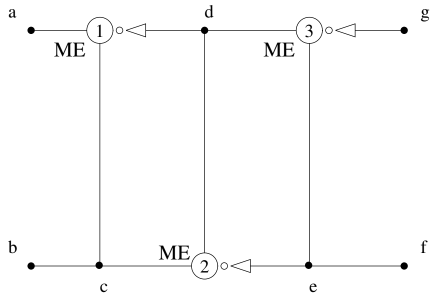

Figure 6. Asynchronous Pipeline

labeled 2 to drive the point *d* high. This causes the output of the inverter connected to the *Mueller* element labeled 1 to go low. Point *d* going high in turn causes the *Mueller* element labeled 3 to drive the point *e* (and therefore point *f*) high. This in turn sets the output of the inverter connected to the *Mueller* element labeled 2 to be driven low.

Now consider a second transition on the input point *a*, this time from high to low. As in the case above, point *d* will eventually be driven low, because the *Mueller* elements labeled 1 and 2 will allow the new state to pass through on the output line. However *Mueller* element labeled 3 will not drive the point *e* low because, the output of the inverter driving one of its inputs is still high. Also note that the output of the inverter connected to the *Mueller* element labeled 1 is now high (because point *d* is low) while the output of the inverter driving the *Mueller* element labeled 2 is still low (because point *e* hasn't changed state). The net result is that the second transition remains blocked at the *Mueller* element labeled 3.

Reasoning in an analogous manner, it is easy to see that a third transition on input point *a* will remain blocked on the *Mueller* element labeled 2 and so on.

Now consider the effects of driving point *g* high. This drives the output of the inverter connected to the *Mueller* element labeled 3 low. This in turn will allow the low value at point *d* to be gated to point *e* (and therefore point *f*). Thus the second transition on point *a* will now be visible at point *f*. Point *e* being driven down will cause the output of the inverter connected to the *Mueller* element labeled 2 to go high. This in turn will allow the high value at point *c* (due to the third transition on point *a*) to propagate until point *d* where it will get blocked by the *Mueller* element labeled 3.

Thus, we see that the circuit functions as an asynchronous FIFO wherein transitions on point *a* are eventually visible at point *f* so long as point *g* is toggled to acknowledge receipt of each successive value available at point *f*. Note that values propagate along the FIFO only when the previous transition has been acknowledged and that the circuit is not explicitly clocked.

The micro-pipeline of Figure 6 can be specified in *Multiclock* ESTEREL by building each component separately and combining them to construct the entire circuit.

A *Mueller* element is a circuit that either maintains its previous state or takes on the value of its inputs *if* all its inputs have the same value. The code below models such a circuit. Note that the system will make transitions only when either input changes and remembers the previous value seen on an input.

```
module MUELLER :
inputs I1, I2
output O1
  newtick I1 or I2 in
    var LAST_VAL=0, I1_VAL=0 I2_VAL=0 in
      loop
        await tick
        present I1 then I1_VAL = ?I1 end
        present I2 then I2_VAL = ?I2 end
        if I1_VAL == I2_VAL then
          LAST_VAL = I2_VAL
        end
        emit O1(LAST_VAL)
      end
    end
  end
end
```

```
module INVERTER :
inputs I1
output O1
  newtick I1 in
    loop
      await tick;
      emit O1(not ?I1)
    end
  end
end
```

The code represents a simple NOT gate or *inverter*. Note that the code makes transitions only when the input changes state. Also note that the circuit has no memory.

```
module COMBINATION :
inputs I1, I2
output O1
  signal T1 in
    run INVERTER [T1/O1] ||
    run MUELLER [T1/I2]
  end
end
```

The code above shows how *Multiclock* ESTEREL modules can be composed to give other modules. Note that the output of the *inverter* is bound to one input of the *Mueller element* through the local signal *T1*. This usage of local signals is very typical of *Multiclock* ESTEREL programming.

```
module MICRO_PIPELINE :
inputs SIG_IN, ACK_IN
inputoutput SIG_OUT, ACK_OUT
  signal T1 in
    run COMBINATION [SIG_IN/I1 T1/I2 ACK_OUT/O1] ||
    run COMBINATION [ACK_OUT/I1 SIG_OUT/I2 T1/O1] ||
    run COMBINATION [T1/I1 ACK_IN/I2 SIG_OUT/O1]
  end
end
```

The above code shows how we can specify complex behaviors in a very modular fashion using *Multiclock* ESTEREL modules bound together using local signals.

5 Conclusion

In the previous sections, we have presented how VHDL constructs may be implemented in *Multiclock* ESTEREL. This makes for interesting possibilities wherein *Multiclock* ESTEREL and VHDL can be used in attempts at co-design at one end, to translating segments of VHDL code into *Multiclock* ESTEREL for verification at the other. It also opens up the possibility of specifying circuits in their entirety in *Multiclock* ESTEREL. The ease with which we could model transport and inertial delays using an external event scheduler shows the flexibility that is built into the design of *Multiclock* ESTEREL. *Multiclock* ESTEREL truly bridges the gap between purely synchronous systems and purely asynchronous ones. In a prototype implementation, we have studied a monochronous implementation of *Multiclock* ESTEREL and studied the basic translations in relation to the classical ESTEREL programming environment. One of the distinct contributions of the paper has been the modeling of a subset of timing, delay and reactive features which form the basis of simulation cycle or the informal semantics of VHDL in terms of the formal rigorously defined notions of synchronous languages. As compared to the other efforts described in section 2, one of the distinct advantages of our approach has been that it provides a unified framework that can effectively use both the synchronous as well as the VHDL paradigms; the unification achieved exploits the fact that *Multiclock* ESTEREL is also imperative like VHDL and supports multiple clocks and asynchrony. In other words, the formal verification capabilities of *Multiclock* ESTEREL due to the underlying verification programming environments (including optimizers) of ESTEREL and the powerful simulation and synthesis tools can be used in a single framework.

References

[1] H. Touati and G. Berry. Optimized controller synthesis using esterel. In *Proc. International Workshop on Logic Synthesis IWLS'93, Lake Tahoe*, 1993.

[2] S. A. Edwards. *The Specification and Execution of Heterogenous Synchronous Reactive systems.* PhD thesis, University of California, Berkeley, 97.

[3] Mohammed Belhadj. Vhdl & signal: A cooperative approach. Manuscript, 1992.

[4] Max Fuchs and Michael Mendler. A Functional Semantics for Delta-Delay VHDL Based on Focus. In Carlos Delgado Kloos and Peter T. Breuer, editors, *Formal Semantics for VHDL*, pages 9–41. Kluwer Academic Publishers, 95.

[5] Peter T. Breuer, Luis Sanchez Fernandez and Carlos Delgado Kloos. A Functional Semantics for Unit-Delay VHDL. In Carlos Delgado Kloos and Peter T. Breuer, editors, *Formal Semantics for VHDL*, pages 43–70. Kluwer Academic Publishers, 95.

[6] Krzysztof Wolinski and Mohammed Belhadj. High Level Synthesis of Globally Asynchronous Locally Synchronous Circuits. Manuscript, 1993.

[7] Basant Rajan. *Programming Languages: Specification & Design of Multiple-Clocked Systems.* PhD thesis, Tata Institute of Fundamental Research, 98.

[8] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design semantics, Implementation. *SCP*, 19(2):87–152, Nov 92.

[9] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. *ACM POPL*, pages 85–99, Jan 93.

[10] G.Berry. Preemption in Concurrent Systems. volume 761 of *LNCS*, pages 72–93. Springer-Verlag, 93.

[11] R.K. Shyamasundar. Specification of Hybrid Systems in CRP. In *Proc. of AMAST 93*, Workshops in Computing Series, pages 227–238. Springer-Verlag, 93.

[12] G.Berry. Preemption in Concurrent Systems. In *Proc. FSTTCS*, LNCS, pages 72–93. Springer-Verlag, 93.

[13] D.Harel and A.Pnueli. On the development of reactive systems: logics and models of concurrent systems. In *Proc. NATO ASI Series F 13*, pages 477–498. Springer, New York, 85.

[14] Basant Rajan and R.K. Shyamasundar. An Implementation of CRP. In *IASTED*, Singapore, 97. (Also TIFR/TCS-95/7 Bombay, India).

[15] Rajeev Alur. Reactive Modules. In *Proc.* 11^{th} *Annual IEEE Symposium on Logic in Computer Science*. IEEE, 96.