MODELING DISTRIBUTED EMBEDDED SYSTEMS IN *Multiclock* ESTEREL*

Basant Rajan
VERITAS Software India Pvt. Ltd.
S.No. 210 A/1, Range Hills,
Pune 411 020, India
basant@tcs.tifr.res.in

R.K. Shyamasundar
School of Technology and Computer Science
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay 400 005, India
shyam@tcs.tifr.res.in

Abstract In this paper, we show that the paradigm of *Multiclock* ESTEREL can be effectively used for the design of asynchronously communicating distributed systems. First we show that the protocol used in *Multiclock* ESTEREL for the modeling of VHDL can be used for the design of asynchronous interaction of processes, and an analysis can be made relative to speed or periodicity of the underlying processes for a safe implementation without missing any signals. The analysis also shows that one can arrive at a tradeoff between the periodicity and the buffer requirements on the average over a sequence of periods. Then, we illustrate the modeling of *communicating reactive processes* (which is essentially a network of ESTEREL nodes communicating via the rendezvous mechanism) as an instance of *Multiclock* ESTEREL.

Keywords: Embedded Systems, Multiclock Esterel, Synchronous languages.

1. INTRODUCTION

Programming languages (such as ESTEREL, Lustre, Signal, Statecharts etc.) based on the *perfect synchrony paradigm* [3] have proven useful for

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

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35533-7_26

Tommaso Bolognesi and Diego Latella (Eds.), *Formal Methods for Distributed System Development.*
© 2000 IFIP International Federation for Information Processing.
Published by Kluwer Academic Publishers. All rights reserved.

programming reactive systems where determinism is a must and logical concurrency is required for good programming style. One of the main reasons for its success is due to the fact that it permits the programmer to focus on the logic of reactions and makes it possible to use several automata-based verification systems for correctness proofs.

On the other hand, asynchronous languages (like Ada) have proven useful in distributed processing and implementing algorithms on a network of computer systems where nondeterminism is an appropriate abstraction at the logical and physical levels. Presently, each class of languages is unable to handle problems for which the other class is tailored.

However, complex systems do require the abilities of both types of languages. For instance, a robot driver could use a specific reactive program to control each articulation, but the global robot control may remain asynchronous because of limitations of networking capabilities.

In [4, 8, 2], a paradigm referred to as *Communicating Reactive Processes* (CRP) that provides a unification of perfect synchrony and asynchrony has been proposed. CRP explicitly distinguishes the parallelism at the network layer and ESTEREL layer. In a limited way, the concept of hierarchical network of synchronous nodes was introduced in *Hierarchical CRP* [12] without integrating the two notions of parallelism. In [1, 9, 10], the paradigm of *Multiclock* ESTEREL has been proposed as a unified framework for synchrony and asynchrony that overcomes the need to distinguish the various levels of concurrency. Some of the distinguishing features of *Multiclock* ESTEREL that set it apart from its predecessor ESTEREL are:

- *Multiclock* ESTEREL allows the specification of individual clocks for modules - *Multiclock* ESTEREL can now address a wider class of programs without sacrificing modularity.

- *Multiclock* ESTEREL is truly modular, even when using multiple clocks - 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 faced no difficulty dealing with asynchronous external events.

- *Multiclock* ESTEREL implements internal latching of all inputs - This feature serves to integrate seamlessly the domain of asyn-

chronous events with the realm of synchronous execution semantics.

In this paper, we show that the framework of *Multiclock* ESTEREL and the protocols proposed for implementation can be effectively used for:

1 the specification and analysis of asynchronously interacting distributed systems.

2 the specification and analysis of networks of communicating processes with a unified view of synchrony and asynchrony. This in turn shows that classes of distributed systems such as quasi synchronous systems [5] can be effectively specified, implemented and verified in this framework.

The rest of the paper is organized as follows: An informal introduction to *Multiclock* ESTEREL is given in section 2 with an appendix in section 6 giving the informal semantics of *Multiclock* ESTEREL statements. Sections 3 and 4 describe the analysis and specification of asynchronous interaction among processes and the specification of CRP [4] in *Multiclock* ESTEREL respectively. These sections are followed by the concluding remarks.

2. DESCRIPTION OF *Multiclock* ESTEREL

Multiclock ESTEREL is based on the synchronous language ESTEREL which in turn is based on the *perfect synchrony hypothesis. 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, 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.

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*. However, complex data structures like records are not supported.

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 signal will see the signal in its current execution environment.

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. In the sequel, we shall describe latches[1] in section 2.1, expressions in section 2.2 and statements in section 6.

2.1. LATCHED SIGNALS

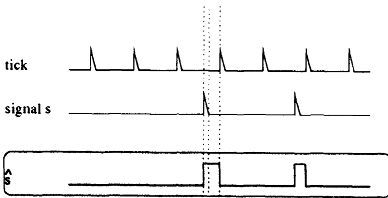


Figure 1 Clock, signal and latches

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

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. *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.

2.2. 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*.

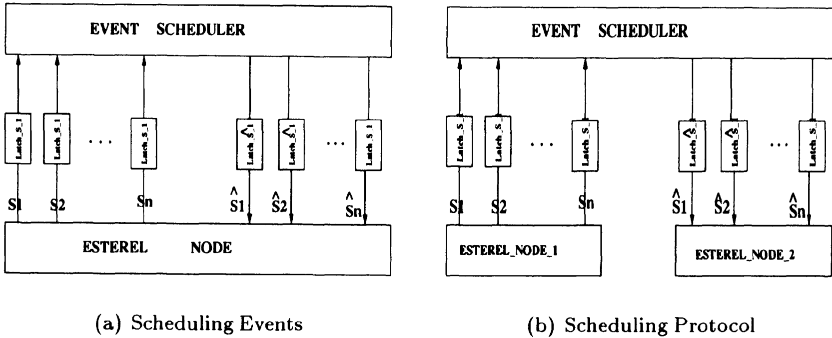
2 The second type of expression ranges over variables as well as *latched* version of signals and are referred to as signal expressions. These expressions are what *Multiclock* ESTEREL code can test for as part of its execution sequence. Note that clock expressions can only be used to bind clock signals. *Latches* of signals are derived entities.

3. MODELING INTERACTION BETWEEN ASYNCHRONOUS PROCESSES

In this section, we show that *Multiclock* ESTEREL and its implementation techniques can be used for modeling asynchronous interaction among processes. We begin with a method for modeling the VHDL feature of *scheduling events at later instants* in *Multiclock* ESTEREL as proposed in [1, 9]. The broad block diagram of the modeling is shown in Fig. 3.

Since the Esterel-module and the Signal-Driver are part of the Esterel code at the same node, we assume without loss of generality that the Esterel program can be denoted by a single node where the synchrony hypothesis holds. Under this assumption, the underlying protocol for scheduling of events is depicted in Fig. 2 where:

$S_1, \cdots, S_n$:	signals to be scheduled later
$latch_S_1, \cdots, latch_S_n$:	signals latched for the event scheduler
$latch_\hat{S}_1, \cdots, latch_\hat{S}_n$:	signals sent from the event-scheduler



(a) Scheduling Events

(b) Scheduling Protocol

Figure 2 Scheduling in Multiclock Esterel

The steps of the implementation as proposed in [1, 9] are given below:

1 On signal emission from Est-node (S_i) working on clock t_{est}, it sets the variable "$latch_S_i$" to $value(S_i)$.

2 Variable "$latch_S_i$" is read by the event-scheduler working on its clock t_{sch}.

3 The event-scheduler sets "$latch_S_i$" to $\perp$ (denoting absence of signal) after reading it so that the read signal will not be mistaken as an unread signal in the next instant.

4 Similar reading/resetting takes place on the dual side.

The requirement of not missing any signal corresponds to the condition given below:

$$\forall_t \forall_i \; \widehat{S_i} = S_i \qquad (1)$$

Let t_{est} and t_{sch} be the clocks of the ESTEREL module and the scheduler-module respectively. Satisfaction of condition (1) implies the following condition:

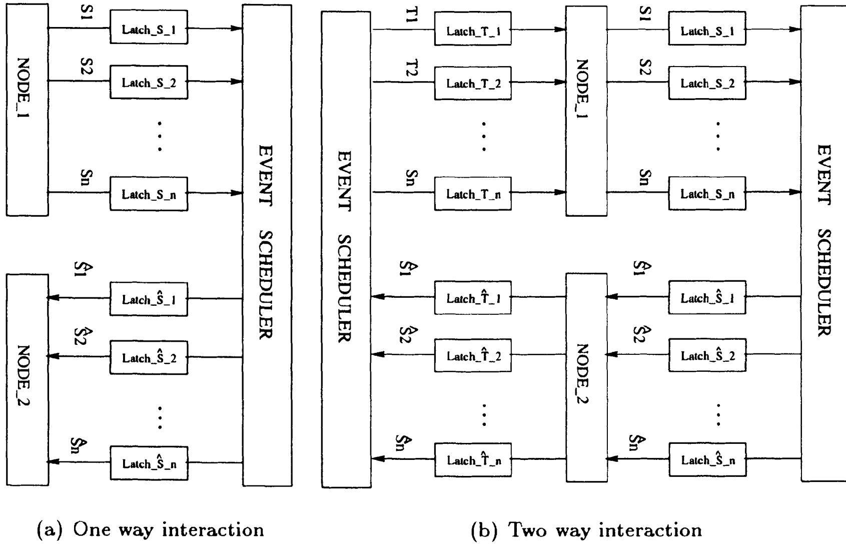
$$t_{sch} \text{ "faster than" } t_{est} \cdots (2)$$

where the relation *"faster than"* is a relation that must ensure the following requirement:

> The clock tick of t_{sch} is frequent enough to consume and deliver it to the Esterel module on t_{est} before the ESTEREL module produces another item to be scheduled.

Notationally, we represent *"faster than"* by the symbol "$\sqsupseteq$". $\cdots$ (2a)

In the above protocol, the interaction was from the Esterel-module to itself via the event-scheduler. Now, let us see the extension of the same



(a) One way interaction (b) Two way interaction

Figure 3 Communication in Multiclock Esterel

where one process interacts with another. The extended protocol with such an extension is depicted in Fig. 2.b where t_{N_1} and t_{N_2} are the clocks of nodes Node_1 and Node_2 respectively. The constraints required for not missing signals are given below:

$$t_{sch} \supseteq t_{N_1} \quad \cdots \quad (3)$$
$$t_{sch} \supseteq t_{N_2} \quad \cdots \quad (4)$$

Obviously, conditions (3)-(4) are not sufficient as we need to ensure relation (2a) between the two processes logically. The corresponding equation follows by considering Node_2 as the consumer and Node_1 as the producer and is given below:

$$t_{N_2} \supseteq t_{N_1} \quad \cdots (5)$$

Now let us consider the situation wherein the processes (Node_1 and Node_2) interact with each other. The scenario is depicted in Fig. 2 wherein for simplicity we have used the same scheduler instead of two schedulers. Now, in addition to conditions (3)-(5), we need the following condition as well (corresponding to the fact that Node_1 is the consumer and Node_2 is the producer):

$$t_{N_1} \supseteq t_{N_2} \quad \cdots (6)$$

From conditions (5) and (6), we can conclude the following condition:

Node_1 and Node_2 are *no faster than* each other ··· (7)

Condition (7) can be interpreted as follows:

Within any two clock ticks of one process the other process can have only one tick. ··· (8)

The above condition is exactly the condition used in [5] for the distributed set of processes called *quasi synchronous processes.* These are periodic processes asynchronously interacting with each other. In other words, it amounts to saying that the period of the processes are the same except that they can be out of phase. The advantage of the approach described above is that we can effectively use the framework of synchronous languages in terms of techniques and tools of verification of synchronous languages for the quasi synchronous systems.

The above conditions can be generalized relative to clocks and buffers. For instance, it is possible to consider the buffer requirements over a sequence of periods of certain processes so that it is possible to achieve the soundness requirement of not missing any signal. In fact, the semantic framework described in [13] can be used.

4. MODELING CRP

A new programming paradigm called Communicating Reactive Processes or *CRP*[4] that unifies the capabilities of asynchronous and synchronous concurrent programming languages is described in [3, 4] with a view to specifying complex reactive systems which usually have both synchronous and asynchronous features. A *CRP* program consists of independent locally reactive ESTEREL nodes that communicate with each other by *rendezvous* communication. Essentially, a *CRP* program consists of a network $M_1 \parallel M_2 \parallel M_n$ of ESTEREL reactive programs or *nodes*, each having its own input/output signals and its own notion of an instant.

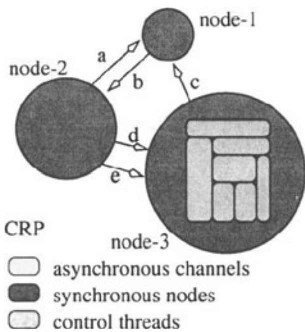


Figure 4 The *CRP* Environment

4.1. EMBEDDING *CRP*

Embedding CRP in *Multiclock* ESTEREL involves (i) controlling the visibility of the channels as required and (ii) the protocol for implementing the rendezvous and the guarded choice.

First, let us consider access control in *Multiclock* ESTEREL . In addition to normal block structured scoping, *Multiclock* ESTEREL permits the use of declarations to *restrict* the visibility of objects. This feature does not modify any of the underlying concepts involved in latching and clocking modules.

This is a generalization of the *CRP* scenario (Figure 4) where ESTEREL nodes had explicit names and channels declared between nodes were visible only to the nodes they connect.

Example 4.1 The ability to arbitrarily restrict the visibility of objects is a prerequisite if we are to correctly implement *CRP* programs within the framework of *Multiclock* ESTEREL because *CRP* programs involve the existence of named channels that are private to the nodes connected by them.

Now, let us turn to protocols required for implementation. The general framework for the implementation would involve implementing each *CRP* node as an independently clocked *Multiclock* ESTEREL module and using signals (which are automatically latched) to communicate amongst them. Private channels are implemented by restricting the scope of the signals constituting the channel.

Considering the configuration given in Figure 4 we could have the following partial implementation.

```
module node3 :
input d, e;
output c visible only to node1
   newtick clock3 in
      statement
   end
end
```

Note that the code for node *node-3* is free to execute on its own private clock *clock3*. This code segment however does not realize *rendezvous* as defined by the *CRP* paradigm. This example only provides the necessary framework to implement *rendezvous*.

We now extend Example 4.1 to implement *rendezvous* as defined in *CRP* on the lines of the implementation described in [2]. Note that we follow the synchronization protocol as given in [2].

Example 4.2 We give below code snippets that implement the *rendezvous* protocol of *CRP*. The code is divided into *sender* and *receiver* components.

```
01 module sender :
02 input rcpt, rabrt, labrt;
03 output data, conf, sabrt, success, fail;

05    present labrt else
06        emit data;
07        await case
08            labrt :  emit sabrt
09            emit fail
10            rcpt :  emit conf
11            emit success
12            rabrt :  emit fail
13        end
14    end
15 end
```

The sender module above works by first emitting the signal *data* and then waiting for either an *rabrt* or *rcpt* signal to arrive from the receiver. On receipt of the *rcpt* signal, the sender confirms the *rendezvous via* the *conf* signal. If on the other hand, the sender is locally preempted using the signal *labrt* it responds by emitting the *sabrt* signal and terminates. Receipt of signal *rabrt* also causes unsuccessful termination of the *rendezvous.* Note that the sender module has not defined an explicit clock and will run at the default one. Further note that preemption at the initial instance precludes the emission of signal *data.* A successful *rendezvous* is characterized by the presence of signal *success* at termination.

```
01 module receiver :
02 input data, sabrt, conf, labrt
03 output rcpt, rabrt, success, failure

05    present labrt else
06       await case
07          labrt :  present data then
08                   emit rabrt
09                   emit failure
10          end
11          data :  emit rcpt
12          await case
13             labrt :  emit failure
14             conf :  emit success
15             sabrt :  emit failure
16          end
17       end
18    end
19 end
```

The receiver module given above works by waiting for a *data* signal to arrive from a sender module and then acknowledging it by emitting a *rcpt* signal. It then awaits the arrival of a *conf* signal which will indicate successful termination. Note that preemption at the initial instant is transparent to the sender whereas preemption simultaneous with the receipt of the *data* signal causes the emission of the *rabrt* signal. Note also that preemption occurring after the *rcpt* signal is emitted will not prevent the sender from treating the *rendezvous* as complete. As in the case of the sender module, no explicit clock has been specified.

Example 4.2 above provides an overview of how *Multiclock* ESTEREL could be used to implement a *CRP* style *rendezvous*. However, this alone does not suffice to implement the generic *CRP* paradigm wherein *guarded choice* could be involved in *selecting* a particular *channel* from amongst many possibilities. The following example outlines how this could be achieved.

Example 4.3 We present an example of how *guarded choice* as in the *CRP* paradigm could be implemented in *Multiclock* ESTEREL. For the sake of brevity we consider only two *rendezvous* in each guard.

```
01 module receiver :
02 input dataA, sabrtA, confA,
      dataB, sabrtB, confB, labrt;
04 output rcptA, rabrtA,
      rcptB, rabrtB, success, failure;

07    present labrt else
08       await case
09          labrt :  present dataA then
10             emit rabrtA
11          end
12          present dataB then
13             emit rabrtB
14          end
15          dataA : emit rcptA
16          await case
17             labrt :  emit failure
18             confA : emit success
19             sabrtA : emit failure
20          end
21          dataB : emit rcptB
22          await case
23             labrt :  emit failure
24             confB : emit success
25             sabrtB : emit failure
26          end
27       end
28    end
29 end
```

The code for a receiver module above works by making a choice on which of signals *dataA* and *dataB* to respond to and then behaving like that was the only rendezvous being attempted. As a consequence of the deterministic nature of *Multiclock* ESTEREL, the implementation above is constrained to select signal *dataA* whenever both signal *dataA* and signal *dataB* occur together. Also note that the decision to implement choice in this manner precludes the possibility of allowing *mixed* guards (*i.e.* mixed input and output channels) as this could lead to implementation induced deadlocks.

```
01 module sender :
02 output dataA, sabrtA, confA, dataB,
03       sabrtB, confB, success, failure;
04 input rcptA, rabrtA,
05       rcptB, rabrtB, labrt;

06    present labrt else
07       emit dataA;
08       emit dataB;
09       await case
10          labrt :  emit sabrtA
11          emit sabrtB
12          rcptA : emit confA
13          emit sabrtB
14          emit success
15          rcptB : emit confB
16          emit sabrtA
17          emit success
18          rabrtA : present rabrtB then
19             emit failure
20          else
21             await case
22                labrt :  emit sabrtB
23                emit failure
24                rabrtB : emit failure
25                rcptB : emit confB
26                emit sabrtA
27                emit success
28             end
29          end
30          rabrtB : present rabrtA then
31             emit failure
32          else
33             await case
34                labrt :  emit sabrtA
35                emit failure
36                rabrtA : emit failure
37                rcptA : emit confA
38                emit sabrtB
39                emit success
40             end
41          end
42       end
43    end
44 end
```

The structure of the sender code above implementing *guarded rendezvous* makes it amply clear that it depends on the receiver to limit the choice. The sender emits both signal *dataA* and signal *dataB* and awaits a response of either signal *rcptA* or signal *rcptB*. Receipt of either one of these signals guarantees that the *rendezvous* can now successfully terminate or will be locally preempted, in which case, it is acceptable to issue an abort on the other channel. Note that receipt of an abort message from a receiver causes the sender to await a response from the remaining receiver unless there already is an abort message from it too.

It is obvious that in both the sender and receiver modules presented, only one *rendezvous* can terminate successfully. It is also easy to see that this implementation does not introduce any deadlock.

In the above, we have illustrated that the framework of *Multiclock* ESTEREL embed the CRP as an instance. Since the framework does not need to distinguish various levels of concurrencey and also the constraints of causality of ESTEREL (due to the underlying asynchronous interfaces), it can be generally used the specification and verification of a hierarchical network of processes.

5. CONCLUSIONS

We have shown how the paradigm of *Multiclock* ESTEREL and the protocols used for the implementation can be used for the specification and implementation of distributed systems and in particular, its advantages for the specification of globally asynchronous and locally synchronous systems (often called GALS in the hardware literature). In section 3, we illustrated the specification of quasi synchronous systems [5] in *Multiclock* ESTEREL and showed how such an analysis can be augmented for buffer requirements with the given underlying architectural constraints; in fact, such an analysis can be augmented with the analysis relative to various communication mechanisms as discussed in [13]. In section 4, we illustrated CRP [4] as an instance of *Multiclock* ESTEREL . Using these illustrations, it is possible [11] to effectively use *Multiclock* ESTEREL for the specification, design and implementation of time-triggered architectures [7] that are widely used for distributed embedded systems including web-based interactions. The study of the relationship of *Multiclock* ESTEREL with other clocked frameworks such as Lustre-scade [5], Mode automata [6] with the view of designing distributed systems is in progress.

References

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

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

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

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

[5] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with lustre. In *SafeCom99*, Sept 1999.

[6] Yann Remond F. Maraninchi. Mode-automata: About modes and states for reactive systems. volume ESOP of *LNCS*. Springer-Verlag, 98.

[7] H. Kopetz, M.Kucera, D. Millinger, C. Ebner, and I. Smaili. Interfacing time-triggered embedded systems. In *Proc. of the Int. Symposium on Internet Technology*, pages 180–186, Taipei, Taiwan, Apr 1998.

[8] B. Rajan and R.K. Shyamasundar. Networks of preemptible reactive processes: An implementation. In *Int. Conf. on VLSI Design*, New Delhi, India, Dec 1995.

[9] 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.

[10] B. Rajan and R.K. Shyamasundar. *Multiclock* ESTEREL: An asynchrnous framework for asynchronous design. In *Int. Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[11] Basant Rajan, S.A. Dahodwala, T.M. Topiwalla, and R.K Shyamasundar. Time triggered systems in *Multiclock* ESTEREL'Manuscript, 2000.

[12] S. Ramesh R.K. Shyamasundar. Semantics and verification of hierarchical programs. volume 999 of *LNCS*, pages 436–461. Springer-Verlag, 1995.

[13] T. Pitassi RK Shyamasundar, KT Narayana. Semantics of nondeterministic asynchronous broadcast networks. *Information and Computation*, 104:215–252, Jun 1993.

6. APPENDIX: INFORMAL SEMANTICS

Being a synchronous language, *Multiclock* ESTEREL statements follow a *zero delay* semantics and are therefore considered to execute (and terminate) instantaneously. The only exception to this being an explicit *pause* statement that terminates only on the next occurrence of the current clock.

Unlike ESTEREL, where there is an implicit *global* clock (the *tick*) at which the entire ESTEREL program makes transitions, in *Multiclock* ESTEREL modules are independently and explicitly clocked. This feature is a major deviation from ESTEREL and accounts for almost all the enhancements that *Multiclock* ESTEREL offers. Clocks can be specified as boolean expression over input signals.

We will now describe the individual constructs that *Multiclock* ESTEREL programs are built from. We shall let p, q range over programs and s over signals in the rules that follow.

*Statement : **nothing***
This statement does nothing and terminates instantaneously.

*Statement : **pause***
Waits until the next occurrence of the clock to which it is bound. In the classical ESTEREL it corresponds to waiting for the next input event.

*Statement : **emit** s*
This statement causes an instantaneous broadcast emission of the signal s and terminates instantaneously with the current clock.

Statement : p;q
The statement starts executing p immediately and behaves as it will as long as statement p (or its derivatives) remains active. Once p terminates control is passed to q and the behavior thereafter is defined by it. Due to the synchrony hypothesis, q starts in the same instant in which p terminates. For example, " `emit s1 ; emit s2` " results in the simultaneous broadcast of `s1` and `s2` and the statement terminates instantaneously.

*Statement : **loop** p **end***
The body of the loop p is started *instantaneously* the first time *and* whenever statement p or its derivatives terminates. For this reason, it is necessary that p does not terminate instantaneously. The `loop` statement never terminates but could be *exited* due to preemption.

Statement : ***present*** *s* ***then*** *p* ***else*** *q* ***end***
The statement tests for the presence of signal *s* and executes either statement *p* or statement *q* depending on whether signal *s* is present or absent. Both the test and the transfer of control to either statement *p* or statement *q* are done synchronously with the clock.

Statement : $p \,||\, q$
Each of the composed statements share the same clock and make independent transitions of their own. Parallel components can communicate only through shared signals. Variables cannot be shared by parallel modules. Both the component statements are initiated synchronously with the initiation of the construct and the construct terminates synchronously with the termination of the component that terminates last.

Statement : ***signal*** *s* ***in*** *p* ***end***
The statement *p* is started immediately with a fresh signal *s* local to the block *p* overriding other bindings. It ensures that any reference to signal *s* within the statement block *p* is bound to the local signal *s* and this signal is not exported outside the block.

Statement : ***abort*** *p* ***when*** *s*
This constructs implements strong preemption in *Multiclock* ESTEREL. Statement *p* is started immediately and terminates either when statement *p* terminates normally or whenever the signal *s* occurs in the future, in which case the whole statement gets *preempted* instantaneously. That is, if signal *s* occurs during the execution of statement *p*, all processing scheduled for the current instant by statement *p* is abandoned.

Statement : ***weak abort*** *p* ***when*** *s*
Multiclock ESTEREL also provides for an alternate form of preemption termed weak preemption. The salient difference being that all processing scheduled for the current instant by statement when preemption is effected is allowed to execute to completion performing the *last wills.*

Statement : ***suspend*** *p* ***when immediate*** *s*
This 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.

Statement : ***trap*** *T* ***in*** *p* ***end***
The body **s** is started immediately and behaves as *p* until termination or an explicit exit. If statement *p* terminates, so does the **trap** construct. If *p* exits *T*, then the **trap** terminates in the same instant after completing the concurrent reactions. If *p* exits an enclosing trap *U*, then the exit is propagated upwards appropriately.

Statement : ***exit*** *T*
The statement exits the trap *T* instantaneously terminating the corresponding trap statement unless an enclosing trap is exited synchronously.

Statement : ***newtick*** *s* ***in*** *p* ***end***
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 will in turn terminate 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.

Statement : ***weak newtick*** *s* ***in*** *p* ***end***
This is a variant of the newtick construct introduced above. This construct starts synchronously at the current clock and initiates execution of statement *p* on clock *s*. However, statements in statement block *p* are *not* insulated from any suspensions imposed on this construct. The keyword **weak** is used to signify that the change of clock is not sufficient to protect the enclosed statements from suspensions imposed on the construct.

Some of the derived constructs which are used widely are given in the following. The **halt** construct is given by "**loop** pause **end**". Similarly, the derived construct "**await** s" can be written as "**abort halt when** s". Similarly, "**sustain** s" can be rewritten as " **loop emit** s ; **pause end**. A module can be instantiated in another module through the **run** statement. For example " **run** DRAM" replaces the **run** statement by the body of the DRAM module.