Multiclock ESTEREL: A Reactive Framework for Asynchronous Design*

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

1 Introduction

Synchronous circuit design is supported by a wide range of design automation tools both at academic and industrial scales. Correctness of synchronous circuits is very much dependent on the accuracy of the distribution of clocks. Hence, analysis and characterization of the clocks has been one of the priority areas of research in synchronous circuit design. With increases in clock frequency the design and correctness issues become very complex due to the underlying clock skews, power consumption and meta-stability. On the other hand, asynchronous circuit design offers an alternative that does not have these problems due to the absence of a global clock. Further, it has the advantages of composability and robustness. However, it suffers from the lack of mature behavioral synthesis tools.

The two most important issues of hardware design are: data path and control path specification and synthesis. While the data-path can be handled effectively through Boolean equations and the associated tools, the specification of large and complex controls is quite involved. High level languages such as VHDL or VERILOG have been used widely for this purpose. Though these languages have become the *de facto* standards and have mature industrial scale tools, these languages suffer from one or more of the following drawbacks: (i) lack of formal semantics, (ii) lack of reactive features such as preemption, suspension, exception handling, (iii) inability to support modular design of hardware with multiple clocks and asynchronous circuits.

Synchronous circuit modelling has made considerable gains from the use of synchronous languages such as ESTEREL, LUSTRE, SIGNAL as well as Statecharts. One of the most significant gains of using such models is that it has become possible to move from the realm of simulations to that of proofs about the correct behaviour of hardware circuits before they are actually implemented. In fact, the compiling techniques of ESTEREL seem very effective for deriving efficient programs.

While ESTEREL and other reactive languages are extremely useful in designing verifiable circuits, they lack the capability to specify hardware with multiple clocks and asynchronous circuits in a modular way.

In this paper, we discuss a new paradigm called *Multiclock* ESTEREL, based on the paradigm of the synchronous reactive language, ESTEREL, used for reactive systems and synchronous circuit design. We show that the *Multiclock* ESTEREL paradigm provides a general framework for the design of systems with multiple local clocks and the earlier paradigm of *CRP* (*Communicating Reactive Processes*) [1] can be obtained as an instance of the newly proposed paradigm. Furthermore, it preserves the advantages of the classical ESTEREL paradigm and thus benefits from the advantages of verifiability of specifications/models.

Multiclock ESTEREL provides a formal basis for designing asynchronous circuits and provides a succint unification of synchrony and asynchrony. The main contributions of are :

1. provides a multi-clock generalization of ESTEREL, eliminateing the shortcomings of the classical ESTEREL such as weaknesses of module definitions, parameterizing clocks, context-dependent suspension operators etc.

2. provides a unification of synchrony and asynchrony, thus leading to modular specification of asynchronous systems with multiple clocks.

3. provides a clean model for VHDL/VERILOG features.

2 Need for a Multiclocked Paradigm

First, we shall highlight the limitations of globally clocked systems and the advantages of multiple clock systems. It may be pointed out that ESTEREL is a monoclock

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

system. We shall base our discussion of these aspects on an example of a simple CPU-RAM system.

Consider a simplified example of modelling a system consisting of a CPU and some RAM. Let us assume that the CPU runs at a faster clock than the RAM and that they synchronize only during data transfers between them. Modelling such a system using existing synchronous languages needs to analyze the following issues:

1. Global *vs.* Local Clocks: With increase in the complexity of systems, it becomes necessary and convenient to *distribute* functionality of a program. Physical distribution naturally leads to multiple clocks. For instance, in the above example, it would be nice to develop the model for the CPU and the RAM by specifying the clock of each system independently. However, if we use a global clock, we need to explicitly filter clock ticks to yield the desired clock; such a scenario will require complete re-design if we change the clock.

   Synchronous languages implicitly work with a notion of a global clock and hence, it becomes an impediment especially when independent module development is attempted. The paradigm of CRP [1] provides a method of dealing with multiple clocks by distinguishing explicitly between synchronous i and network parallelism.

2. Preserving Modularity - In the context of the above CPU - RAM example above, independent code development for the CPU (or RAM) module cannot be done without explicit knowledge of the clock for the RAM (or CPU) module. This information would be required to build the explicit clock filters for these modules The applicability of a language to characterize and implement real systems will be enhanced if modularity of the language is not affected by the introduction of independent clocks.

   Classical ESTEREL tacitly assumed that there existed a global clock in the system of interest. This limitation comes in the way of maintaining modularity when dealing with systems with multiple clocks.

3. Synchrony *vs.* Asynchrony - Extending our CPU - RAM example above to include *interrupts*, it becomes obvious that one needs to be able to handle asynchronous interrupts in the CPU module since external interrupts are not constrained to be synchronous with the CPU clock. Another area where synchronization comes to the fore in this example is when the CPU and RAM modules exchange data. Even though the CPU and RAM run on independent clocks, they do synchronize between themselves. The synchronous paradigm imposed severe constraints on the kind of systems that could be effective characterized using it. In particular, it required that input events occurred synchronously with the internal system clock. Unfortunately, very many systems have asynchronous interactions with their environment. The need to deal with asynchronous events in the environment require that reactive languages be adapted to interface to these asynchronous events. Furthermore, it would be nice to have a unified way of abstracting synchrony and asynchrony from a theoretical as well as practical perspective.

4. Verifiability - Naturally, we would like to be able to prove that, say, interactions between the CPU and RAM modules satisfy their design specifications. Synchronous languages supported the development of *verifiable* code. By judiciously relaxing the constraints imposed by the synchronous paradigm, it should still be possible to apply proof techniques developed for the synchronous language.

5. Retention of existing features of reactive languages - In our example, we would like to be able to model the CPU and RAM modules working in parallel in situations such as: prefetching instructions, suspension of normal operations when interrupts occur, or preemption when error conditions occur.

   Synchronous languages provide features like parallelism, preemption and suspension, making them well suited to address problems in certain application areas. These features should continue to be supported in any alternate proposal to augment functionality.

3 *Multiclock* ESTEREL, an Introduction

Multiclock ESTERELis 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 support the explicit specification of clocks in the sense that while in ESTEREL the reactions to external events were controlled just 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. Rules for defining or accessing variables or signals or sensors, data structures and host languages follow essentially on the same lines as that of ESTEREL with the possibility of explicit access controls.

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 describe latches and the structure of signal expressions in sections 3.1 & 3.2 respectively.

3.1 Latched Signals

ESTEREL or synchronous languages in general do not require latching of signal. The underlying model followed assumed that the system was fast enough to react to every input event. The capability of *Multiclock* ESTEREL to explicitly bind *clocks* to specific signals (including boolean function on them) introduces certain problems. For instance, how does one treat events that occur *out of sync* with the current *tick* signal?

The form of latch used in *Multiclock* ESTEREL(refer to [2] for other possible notions) is shown in Figure 1.

- the waveform labeled *tick* is assumed to be the current clock. It is not necessary that the clock be periodic as shown. In fact, in the case of event driven systems this is likely to be the norm rather than the exception.

- 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. This implies that, for a given set of clocks and signals, it is possible to lose all record of certain events when the clocks used are too slow for the signals being monitored. This condition is referred to as *under sampling*. As a direct consequence of under sampling, we cannot hope to distinguish between a single occurrence of an event and multiple occurrences of an event between successive clock ticks. *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.

1. The first kind ranges over signals and are referred to as *clock* expressions. Clock expressions are booleans functions over signals and cannot contain references to variables or sensors. They are used to specify bindings for the local clock or *tick*.

2. 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 *Multiclock* ESTEREL Statements

For simplicity, we discuss only pure *Multiclock* ESTEREL i.e., without value passing and variable assignments etc. We shall let p, q range over programs and s over signals in the rules that follow.

The kernel statements in *Multiclock* ESTEREL along with its intuitive meaning are given in Table 1. It can be seen that the constructs look the same as those of ESTEREL except the new statement, the *newtick* statement, that binds clocks to statements. For lack of space, we shall not go into detail in each of the constructs. The meaning of all the statements follows on the lines of those of ESTEREL except that whenever one refers to the presence/absence of a signal, it refers to the *latched version of the signal*.

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*
`var` *s* **`in`** *stat* **`end`**	*scope of* s *is limited to* stat
`await` *s*	**`abort halt when`** *s*

Table 1. Syntax of *Multiclock* ESTEREL

The meaning of the `newtick` statement is given below. This construct is what sets *Multiclock* ESTEREL apart from ESTEREL. This construct starts 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.

3.4 Program Development in *Multiclock* ESTEREL

In this section, we shall illustrate style of development of programs in *Multiclock* ESTEREL using a nice asynchronous system referred to as *micropipelines*, first introduced in [3] and discussed in [4].

Example 3.1 Micro-pipelines - Let us first review the working of the circuit shown in Figure 2. In Figure 2, 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 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 pointc (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 on 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 2 can be specified in *Multiclock* ESTEREL by building each component separately and combining them to construct the entire circuit follows from the following discussion.

```
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
```

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 above code 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. Note that the construct **`await`** *tick* is a derived construct whose effect is to wait until a *tick* occurs[2].

```
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. The notation **?***I1* is used to query the value of the latched version of signal *I1* at the current instance[2].

```
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.

Multiclock ESTEREL Statement	$\rightarrow$	Clocked Form
(1) $stat$	$\rightarrow$	$stat\_$
(2) $(stat\_)_c$	$\rightarrow$	$_c(stat_c)$
(3) $(stat1\_; stat2\_)_c$	$\rightarrow$	$_c(stat1_c; stat2_c)$
(4) $(stat1\_ \Vert stat2\_)_c$	$\rightarrow$	$_c(stat1_c \Vert stat2_c)$
(5) $(\textbf{loop}\ stat\_\ \textbf{end})_c$	$\rightarrow$	$_c(\textbf{loop}\ stat_c\ \textbf{end})$
(6) $(\textbf{abort}\ stat\_\ \textbf{when}\ S)_c$	$\rightarrow$	$_c(\textbf{abort}\ stat_c\ \textbf{when}\ S)$
(7) $(\textbf{signal}\ S\ \textbf{in}\ stat\_\ \textbf{end})_c$	$\rightarrow$	$_c(\textbf{signal}\ S\ \textbf{in}\ stat_c\ \textbf{end})$
(8) $(\textbf{present}\ S\ \textbf{then}\ stat1\_\ \textbf{else}\ stat2\_\ \textbf{end})_c$	$\rightarrow$	$_c(\textbf{present}\ S\ \textbf{then}\ stat1_c\ \textbf{else}\ stat2_c\textbf{end})$
(9) $(\textbf{newtick}\ S\ \textbf{in}\ stat\_\ \textbf{end})_c$	$\rightarrow$	$_c(\textbf{newtick}\ S\ \textbf{in}\ stat_S\ \textbf{end})$
(10) $_c(\textbf{nothing}_c)$	$\rightarrow$	$\textbf{nothing}_c$
(11) $_c(\textbf{nothing}_s)$	$\xrightarrow[s \in E]{}$	$\textbf{nothing}_c$

Table 2. Clock Binding and Propagation

```
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.

4 *Multiclock* ESTEREL: Semantics

In this section, we shall sketch the behavioral semantics of *Multiclock* ESTEREL. Being multi-clocked, execution of statements in *Multiclock* ESTEREL is initiated at every occurrence of the current local clock and therefore clock distribution forms a very important part of the execution model. We therefore first describe how statements are bound to clocks before we present a more formal, semantic treatment of *Multiclock* ESTEREL's programming constructs.

4.1 Propagation and Binding of clocks

Statements in *Multiclock* ESTEREL can be bound to different clocks but necessarily need to be bound to a clock before it can be scheduled for execution. *Multiclock* ESTEREL being a synchronous language, even termination of statements is tightly coupled to the clock in effect. We will use the notation $stat_c$ to denote the statement *stat* running at clock *c*, $stat\_$ to denote that no clock has yet been specified for the statement *stat* and $_c(stat)$ to denote that statement *stat* is scheduled to terminate on clock *c*. The clock distribution for some of the statements is given in Table 2.

Rule (1) effectively means that initially statements are not bound to any clock and rule (2) ensures that statements that have unbound clocks get bound to the current clock before execution. Note that they are constrained to terminate on the same clock. The other show how the clocks are propagated. For instance, rule (3) shows how clock bindings distribute across the sequential operator. Note that for the rule to be applicable both components of the compound statement must be free from any clock bindings. Each component is then bound to the clock *c*. Also note that the whole construct is constrained to terminate on clock *c*. Rule (6) implies that statements inside the scope of an **`abort`** construct bound to a clock *c* is bound to the same clock *c* and is also constrained to terminate on clock *c*. The **`newtick`** construct can explicitly bind statements to specific clocks. Rule (9) causes statement *stat* to be bound to the clock *S* though the **`newtick`** construct itself may be bound to clock *c*. Also note that the entire construct is still constrained to terminate on clock *c*. Rule (10) differs from the other rules in that it deals with synchronization of statement termination. This rule implies that a **`nothing`** construct bound to clock *c* and constrained to terminate also on clock *c* is equivalent to a **`nothing`** construct bound to clock *c*. This rule ensures that an extra tick of clock *c* is not consumed upon termination of the construct. Rule (11) can be applied *only* when clock *s* is present. This rule implies that a **`nothing`** construct bound to clock *s* and constrained to terminate on clock *c* is equivalent to a **`nothing`** construct bound to clock *c* (after clock *s* occurs). This rule ensures that the construct terminates on clock *c*.

From the above rules we see that

- the execution clock for a given statement or block of statements is inherited from the enclosing block, unless explicitly reassigned by a **`newtick`** statement,
- statements are constrained to terminate on the clock they started on, and
- the execution clock distributes over the sequence and parallel operators.

Now that the rules regarding clock propagation have been dealt with, we will discuss the semantics of the individual statements in *Multiclock* ESTEREL.

4.2 Behavioral Rules for Kernel Statements

We shall use the notations from ESTEREL and *CRP*[1] in the rules described below.

Given a module *M* and an input event *I*, the behavioral semantics determines a transition $M \underset{I}{\overset{O}{\rightarrow}} M'$ where *O* is the generated output event and *M'* is another module suited to perform further reactions. The derivative *M'* has the same interface as *M* and differs only by its body. The reaction to a sequence of input events is computed by chaining such elementary transitions. The relation between the two transition systems is given by

$$M \underset{I}{\overset{O}{\rightarrow}} M' \text{ iff } < stat > \xrightarrow[I \cup O\{tick\}]{O,L,b} < stat' >$$

where $stat$ and $stat'$ are the bodies of M and M' respectively, `tick` denotes the current clock in the absence of which all transitions are disabled and the transition relations on statements is defined below. We assume the harmless restriction that $stat$ cannot internally emit input signals. The transition relation on statements is defined using an auxiliary inductive relation

$$< stat > \xrightarrow[E]{E',L,b} < stat' >$$

where the old and the new signal environments are defined by:

- *E* the current event to which *stat* reacts and
- *E'* the event emitted by *stat* in response to the event *E*.
- L is the set of labels associated to **`trap`** statements currently active in $stat$. To avoid cluttering, this term is dropped in the rules presented in Tables 3 and 4 For a complete set of rules, please refer [2].
- The integer *termination level b* determines how control is handled. In each reaction, a statement can behave in either of two ways: it can terminate and release the control or it can retain the control and wait for further events. We set $b = 0$ for proper termination, and $b = 1$ for retention of control.
- Since signals are broadcast synchronously, *stat* receives the signals it emits and *E'* will always be contained in *E*.

The transition rules for the main kernel statements are given in Table 3-4. In the rules shown, E_c denotes that c is the clock on which the transition was enabled. Thus, rules 1-6 follow trivially follow on the same lines as that of ESTEREL and keeping in view the clock propagation rules. For instance, the parallel construct (rule 6) terminates only when both the branches terminate on the clock c on which the parallel operator is bound; no extra clock `tick` is lost follows from the clock propagation rules given earlier for `nothing`. Rules 7-8 describe the behavior for the present-statement; the only difference to be noted is that signal S referred is the latched-version of S of clock c on which the present-statement is bound. Rules 9-11 describe the rules for the immediate-strong preemption construct; rule 9 corresponds to preemption due to the presence of the latched version of S (relative to c) and discarding the output; rule 10 corresponds to the case of the absence of latched S in the current clock and rule 11 corresponds to termination of the body before the latched version of S. Note that the termination happens on clock c. The strong-preemption on the next occurrence of signal is rewritten in terms of the immediate-strong preemption construct. Suspend-constructs are one of the most powerful constructs (cf. [5]) of synchronous languages wherein the underlying clocks are modified. Rules 13 corresponds to the suspend-immediate construct; here, we are re-assigning a new clock derived as a function of the existing clock d_u and the signal s to the statement block *stat* where u and v denote suspension contexts. The suspension context u and the signal s are used to derive a new suspension context v which controls the execution of the statement *stat*. Informally, a function of the suspension context is used to gate the clock to the statement block within the scope of the **`suspend`** statement. The details of such an implementation are omitted here. The rule for this statement cannot be modeled in terms of the rules for suspend in ESTEREL. Rule 14 denotes that the suspend-construct behaves like the statement within its scope and then transforms into the **`immediate`** variant of the **`suspend`**. Note that for this transition to be enabled the statement within the scope of the suspend statement ($stat$) should not terminate at the first instance. Rule 15 captures how a **`suspend`** statement is treated when the statement block it encloses terminates in the first instance. Note that the whole construct transforms into the **`nothing`** construct.

4.3 Expressiveness and Implementation

Having described the semantics of the language in terms of the inductive rules, its consistency and soundness are established here. The theorems are stated in an informal way to avoid further notations.

Theorem 1: With the restriction that output signals are not used in clock expressions, the clock propagation rules provide a unique non-cyclic binding of clocks for the statements and the process of clock propagation terminates.

Theorem 2: If there is no explicit clock i.e., no `newtick` statement used in the program the semantics coincides with the classical semantics of ESTEREL.

Theorem 3: If the given *Multiclock* ESTEREL program is non-causal (in the sense of the notion of ESTEREL) and the clocks are not cyclic (verifiable by the simple restrictions mentioned earlier), then the program is executable ignoring *metastability* issues due to the density of clocks.

From this theorem, the following corollary can be established; this result also provides a comparison to the other synchronous language SIGNAL that permits oversampling and hence, there are several SIGNAL specifications that cannot be compiled.

Corollary 1: Oversampling that is possible in SIGNAL language is not possible in *Multiclock* ESTEREL .

Communicating reactive processes (CRP) is essentially a network of ESTEREL nodes connected by synchronous communication channels. The following theorem shows the implementability of CRP in *Multiclock* ESTEREL .

Theorem 5: Using the channels of CRP as input clocks, CRP can be realized in *Multiclock* ESTEREL .

Another interesting question that can be asked is:

> *Is it meaningful to have a translation of* Multiclock ESTEREL ESTEREL *?*

The following theorem answers the following question:

Theorem 6: *Multiclock* ESTEREL program can be translated into ESTEREL at the cost of modularity.

From theorem 6, it can be seen that due to the process of synchronization of the asynchronous behaviour, one loses the modularity. Specifically, *shallow suspend* can not be translated into ESTEREL without loss of modularity. Further, the suspend construct as it exists in classical ESTEREL is itself not truly modular.

4.4 Modelling a VHDL subset

Here, we sketch how *Multiclock* ESTEREL can be used to provide a rigorous semantics for VHDL subsets[6].

VHDL uses 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 point 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.

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 realtime 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. We model these aspects using a eventscheduler shown in Figure 3.

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 for the *Multiclock* ESTEREL program. The entire configuration is depicted 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. Using such a scheme, we can model the timing, reactive and delay features of VHDL. For details the reader is referred to the thesis of the first author.

5 Discussions

In the previous sections, we presented the paradigm of *Multiclock* ESTEREL. *Multiclock* ESTEREL truly bridges the gap between purely synchronous systems and pure asynchronous ones. It really unifies synchrony and asynchrony as opposed to CRP wherein one has to keep distinction between ESTEREL parallelism and network parallelism. 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. Further, several varieties of suspend and clock binding operators are possible that can be used in a modular way that have parallels in hardware design. In fact, *Multiclock* ESTEREL easily models space and time scaling as envisaged in *reactive modules* with the additional benefits of preemptive features and integration of synchrony and asynchrony for hardware specification.

Furthermore, we have also used *Multiclock* ESTEREL to model interesting (from the semantic point of view) subsets of VHDL. This study has brought out 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 transport and inertial delays can be modelled using an external event scheduler shows the flexibility that is built into the design of *Multiclock* ESTEREL. One of the distinct contributions of *Multiclock* ESTEREL paradigm has been the modelling of timing, delay and reactive features that form the basis of simulation cycle or the informal semantics of VHDL in terms of the formal rigorously defined notions of synchronous languages.

From the perspective of future work, inclusion of support for structured data types would help in making the language more programmer friendly. Work in the areas of dealing with analog systems could also be explored.

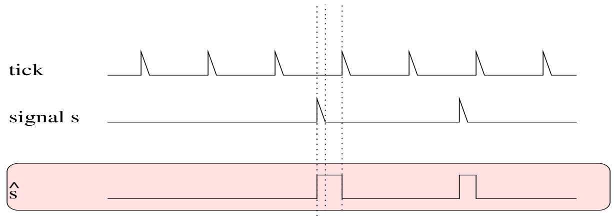

Figure 1. Clock, signal and latches

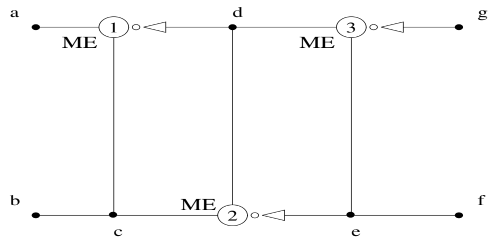

Figure 2. Asynchronous Pipeline

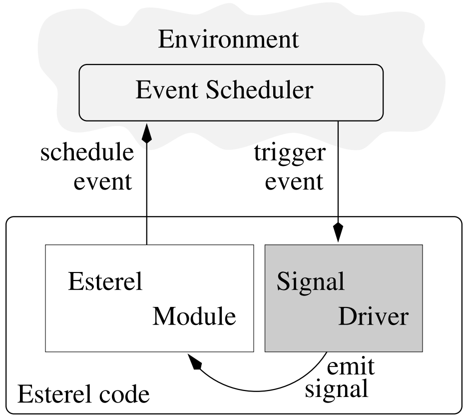

Figure 3. Event Scheduling

References

[1] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. *20th ACM POPL*, pages 85–99, Jan 93.

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

[3] I.E. Sutherland. Micropipelines. *Comm. of the ACM*, 32(6):720–738, June 1989.

[4] Scott Hauck. Asynchronous design methodologies: An overview. *Proc. of the IEEE*, 83(1):69–93, Jan 1995.

[5] Sophie Pinchinat and Eric Rutten and R.K. Shyamasundar. Taxonomy and Expressiveness of Preemption: A syntactic approach. In *Proceedings ASIAN 98*, 98.

[6] B. Rajan and R.K. Shyamasundar. *Multiclock* ESTEREL: A reactive framework for asynchronous design. In 13th *Intl. Conf. on VLSI Design*, pages 76–83, Calcutta, India, Jan 2000.

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

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

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

[10] D.Harel and A.Pnueli. On the development of reactive systems: logics and models of concurrent systems. In *Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, NATO ASI Series F 13, pages 477–498. Springer, New York, 85.

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

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

$nothing_c \xrightarrow[E_c \cup \mathcal{L}]{\phi,0} nothing_c$	...1
$halt_c \xrightarrow[E_c \cup \mathcal{L}]{\phi,1} halt_c$	...2
$emit_c S \xrightarrow[E_c \cup \mathcal{L}]{S,0} nothing_c$	...3
$\dfrac{stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',0} {}_c(stat1')\ ,\ stat2_c \xrightarrow[E_c \cup \mathcal{L}]{E'',b} {}_c(stat2'_d)}{stat1_c; stat2_c \xrightarrow[E_c \cup \mathcal{L}]{E' \cup E'',b} {}_c(stat2'_d)}$	...4
$\dfrac{stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d)\ b \neq 0}{stat1_c; stat2_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d); stat2_c}$	...5
$\dfrac{stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',b'} {}_c(stat1'_d)\ ,\ stat2_c \xrightarrow[E_c \cup \mathcal{L}]{E'',b''} {}_c(stat2'_e)}{stat1_c \parallel stat2_c \xrightarrow[E_c \cup \mathcal{L}]{E' \cup E'',b' \wedge k''} {}_c(stat1'_d) \parallel_c (stat2'_e)}$	...6
$\dfrac{S \in E_c,\ stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d)}{present\ S\ then\ stat1\ else\ stat2\ end_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d)}$	...7
$\dfrac{S \notin E_c,\ stat2_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat2'_d)}{present\ S\ then\ stat1\ else\ stat2\ end_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat2'_d)}$	...8

Table 3. Behavioral Rules I

$\dfrac{S \in E_c\ ,\ stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d)}{abort\ stat1\ when_c\ immediate\ S \xrightarrow[E_c \cup \mathcal{L}]{\phi,0} nothing_c}$	...9
$\dfrac{S \notin E_c\ ,\ stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d)}{abort\ stat1\ when_c\ immediate\ S \xrightarrow[E_c \cup \mathcal{L}]{E',b} abort\ {}_c(stat1'_d)\ when_c\ immediate\ S}$	...10
$\dfrac{S \notin E_c\ ,\ stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',0} {}_c(stat1'_d)}{abort\ stat1\ when_c\ immediate\ S \xrightarrow[E_c \cup \mathcal{L}]{E',0} nothing_c}$	...11
$\dfrac{stat1_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat1'_d)}{abort\ stat1\ when_c\ S \xrightarrow[E_c \cup \mathcal{L}]{E',b} abort\ {}_c(stat1'_d)\ when_c\ immediate\ S}$	...12
$\mathtt{suspend}\ {}_c(stat_{d_u})\ \mathtt{when}_{c_u}\ \mathtt{immediate}\ s \rightarrow_c (stat_{d_v})$	...13
$\dfrac{stat_c \xrightarrow[E_c \cup \mathcal{L}]{E',b} {}_c(stat'_d)}{\mathtt{suspend}\ stat\ \mathtt{when}_c\ s \xrightarrow[E_c \cup \mathcal{L}]{E',b} \mathtt{suspend}\ {}_c(stat'_d)\ \mathtt{when}_c\ \mathtt{immediate}\ s}$	...14
$\dfrac{stat_c \xrightarrow[E_c \cup \mathcal{L}]{E',0} {}_c(stat'_d)}{\mathtt{suspend}\ stat\ \mathtt{when}_c\ s \xrightarrow[E_c \cup \mathcal{L}]{E',0} \mathtt{nothing}_c}$	...15
$newtick\ S\ in\ stat\ end_{c_u} \xrightarrow[E_c \cup \mathcal{L}]{\phi,b} {}_c(stat_{s_\phi})$	...16

Table 4. Behavioral Rules II