Multiclock Esterel

Gérard Berry[1] and Ellen Sentovich[2]

[1] Esterel Technologies, 885 av. J. Lefebvre, 06270 Villeneuve-Loubet, France
`Gerard.Berry@esterel-technologies.com`
[2] Cadence Berkeley Laboratories, 2001 Addison Street, Berkeley, CA
`ellens@cadence.com`

Abstract. We present the Multiclock Esterel language, which extends the synchronous language Esterel to multiple clock zones. While Esterel is good for compact single-clocked hardware or software designs, modern electronic designs are growing rapidly and they can no longer be designed in a monolithic fashion. Problems such as clock distribution, complexity, and power limitations have led designers to construct designs in a modular, multiple clock fashion. Multiclock Esterel is designed precisely to address this design style. It is a natural extension of Esterel, and retains its strong synchronous semantics and internal determinism. Statements driven by different clocks communicate through two special devices called the sampler and the reclocker. Multiclock Esterel should be understood as a preliminary language proposal meant to study multiclocking. It has not yet been validated by large experiments.

1 Introduction

The Esterel synchronous reactive language [5,4,6], which we call Classic Esterel in this paper, is based on a single-clock instantaneous interaction principle. The behavior of a program is defined by a sequence of reactions to input sequences. The execution environment decides when the program is provided an input, and a reaction is viewed as the simultaneous production of an output response to an input event. We call *ticks* the instants in which reactions occur, and *master clock* the sequence of these instants (it is a logical clock and no time regularity is required). This synchronous view has proved useful for a fairly large class of reactive applications such as process or human-machine interface controllers, communication protocols, and hardware circuits. For single-clocked synchronous circuit synthesis [13], the master clock is simply the circuit's clock [2,4]. Then, reaction to an input is not instantaneous, but it is guaranteed to be computed before the next clock tick; this is the best practical approximation of perfect synchrony.

In practice, synchronous single-clocking makes perfect sense for *compact* systems for which it is reasonable to pretend that all the system's component fit within a single circuit or within a single micro-computer for software applications. However, modern electronic designs are characterized by unwieldy size, clock distribution complexity, and power limitation. Their hardware, software, or mixed implementation tends to abandon the classical single-clock framework in favor of a multiclock one, each clock controlling a circuit zone or a local processor. A good example of this is described in [12], where a single global clock was abandonned for a design style with multiple local clocks. Communication between clock zones becomes a critical issue and is carefully controlled either by inserting special devices such as latches or by fine timing analysis.

In this paper, we present an extension of Esterel for such multiclock systems, which we call Multiclock Esterel. The new language proposal extends Classic Esterel in two ways. First, it deals with a set of primary clocks instead of a single global clock, each individual reactive module being clocked by one of the primary clocks. Second, it introduces two new communication primitives between clock zones, the *sampler* and the *reclocker*, which make it possible to send an information on a given clock and to receive it on another clock. These extensions are quite minimal, semantically well-defined, and physically reasonable.

Semantically speaking, Multiclock Esterel is still a synchronous language in the sense that clock ticks of different clocks remain comparable in time and information transmission remains instantaneous. We thus retain the strong determinism property which has proved so useful in the synchronous language framework: the behavior of a program is completely determined once the clock timings and the input flows are known. There is a larger amount of external non-determinism (the timings of the clocks), but still no internal non-determinism. Therefore, Multiclock Esterel is not a language for large asynchronous distributed applications for which such notions are not applicable. Since words are heavily overloaded in this field, we shall speak of *single clocked* and *multiclocked* parts of systems, and we shall avoid using the ambiguous word *asynchronous*.

As in [4,3], we concentrate on the kernel Multiclock Esterel calculus, ignoring software engineering issues related to full language development. Our goal is to develop the foundations of multiclock langages, and in particular to study preemption operators in the multiclock framework. In Sect. 2, we study how communication can be organized in a multiclock context, and we define the sampler and the reclocker.

In the rest of the paper, we study the Multiclock Esterel calculi. The syntax defines two kinds of terms: the *clocked reactive statements*, which are basically those of Classic Esterel, and the *multiclocked processes*. A multiclocked process it is either the pair of a reactive statement and of the clock that drives it, or a compound structure involving several of these clocked reactive statements driven by different clocks. The semantics is given by a modeling in Classic Esterel.

In Sect. 3 we begin with the *basic calculus*, where a multiclocked process is a flat network of reactive modules, each controlled by a given clock. In the *full calculus* in Sect. 3.3 we achieve full orthogonality by allowing processes to be recursively launched from within reactive statements and vice versa. We can then define how an arbitrary multiclocked process can be preempted by a reactive preemption statement. We also enrich processes by allowing sequencing and looping of them. We give a simple example of preemption control of a fast process by a slow one. We conclude in Sect. 4.

2 Clocks, Signals, and Communication

2.1 Clocks

We consider a set $C = \{c, c', c_0, c_1, \ldots\}$ of *clocks*. Intuitively, a clock is a primitive object that delivers ticks in sequence. Clocks are logical and need not be periodic in time. For instance, a clock can be generated by a quartz, by the depressing of a button, or by the decision to call a program. Clocks determine *tick sequences*, which can be dealt with in two ways:

– *Discrete time:* A tick sequence is a sequence of *global ticks*, where a global tick is characterized by a set of clock names:

$$c_1,\ c_2,\ c_1{\cdot}c_2,\ c_1,\ c_3,\ c_2,\ c_1{\cdot}c_2{\cdot}c_3, \ldots$$

Here, $c_1 \cdot c_2$ means that the clocks c_1 and c_2 tick simultaneously. Allowing simultaneity is useful for several reasons: we may want to set two clocks equal in a program, we may want to extract a clock from another one, or we may simply want to deal with coincidental simultaneity of *a priori* independent clocks. When needed, we can require clocks to be non-simultaneous for all of their ticks by asserting an *exclusion relation* of the form $c\#c'$ as we do for signals in Classic Esterel.

– *Continuous time:* here time is continous and a clock is defined by a finite or infinite increasing sequence of real numbers representing the instants at which the clock ticks. Ticks of distinct clocks can coincide in time.

In our framework, both models are essentially equivalent, but one or the other may be more natural in intuitive or formal descriptions. It is easy to go from one model to the other: from discrete time to continuous time, associate any increasing sequence of reals with a discrete tick sequence; conversely, from continuous time to discrete time, extract the clock set sequence from the increasing sequence of real numbers obtained by taking the union of the individual clock sequences.

2.2 Signals and Communication

In Classic Esterel, there is a single implicit clock c for all statements, which makes communication very simple. In Multiclock Esterel, we organize communication between different clock zones by sampling and reclocking the exchanged signals. We explain the semantics of communication and discuss the appropriate software or hardware devices for implementing it.

Communication in Classic Esterel. Two Classic Esterel statements p and q communicate by instantaneously propagating signals: for example, at some tick of c, p emits s and q instantaneously receives s. A *pure signal* is characterized by its broadcast *status* at each tick of c, which is either *present* (*high*) or *absent* (*low*). A pure signal s is absent by default, and it is set present at a tick of c if and only if at least one "`emit` s" statement is executed at that tick. Simultaneous emission of s by concurrent `emit` statements is allowed and simply sets s present.

In addition to the status information, a *valued signal* broadcasts a value belonging to some data type, with the restriction that a value change can occur only when the signal is present. A valued signal is emitted by executing an "`emit` $s(e)$" statement, where e is a value expression. The value of s is read by the expression '`?`s'. To give a meaning to concurrent emission of a valued signal, one associates with it an associative and commutative *combination function*, which is used to combine all the separately emitted values into one final value. For example, "`emit` $s(1)$ `||` `emit` $s(2)$" results in `?`$s = 3$ if combination is done by addition.

Pure Communication in Multiclock Esterel. In Multiclock Esterel, we retain the principle of instantaneous reaction to clock ticks and communication by signals, but we may emit and receive signals according to several clocks. Instants become relative to clocks. Consider two processes A and A' running reactive statements respectively clocked by two different clocks c and c'. Each process is locally single-clocked: communication within A (resp. A') is governed by c (resp. c') as in Classic Esterel. To communicate with each other, A and A' can share an instantaneously broadcast signal s without sharing a clock: the emitter A will emit s at some tick of c, and the receiver A' will receive s at some tick of c'. Instantaneous propagation means that A' will receive s at the very first tick of c' that follows the emission tick of c, however close these ticks are in time, up to tick equality for which A' will also receive s as in Classic Esterel.

Which status s' of s will A' receive at c'? This question doesn't arise in Classic Esterel, since there is only one clock for the emitters and receivers: communication occurs only on ticks, and statuses outside ticks are irrelevant in the model (they are of course relevant in implementations). In Multiclock Esterel, we view signals as continuous objects and statuses as being held high or low during the emitter's clock cycle according to their emission statuses at the clock tick, as pictured in Fig. 1. We think there are two fundamental choices for s':

1. *Sampling*: s' is present at c' and set high for the c' cycle if and only if s is high precisely at c'.
2. *Reclocking*: s' is present at c' and set high for the c' cycle if s has been high at least once at any time since the previous occurrence of c'.

Sampling is the classical operator of Signal Theory. It is like taking a snapshot of the signal at c'. Reclocking has a built-in memory, and amounts to taking a long exposure on s not to miss any occurrence. This can be necessary if the sender's clock c is faster than the receiver's clock c': for instance, if the sender is sending transient alarms, and if the receiver worries about the emission of at least one transient alarm since its last tick, sampling is not enough to catch the required information at the receiver and reclocking is mandatory (another example will be given in Sect. 3.5). In Multiclock Esterel, we make both choices available.

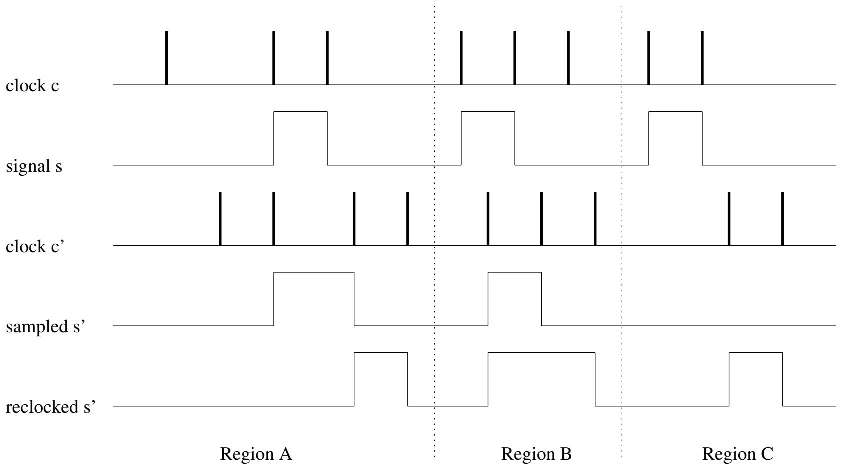

Fig. 1. Communication devices in continuous time

As usual with synchronous languages, there are boundary problems to be solved:

– *Sampling:* if the ticks of c and c' coincide, the sampled status is the *new* status of s w.r.t. c, i.e. the one which will be valid for s in M until the next tick of c.
– *Reclocking:* if the ticks of c and c' coincide, the new status of s in M is not considered for s' in M'; reclocking checks that s has been emitted since the *previous tick of* c *included, current tick excluded.*

The other boundary cases can be obtained simple Boolean combinations of the above ones. If $c' = c$, sampling is simply transparent, while reclocking amounts to taking the previous status of the signal, as for the Classic Esterel `pre` operator.

In Fig. 1, the following three differences between sampling and reclocking cases are illustrated:

Region A: the reclocked signal can be delayed in its rising transition with respect to the sampled signal when the clocks c and c' coincide on a rising edge of s: reclocked s' cannot see the current value of s until the next tick of c', while sampled s' can.

Region B: the reclocked signal is delayed in its falling transition with respect to the sampled signal: reclocked s' remains high if s was high at any time since the last tick of c', while sampled s' goes low as soon as s is low at a tick of c'. (Region B illustrates this for the case in which c and c' do not coincide for s's falling transition. The same is true when c and c' do coincide, but we have not shown this case.)

Region C: The reclocked signal can detect spurious high values for s that might be missed by the sampled signal.

An example where a reclocker is needed is presented in Sect. 3.5. The extended version of this paper, available from the authors, contains another illustrative example of sampling and reclocking. The example is a video game where a process controls a joystick and another process controls the game proper. The joystick and game processes have different clocks. The joystick process sends position signals to the game, which samples them. The joystick also sends a reset signal to the game. This signal must be reclocked by the game, otherwise it could be lost.

Value Communication in Multiclock Esterel. As far as valued signals are concerned, sampling extends trivially by inputting the current value of a signal in addition to its status at the receiver's clock tick. The value may have changed many times since the previous tick, but we only sample the most recent one, as for the status; this is exactly what sampling means in signal theory. The memory effect of reclocking makes defining the value of a reclocked signal less obvious; a natural extension of the Classic Esterel concept of multiple simultaneous emission would be to combine all the successive values received since the previous tick by an associative-commutative operation. Since we have no really good supporting example, we prefer to postpone the decision and to currently restrict Multiclock Esterel to pure or valued sampled signals and to pure reclocked signals.

Single Clocked and Multiclocked Signals. Given a clock c, we say that a signal is *clocked by* c if its status changes occur only on ticks of c. A signal which is emitted by reactive statements driven by a single clock c is always clocked by c. Sampling and reclocking always generates local views that are clocked by the receiver's clock.

An interesting question is whether or not we want to allow multiclocked signals. A pure multiclocked signal could be handled by merging the results of two different `emit` statements driven by different clocks, saying that s is high at time t if any one of the outputs of the `emit` statements is high at that time. The device to do this is a *wired or*. However, multiclocked merging does not extend trivially to valued signals. We said that multiply emitted values should be combined by an appropriate combination function. We should also combine the values if the ticks of two distinct emitter clocks happen to be simultaneous. This would be semantically meaningful, but we lack magical tricks for the implementation. Therefore, we choose to postpone this problem and to forbid multiclocked signals for the time being. We do not see this as a limitation: in most hardware systems, each signal is computed from a single clock zone.

2.3 Sampling and Reclocking Devices

Having seen that for Multiclock Esterel have two fundamental choices in reading a signal that passes from one clock zone to another, and having defined what sampling and reclocking semantically mean, we now study devices that can do the job for different styles of implementation. We name them generically the *sampler* and the *reclocker*. To give an operational semantics of Multiclock Esterel, we first encode them within Classic Esterel. We then discuss hardware and software realizations.

Classic Esterel Encoding. To model Multiclock Esterel in Classic Esterel, we can write single-clocked programs where the tick is that of a fictitious global clock of the Multiclock Esterel programs, which is faster than all of the actual clocks. A Multiclock Esterel clock can then be modeled as an ordinary Esterel signal. (This is *modeling*, not programming, since this global clock will be inaccessible from within Multiclock Esterel programs.) The obtained Classic Esterel program is deterministic, which means that Multiclock Esterel has no internal non-determinism. However, the fact that Multiclock Esterel clocks have become free input signals leaves room for a large amount of external non-determinism.

We take the freedom of considering `C'` and `S'` as valid Esterel identifiers, using primes for the receiver side. The sampler takes as input the signal `S`, the receiver clock `C'`, and it returns as output the local receiver view `S'` clocked by `C'`:

```
module Sampler :
input S;
input C'; % the clock
output S';
loop
   present S then
      sustain S'
   end present
each C'
end module
```

Although the module is active on every tick of the fictitious global clock, sampling `S` only happens on ticks of the clock `C'`. On any tick of `C'` in which `S` is present, `S'` is emitted, and it keeps being emitted on every global clock tick (i.e. "continously in discrete time") until the next tick of `C'` arrives. Notice that `loop...each` is stongly preemptive, so that when `C'` occurs, the `sustain` statement is preempted without being executed, the loop loops, and the `sustain` statement is instantaneously restarted only if `S` is present. Therefore, we indeed sample the new value of `S` as required in the sampler specification.

The reclocker has the same interface. It can be coded as follows:

```
module Reclocker :
input S;
input C'; % the clock
output S';


signal MEM in
   loop
```

```
         await immediate S;
         sustain MEM
      each C'
   ||
      loop
         present pre(MEM) then sustain S' end
      each C'
   end signal
   end module
```

The first part of the parallel statement implements a memory to remember the occurrence of `S`. As soon as `S` goes high on a global clock tick, the internal signal `MEM` is set high, where it remains (by being emitted each global clock tick) until a tick of `C'` occurs. When `C'` occurs, the previous value of `MEM` w.r.t. the global clock is tested and `S'` is sustained for the next cycle if `MEM` was high. Notice that an `S` occuring at the same time as a `C'` is taken into account only in the next tick of `C'`, as required by the reclocker specification.

Software Implementation. A software implementation of Multiclock Esterel can follow the principles of Polis [1]. Clock ticks correspond to module activations, acting as Polis CFSMs. A sampler is a single memory cell in which writers write and readers read, i.e., a Polis 1-place buffer. A reclocker is a 1-place buffer with an additional bit set at each write and reset by the reader. For multiprocessor applications, atomicity conditions on activations, reads and writes are required as usual to provide a consistent view between readers and writers.

Hardware Implementation. In hardware, one can build a sampler and a reclocker using electronic devices such as *latches* and *flip-flops* (unfortunately, the terminology is a little sloppy in this field). For instance, the sampler can be built as a *positive edge-triggered D flip-flop* [10]. However, it is impossible to build a *perfect* sampler or reclocker, because of the standard metastability problem. For any sampling device, there is a time interval δ around the clock edge inside of which the sampled signal cannot itself have an edge. Otherwise, the device can enter a metastable state in which it incorrectly drives its output for some amount of time.

The metastability problem is not specific to Multiclock Esterel. It arises in any kind of harware multiclock design. In today's technology for multiclock design, a sophisticated timing analysis program is run to ensure that there are no violations of clock setup and hold times. A similar analysis should be made for a synthesized implementation of an Multiclock Esterel program. In this exploratory paper, we shall ignore these issues since we want to experiment with the technology-independent aspects of language design.

3 The Multiclock Calculus

In this section, we develop the Multiclock Calculus. We begin with some definitions of basic elements and the flat calculus. This is followed by the modeling of the flat calculus in Classic Esterel. We continue by addressing the issues of process sequencing and preemption. Finally, we give an example containing a slow module controlling a fast process.

Terms in Multiclock Esterel are divided into two categories, multiclocked *processes* and single-clocked *reactive modules*. Recall that Classic Esterel contains only reactive modules.

– Processes P, Q, etc., are elementary or compound. A process has a signal interface and a body. Elementary process bodies consist in executing a reactive module with a given clock and a given sampling / reclocking interface specification. Compound process bodies are built from elementary ones by concurrency and signal scoping declaration, and possibly by sequencing and looping in the extended calculi.
– Reactive modules M, N, etc., are as in Classic Esterel. Each reactive module is governed by a single clock, which is implicit in the module. The interface specifies the input / output signals. The body is a reactive statement. All the statements in kernel Classic Esterel are imported in kernel Multiclock Esterel. Only one statement is added, execution of a process.
– A *program* is defined by a set of clocks and a process.

For compound processes, clocks appear only in the leaf elementary processes, which in turn call clocked reactive modules. Therefore, there is no global clock accessible to the executable reactive statements: a reactive statement knows only the clock that runs it.

We first present the *flat calculus*, limited to putting classical reactive modules into a flat multiclocked parallel structure. Then, we present the full calculus, where we can recursively embed multiclocked processes into single-clocked reactive modules and conversely, up to any depth. In this fully orthogonal language, one can deal in a general way with preemption of multiclocked processes. As in [4], we use indifferently a mathematical style or a programming language style syntax.

3.1 The Flat Calculus

In the flat calculus, a process is limited to being composed of reactive modules driven by given clocks and put in parallel.

Processes. A process $P = (I, O).A$ has a list I of input signals, a list O of output signals, and a body. Process bodies are written A, B, etc. Their syntax is as follows, M denoting a reactive module defined below:

$c * M\,(I^m)$	`run M clock C input sample I ...`
$A \,\vert\, B$	$A \,\vert\vert\, B$
$A \setminus s$	`signal S in` A `end`

There are clock constaints on signals, which will be presented in Sect. 1.

In the $c * M\,(I^m)$ reactive module run statement, the decorated vector I^m of *input modes* specifies an exponent $m \in \{s, r\}$ for each input signal of M. The exponent defines how the signal is brought into the clock c: s means that the signal is sampled, r means that it is reclocked.

Reactive Modules. A reactive module M is defined by an interface specification and a body:

$$M = (I, O).p$$

The interface specifies the input signal vector I and the output signal vector O. The body is a reactive statement p[1], with syntax that of Classic Esterel:

0	`nothing`
1	`pause`
k (for $k > 1$)	`exit` t
$!s$	`emit` s
$s\,?\,p\,,\,q$	`present` s `then` p `else` q `end`
$s \supset p$	`suspend` p `when` s `end`
$p\,;\,q$	$p\,;q$
$p\,*$	`loop` p `end`
$p\,\vert\,q$	$p \parallel q$
$\{p\}$	`trap` t `in` p `end`
$\uparrow p$	
$p \setminus s$	`signal` s `in` p `end`

Notice that there are two concurrency operators '|', one for reactive statements and one for processes, and similarly two local signal declaration operators '\'. These operators perform the same kind of operation in both worlds, although the behaviors are technically different. There is no danger in overloading the symbols since one can always determine unambiguously from the syntax whether one is inside a process body or a reactive module body.

Signal and Clock Constraints. Signals are subject to the usual visibility rules. A signal refered to in a reactive statement must be an interface signal or a local signal in the enclosing reactive module. For any reactive module run $c * M\,(I^m)$ occuring in the body A of a process P, a signal s in M's interface must be declared as an interface signal of P or as a local signal in an enclosing local signal declaration $A \setminus s$.

Signals in a reactive module are always clocked on the module's implicit clock. Input signals are either sampled or reclocked by the module run interface; local and output signals are set by the `emit` statements which only acts on the module's clock.

Signals in a process must obey the following *clock consistency* rules: *all reactive modules that share a signal as output must be clocked by the same clock.*

[1] We use P for a process and p for a reactive statement, which may be confusing. The whole Esterel literature uses p for statements, a notation we keep here.

This requirement ensures that each signal is properly clocked by a single clock, as required in Sect. 2.2.

3.2 Modeling Flat Multiclock Esterel in Classic Esterel

We give the semantics of a flat Multiclock Esterel program by modeling its behavior in Classic Esterel. For this, we introduce a fictitious base clock faster than all Multiclock Esterel clocks, as in Sect. 2.3. We consider this fictitious clock as the Classic Esterel model base clock and we view each Multiclock Esterel clock c as a pure signal also named c in the Classic Esterel translation. We define a translation function T which translates any multiclocked process P into a Classic Esterel term $T(P)$.

The interface of $T(P)$ is simply that of P. The process body is translated in a trivial recursive way until reaching a clocked module run statement:

$$T(A \,|\, B) = T(A) \,|\, T(B)$$
$$T(A \setminus s) = T(A) \setminus s$$

We now translate a leaf explicitly clocked module run statement $c * M\,(I^m)$, $m \in \{s, r\}$, with $M = (I, O).p$. The basic idea is to introduce local views I' and O' clocked by c' of the input and output signals vectors I and O, ant to use three components in parallel: the appropriate translation of p, an input handler $TI(I^m\,,\, c'\,,\, I')$, which builds the local view I' of I according to the sampling or reclocking directives m, and an output handler $TO(O'\,,\, c'\,,\, O)$ which builds the output signals O from their local views O'.

The input handler puts in parallel individual signal handlers that sample or reclock input signals using the sampler and reclocker defined in Sect. 2.2:

$$TI(I^m\,,\, c'\,,\, I') = TI({i_0}^{m_0}\,,\, c'\,,\, i'_0) \,|\, TI({i_1}^{m_1}\,,\, c'\,,\, i'_1) \,|\, \ldots \,|\, TI({i_n}^{m_n}\,,\, c'\,,\, i'_n)$$
$$TI(i^m\,,\, c'\,,\, i') = \begin{cases} \mathit{Sample}(i,\, c',\, i') & \text{if} \quad m = s \\ \mathit{Reclock}(i,\, c',\, i') & \text{if} \quad m = r \end{cases}$$

The output handler puts in parallel individual signal sustainers $\mathit{Out}(o',\, c',\, o)$ defined in Classic Esterel as follows:

```
loop
   present O' then
      abort
         sustain O
      when C'
   else
      await C'
   end present
end loop
```

The effect of these sustainers is to clock the output signal on c', in the sense of Sect. 2.2. The definition of TO is:

$$TO(O'\,,\, c'\,,\, O) = TO(o'_0\,,\, c'\,,\, o) \,|\, TO(o'_1\,,\, c'\,,\, o_n) \,|\, \ldots \,|\, TI(o'_n\,,\, c'\,,\, o_n)$$
$$TO(o'\,,\, c'\,,\, o) = \mathit{Out}(o',\, c',\, o)$$

To translate p, we use two auxiliary operators. The "`await immediate` c" operator or $sync(c)$ waits for the first occurrence of c and terminates. The c-trigger operator, written "`suspend` p `when immediate not` c" in Esterel and $\bar{c} \mathrel{\dot{\supset}} p$ in short form, triggers p exactly at the instants where c is present, and freezes p at the other instants ($\bar{c}$ is the negation of c). These operators are defined by

$$\begin{aligned} sync(c) &= \{(c\,?\,1\,,\,2)\,*\} \\ \bar{c} \mathrel{\dot{\supset}} p \ &= sync(c)\,;\,\bar{c} \supset p \end{aligned}$$

Let $p\,[I'/I, O'/O]$ be p where I and O are renamed into I' and O'. The final translation is:

$$T(c * M\,(I^m)) = ((\bar{s} \mathrel{\dot{\supset}} p\,)\,[I'/I, O'/O] \mid TI(I^m\,,\,c'\,,\,I') \mid TO(O'\,,\,c'\,,\,O)) \setminus I', O'$$

Notice that the translation of p starts acting at the first tick of c', first instant included, because of the initial $sync(c)$ in $\bar{c} \mathrel{\dot{\supset}} p$.

3.3 Process Sequencing

In the flat calculus, termination of a reactive module is ignored within the process that holds the module since there is no process sequencing. It is sensible to exploit the existing termination information and to define concurrent process termination as termination of all the concurrent reactive modules they contain. In the Classic Esterel translation, we get termination detection for free: a process body A terminates when $T(A)$ terminates. In hardware, concurrent termination detection can be done using devices such as Muller C-elements [11].

Then, we can define process sequencing $A; A'$. Semantically, we just write $T(A\,;\,A') = T(A)\,;\,T(A')$. As before, the syntactic context disambiguates the '`;`' symbol between processes or reactive statements.

There are many elementary delays involved in process sequencing. Consider the trivial example $(c * 1\,())\,;\,(c' * 1\,())$. One first waits for the first occurrence of c to start the first 1 pause statement. This statement waits for one more c and terminates. Then, one waits for the first occurrence of c' to start the second 1 statement, this instant included. The second 1 terminates at the next occurrence of c'. The final translation of the above process sequence in Classic Esterel is

$$sync(c);\ \ await(c);\ \ sync(c');\ \ await(c')$$

Therefore, to implement process sequencing in practice, we have to implement $sync(c)$. We need a device with an input for the incoming control *in* and an output for the outgoing control *out*. The device should immediately set *out* high if *in* is high when c occurs, or else reclock *in* on c. The solution is to use the disjunction of a sampler and a reclocker with input *in* and clock c.

Having defined process sequencing, we can also define *process looping* $A*$ by $T(A*) = (\,T(A))\,*$, provided that A cannot terminate instantaneously, which is checked as in Classic Esterel.

3.4 Process Preemption: The Full Calculus

The last step to the full Multiclock Esterel calculus is to allow an abritrary multiclocked process P to be launched and preempted from within a clocked reactive module M clocked by c. This is done by adding a new reactive statement that starts P within M with a list of inputs and outputs and samples or reclocks the output signals of P on M's clock c since they become inputs of the body of M. The calculus and language syntax are as follows:

$\langle P\,(I, O^m)\,\rangle$ `process run` P `input I`
`output reclock O ...`

This new reactive statement can appear anywhere where a reactive statement can, including in a preemption context. Of course, the construction is fully orthogonal and recursive: P can be a general process that can itself run synchronous modules that themselves run other multiclock processes, etc.

Signal names in P's interface must exist in the scope of the $\langle P\,(I, O^m)\,\rangle$ statement within M's body, and signal binding is by name (in a real language, renaming facilities such as the ones used in the full Classic Esterel language should be added).

However, with the current definition of a reactive module interface, all signals in P's interface would be clocked by M's clock c: inputs come either from M's interface where they are either sampled or reclocked, local signals are directly clocked by c, while outputs are sampled or reclocked for M. Since P is an arbitrary process, it should also be able to view signals which are in the current process scope but not in M's interface. For this, we slightly change the reactive module syntax into $(I, O, H).p$, where H is a set of *hidden signals*. Hidden signals are neither sampled nor reclocked by M, and they cannot be used by M's reactive statements. They can only be passed by M as inputs or outputs to subprocesses. In the call $\langle P\,(I, O^m)\,\rangle$, we set $m = \perp$ for outputs of P which are hidden in M. To sample or reclock the non-hidden outputs of P for M, we rename them O' in P to keep the names O in M (beware, this time the prime is on the emitter side). The translation in Classic Esterel is:

$$T(\langle P\,(I, O^m)\,\rangle)c = T(P[O'/O]) \mid TO(O'^m\,,\, c'\,,\, O)$$

$$T(o'^m)c = \begin{cases} \mathit{Sample}\,(o',\, c,\, o) & \text{if} \quad m = s \\ \mathit{Reclock}\,(o',\, c,\, o) & \text{if} \quad m = r \\ 0 \quad \text{if} \quad m = \perp \end{cases}$$

This very simple translation tells everything about the semantics of the construct, and especially about subprocess preemption. For example, consider the Multiclock Esterel reactive statement

$$s \supset \langle P\,(I, O^m)\,\rangle$$

within a reactive module M clocked by c. When control reaches the statement, P is started autonomously and it communicates with M through its input /

output ports, where M sees all non-hidden signals as clocked by c. If M receives s from the environment or emits it internally, s is sustained for the whole clock cycle of c in the Classic Esterel translation. Therefore, $T(P)$ is suspended for the whole clock cycle as one expects. If, for some reason, M exits a trap that kills P, then $T(P)$ instantaneously dies as any other Classic Esterel term.

3.5 The Bureaucrat Example

Here is an example where a slow module called `Bureaucrat` controls the life and death of a fast process `Worker`. We assume that `Worker` reads an input flow `InFlow`, writes an output flow `OutFlow`, and sends a signal `Done` when its computation has finished. The bureaucrat wakes up every hour and returns `OK` and goes home if `Worker` has finished in the last hour. Of course, by construction, the bureaucrat does not care about the work being done with `InFlow` and `OutFlow`, which are hidden to him. After five hours, if `Worker` has not finished, the bureaucrat kills it, reports `FIRED`, and goes home. In concrete syntax, the program may look like:

```
clock Hour;
clock WorkClock;

process Global :
input InFlow;
output OutFlow;
signal OK, FIRED in
   run Bureaucrat clock Hour input InFlow
                             output OutFlow, OK, FIRED
end signal
end process


module Bureaucrat :
output OK, FIRED;
hidden InFlow, OutFlow;
signal Done in
   weak abort
      process run Worker input InFlow output OutFlow,
                         reclock Done
   when
     case Done do
       emit OK
     case 5 tick do
       emit FIRED
   end weak abort
end signal
end module
```

assuming that `Worker` itself involves one or more reactive modules clocked by the fast clock `WorkClock`.

Notice that bureaucrat termination depends on a `weak abort` preemption statement [5]. Every hour, i.e. on each of its ticks, the bureaucrat checks whether `Done` has occurred since the previous hour using the `reclock` directive (the worker goes home when done and does not sustain `Done`, hence the need to reclock this signal). If so, `OK` is emitted and the bureaucrat terminates. Otherwise, the counter from 5 in the second case is decremented. If the count reaches 0, `Worker` is killed, `FIRED` is emitted and `Bureaucrat` terminates.

3.6 Notes

In full Multiclock Esterel, one should allow direct reactive module inclusion within a module body as in full Classic Esterel. In this case, the clock of the included module is as usual that of the caller. To include a reactive module with a different clock, one needs to use an intermediate process. Process / module strict alternation makes signal usage very clear and is one of the main characteristics of Multiclock Esterel.

The physical hardware implementation of multiclocked preemption is not fully explored yet. The main idea is of course to "gate the clocks" when passing them through `Worker`, but one has to be careful about the numerous boundary problems which can occur, because of clock simultaneity for instance.

4 Conclusion

We have presented the new Multiclock Esterel language proposal that extends Classic Esterel to multiclock systems. Although the language makes it possible to write very complex behaviors including well-defined multiclocked process preemption, it is technically a simple extension of Classic Esterel. We believe that this is a good sign of semantic soundness. It is too early to claim that the language is really well-suited to real-life multiclock systems and that it can be correctly and efficiently implemented in hardware or software. Experiments are on the way.

We did not discuss causality issues and the handling of combinational cycles [4]. Within each node, they are as in Classic Esterel. Between nodes, there is no extra issue if the clocks are declared exclusive since there is always a positive delay from one clock zone to another one. If clocks ticks can be simultaneous, there can be nasty cycles beteen clock zones, which we have not studied yet.

Similar techniques can be used for other synchronous languages such as Lustre [9] and Signal [8], making their nodes communicate through sampler and reclockers. These languages are much simpler since they do not support preemption. See for example [7] for the use of samplers in distributed continuous control applications.

Acknowledgements. We thank Paul Caspi (Verimag), Michael Kishinevsky (Intel), and the reviewers for their helpful comments and suggestions.

References

1. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publishers, 1997.
2. G. Berry. Esterel on Hardware. *Phil. Trans. R. Soc. London A*, 339:87–104, 1992.
3. G. Berry. Preemption in concurrent systesms. In *Proc. FSTTCS93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.
4. G. Berry. *The Constructive Semantics of Esterel.* Draft book, preliminary version available from `http://www.esterel.org`, 1995-1999.
5. G. Berry. *The Esterel v5 Language Primer.* Available from `http://www.esterel.org`, 2000.
6. G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
7. P. Caspi, C. Mazuet, and N. Reynaud-Parigot. About the design of distributed control systems: the quasi-synchronous approach. In *Proc. Safecomp'01*, September 2001.
8. P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming Real-Time Applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.
9. N. Halbwachs, P. Caspi, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.
10. Randy H. Katz. *Contemporary Logic Design.* Benjamin / Cummings, 1994.
11. David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.
12. S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at 3.3-4.5MHz. In *Proceedings of the ISSCC*, February 2000.
13. E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc of the ICCD*, pages 328–333, October 1992.