Preemption Primitives in Reactive Languages *
(*A Preliminary Report*)

Sophie Pinchinat**, Éric Rutten**, R.K. Shyamasundar***

Abstract. In this paper, we study preemption primitives in reactive languages such as ESTEREL and SIGNAL (and its extension SIGNAL*GTi*) in a common framework. This enables us to compare behavioural/structural *expressive* powers of different languages and gives an insight into the complementarity of different control and data-flow abstractions in the reactive languages. Such a study also provides a basis on which a basic set of preemption primitives can be incorporated in reactive languages from the point of view of expressive completeness.

1 Introduction

A wide spectrum of operators for concurrency, synchronization and communication has been developed in [Mil 89, Sim 85]. These works have concentrated on arriving at notions of expressiveness and completeness starting from a basic set of combinators. A notion of completeness has been derived for the synchronous family in [Sim 85] in the sense that any operator which can be defined by rules of action which obey certain natural conditions can also be defined directly in terms of the basic combinators of the calculi such as Meije [Sim 85] and SCCS [Mil 89].

In the context of reactive programming, Gerard Berry has applied with success the Meije calculus for the design and verification of ESTEREL. One of the principal operators that plays a significant role in some form or the other is the process preemption operator – which is the process of controlling the life of a process. Berry [Ber 93] has argued the need and importance of *process preemption* for the specification of reactive systems and has elucidated such primitives available in the synchronous programming language ESTEREL. In this paper, we discuss various forms of primitives for process preemption that have been in use in various process calculi [Hoa 85, Mil 89] and various reactive languages such as ESTEREL and SIGNAL and study how the various forms can be derived essentially from the basic preemption primitives and other basic process calculi combinators. The main motivation for the study has been to:

* Work supported by IFCPAR (Indo-French Center for the Promotion of Advanced Research), New Delhi.

** EP-ATR Group, IRISA, Campus de Beaulieu, F-35042, Rennes, France, e-mail: {pinchina, rutten}@irisa.fr

*** Computer Science Group, Tata Institute of Fundamental Research, BOMBAY-400 005, India, e-mail: shyam@tcs.tifr.res.in

- show how the seemingly different preemption operators can be obtained by a set of basic preemption primitives and compositional operators (or combinators) along with recursive definitions.
- provide a formal basis as to why reactive languages following imperative or data flow style are in fact equivalent and show how the different abstractions are suitable with different styles.
- lead to the understanding of the issues of expressiveness and completeness.

In the sequel, we study preemptive structures of ESTEREL, and those of SIGNAL and its extension with preemptive structures SIGNAL *GTi*[4] in the framework of a synchronous calculus and provide an answer to some of the questions raised above. These are preliminary results of a larger study, which is being attempted at in providing a unified framework for reactive languages and providing a precise comparison of the different sets of primitives available, and the relations among them [PRS 95].

The rest of the paper is organized as follows: Section 2 briefly introduces the basic operators of MEIJE and other derived operators which will be used in the sequel, and we derive a spectrum of preemption primitives (such as interrupts, suspend, when etc.) that are available in various reactive languages in the MEIJE calculus. In Section 3, we consider the different preemption mechanisms of ESTERELfollowed by a description of preemption mechanisms of SIGNAL and SIGNAL *GTi* in section 4. Section 5 discusses the expressive power of operators, and makes comparisons between the different approaches of the languages.

2 Operators of the Process Calculus

In this section, we briefly introduce the MEIJE operators and other operators that are commonly used in an algebraic setting. We follow the usual conventions of [Mil 89, Sim 85]. First, we briefly discuss the MEIJE calculus and discuss several process calculi operators. We will discuss further how the various preemptive suspension operators that exist in synchronous languages such as ESTEREL and SIGNAL can be obtained from such primitives and make a relative comparison of the various operators. With this background, we show how the various preemptive and suspension operators can be derived from each other.

2.1 MEIJE Calculus and Other Compositional Operators

MEIJE actions consist of a free commutative group $\mathcal{G} = (\mathcal{A}, \cdot, \tau, \bar{})$. The infinite set $\mathcal{A} = \{a, b, c, ..., \tau\}$ is a set of *actions* with a special symbolœ τ, denoting the neutral element of $\mathcal{G}$. Elements of $\mathcal{G}$, called *events*, written α, β are products of positive (e.g. a, b) and negative (e.g. $\bar{a}, \bar{b}$) actions with $a \cdot \bar{a} = \tau$ for all $a \in \mathcal{A}$. For example, an action a can be an act of communication along a channel c carrying

[4] *GTi* stands for *Gestion de Tâches et d'intervalles*, the french for *T*asks and *i*ntervals Management.

a message of value v; it is then written in a more structured fashion like $c(v)$, as used in Section 4.

We say an event α is in *reduced form* if there are no occurrences of a and $\bar{a}$ together in α, for some $a \in \mathcal{A}$. In other words, we can get a reduced form of α by equating every pair of the form $a \cdot \bar{a}$ by the identity denoted τ. Note that, $\alpha \cdot a \cdot \bar{a} = \alpha$ (as it is a commutative structure, the other combinations follow). We shall write $\hat{\alpha}$ the reduced form of α - note that it is unique up to commutativity. Events can be seen as multisets; therefore, we shall use classical notations like $\in$, $\cap$, $\backslash$, etc... For example, $a \in b \cdot a \cdot a$, and $\bar{b} \cdot a \cdot a \cap a \cdot \bar{b} = a \cdot \bar{b}$, as well as $a \cdot a \cdot \bar{b} \backslash\, a \cdot \bar{b} \cdot c = a$. Intuitively, actions can be thought of as emissions or receptions of signals; τ classically denotes the silent action. The events describe the signal-environment in which the system reacts. Operator '$\cdot$' is used for instantaneous product of actions (simultaneity).

$$\text{(a)} \quad \alpha \xrightarrow{\alpha, \mathcal{E}} \texttt{nothing} \qquad \text{(b)} \quad P \xrightarrow{\alpha, \mathcal{E}} P' \tag{1}$$

$$\frac{P \xrightarrow{\alpha, \mathcal{E}} P'}{P||Q \xrightarrow{\alpha, \mathcal{E}} P'||Q} \quad \frac{Q \xrightarrow{\beta, \mathcal{E}} Q'}{P||Q \xrightarrow{\beta, \mathcal{E}} P||Q'} \quad \frac{P \xrightarrow{\alpha, \mathcal{E}} P' \quad Q \xrightarrow{\beta, \mathcal{E}} Q'}{P||Q \xrightarrow{\alpha \cdot \beta, \mathcal{E}} P'||Q'} \tag{2}$$

$$\text{(a)} \;\; \alpha : P \xrightarrow{\alpha, \mathcal{E}} P \qquad \text{(b)} \;\; \frac{P \xrightarrow{\alpha, \mathcal{E}} P' \quad s \notin \alpha}{P \backslash s \xrightarrow{\alpha, \mathcal{E}} P' \backslash s} \qquad \text{(c)} \;\; \frac{P \xrightarrow{\alpha, \mathcal{E}} P'}{s * P \xrightarrow{s \cdot \alpha, \mathcal{E}} s * P'} \tag{3}$$

Processes, or reactive systems, are built up from elementary processes and process combinators. An elementary process is either `nothing`, the process doing nothing, or an event α of $\mathcal{A}$. An elementary process α has only a one-step behavior: it produces α and becomes the process `nothing`, as in (1.a). More generally, a process P performs an action α in the context of a signal environment $\mathcal{E}$, then becomes a process P', as written in (1.b). The environment $\mathcal{E}$ has the same meaning as in ESTEREL [Ber 93, BRS 93]: it is the set of all the signals produced at that instant by the whole program of which P is part. This set gives the global information about the presence and the absence of signals; this is needed when the choice of the transition to take involves conditions on the absence of signals. In particular, $\alpha \subseteq \mathcal{E}$. The environment is denoted by the second component ($\mathcal{E}$) of the labels in the transition rules.

The most important combinators are *Prefixing* (:), *Parallel composition* (||), *Ticking* ($*$) and *Restriction* ($\backslash$). Their definitions are given in Equations (1,2,3). It may be pointed out that '$\cdot$' should not be confused with the prefixing operator of CCS [Mil 89]. In this paper, for prefixing, we shall use ':' as in SCCS [Mil 89] and MEIJE[Sim 85]. Prefixing, parallel composition and restriction are interpreted classically. The *ticking* is an operator that allows to synchronize on each behavioral step of an agent, without prejudging on the nature of its behaviour. Operationally it $s * P$ can be interpreted as: all the actions of P are linked to action s.

Regarding parallel composition, two more specific combinators can be defined: the *Synchronous product*, written $\times$ (as in SCCS), where the two involved processes must move together, and the *Interleaving composition*, written $|$, where only one process moves at a time. Their formal definition is given in Equation (4). It follows that: $P \times \texttt{nothing} \equiv \texttt{nothing} \equiv \texttt{nothing} \times P$ and: $P \mid \texttt{nothing} \equiv P \equiv \texttt{nothing} \mid P$.

$$\text{(a)}\ \frac{P \xrightarrow{\alpha,\mathcal{E}} P' \quad Q \xrightarrow{\beta,\mathcal{E}} Q'}{P \times Q \xrightarrow{\alpha\cdot\beta,\mathcal{E}} P' \times Q'} \qquad \text{(b)}\ \frac{P \xrightarrow{\alpha,\mathcal{E}} P'}{P|Q \xrightarrow{\alpha,\mathcal{E}} P'|Q} \quad \text{(c)}\ \frac{Q \xrightarrow{\alpha,\mathcal{E}} Q'}{P|Q \xrightarrow{\alpha,\mathcal{E}} P|Q'} \quad (4)$$

An interesting relationship between $\times$ and the *ticking* can be expressed is given by (cf. [Sim 85]): $P \times Q \equiv (\alpha * P || \overline{\alpha} * Q) \backslash \alpha$.

Definition *(Derivative)*: Whenever $P \xrightarrow{\alpha,\mathcal{E}} P'$, we call the pair (α, P') an *immediate derivative* of P; α is referred to as an *action* of P, and we call P' an α-derivative of P. Further, whenever $P \xrightarrow{\alpha_1,\mathcal{E}} \cdots \xrightarrow{\alpha_n,\mathcal{E}} P'$, we refer $(\alpha_1 \cdots \alpha_2, P')$ a derivative of P.

Definition *(Sorts)*: Let $L \subseteq \mathcal{A}$ be a set of actions of process P where $\mathcal{A}$ is the set of actions and all its derivatives lie in $L \cup \{\tau\}$, then we say that P has sort L or L is the sort of P. In the following $Sort(P)$ denotes the sort of process P.

Here we describe the *sequencing* operator, written *seq*, which involves an event and a process. An expression of the form $a : P$, a being in the set of actions, could be interpreted as "perform action a before behaving like process P". The expression P *seq* Q denotes that Q is to be performed after P terminates. Thus, the usual sequencing operator has to account for termination. This can be achieved as in Milner's abstraction, by introducing the convention that termination is signalled through a communication. Thus, in this context, once Q gets a *communication*, a signal, that P has finished, it can start. This convention is formalized below:

Let us introduce the convention that processes may indicate their termination by a distinguished symbol $\overline{\texttt{done}}$ and the processes follow the conventions given below:

> All process expressions using *sequencing* can be translated to the version using prefixing and the convention of indicating termination through the distinct signal $\overline{\texttt{done}}$. This in turn calls for the following definitions of [Mil 89]: P is terminating if, for any *derivative* P' of P, we have
>
> 1. $P' \xrightarrow{\texttt{done}}$ is impossible
> 2. If $P' \xrightarrow{\overline{\texttt{done}}}$, then $P' \equiv \overline{\texttt{done}} : \texttt{nothing}$.

In other words, $\texttt{done}$ is a fictitious symbol and no process emits a $\texttt{done}$. However, each process on termination emits the signal $\overline{\texttt{done}}$. Now the sequencing operator is defined by: $P\ seq\ Q \triangleq (P[\overline{b}/\overline{\texttt{done}}] \mid\mid b : Q)\backslash b$ where b is a new name of action (in order to make sure that $b \notin sort(P)$) and the notation $P[x/y]$ denotes the substitution of x for y in P. In the above equation, $\overline{\texttt{done}}$ gets renamed to $\overline{b}$ and thus, matches with the prefix b of the second component. The renaming allows us to hide "completion or termination" of local components. Thus, sequencing can be achieved through the parallel composition ($\mid\mid$), restriction and the above mentioned termination convention. Let us define the useful *repeat* operator based on sequencing by: $repeat(P) \triangleq P\ seq\ repeat(P)$.

2.2 Interrupt Operator

In this section, we discuss the basic preemptive operator *interrupt* of [Hoa 85] and show how it provides the basis of the definitions of the preemptive operators. The *interrupt* operator is written $\bigtriangledown$. $P \bigtriangledown Q$ (reads *P interrrupted by Q*) behaves like P until Q does anything at all, and thereafter behaves like Q. The definition of $\bigtriangledown$ is given in Equation (5), from which we have that: $P \bigtriangledown \texttt{nothing} \equiv P$ and $\texttt{nothing} \bigtriangledown P \equiv \texttt{nothing}$.

$$\frac{P \xrightarrow{\alpha, \mathcal{E}} P'}{P \bigtriangledown Q \xrightarrow{\alpha, \mathcal{E}} P' \bigtriangledown Q} \qquad \frac{Q \xrightarrow{\beta, \mathcal{E}} Q'}{P \bigtriangledown Q \xrightarrow{\beta, \mathcal{E}} Q'} \tag{5}$$

Note that, the conditions not being exclusive, the transitions are nondeterministic. Hence, it could be interpreted as: *P may stop as soon as Q starts.* In other words, there is no guarantee that P would always stop as soon as Q starts. Now let us enforce determinism in the above rules with a side condition such that P would stop making transitions as soon as Q starts.

Definition: Let $beg(Q)$ denote the set of events that can *trigger* process Q. That is, for Q to start it is necessary to have at least one signal from $beg(Q)$, e.g. for any process Q and any action r, $beg(r : Q) = \{r\}$.

Without loss of generality, we assume that $sort(P) \cap beg(Q) = \emptyset$. The deterministic version of $\bigtriangledown$ is given in Equation (6). Note that condition $\mathcal{E} \cap beg(Q) \neq \emptyset$ of rule (6) is redundant since condition $Q \xrightarrow{\beta, \mathcal{E}} Q'$ ensures it.

$$\frac{P \xrightarrow{\alpha, \mathcal{E}} P' \quad \mathcal{E} \cap beg(Q) = \emptyset}{P \bigtriangledown Q \xrightarrow{\alpha, \mathcal{E}} P' \bigtriangledown Q} \qquad \frac{Q \xrightarrow{\beta, \mathcal{E}} Q' \quad \mathcal{E} \cap beg(Q) \neq \emptyset}{P \bigtriangledown Q \xrightarrow{\beta} Q'} \tag{6}$$

We can derive another variant of interruption, so-called *reset* operator. In the variants of this class, usually called "reset" or "restart", a process is reset to its initial state whenever a given reset-signal, r, is present. Such an operator is obtained by the process expression: $reset(P, r) \triangleq P \bigtriangledown (r : reset(P, r))$ That is, $reset(P)$ behaves like P, except that on r it starts all over again.

3 Preemption in Esterel

3.1 Strong preemption

The basic preemption construct of Esterel is: $\boxed{\texttt{do } P \texttt{ watching immediate } r}$
The interpretation is: The statement P is executed normally up to proper termination or up to *the current or future occurrence* of signal r, which is referred to as the guard. If P terminates strictly before r occurs, so does the whole watching-statement; in such a case the guard has no action. Otherwise, the occurrence of r provokes immediate preemption of the body P and the whole watching statement terminates immediately.

The basic behaviour of: `do` P `watching immediate` r is described by the process expression: $P \ \bigtriangledown \ r$. One of the important features of Esterel is that transitions take place deterministically. That is, whenever there is more than one possibility, the semantics chooses only one of them in a predictable fashion. For instance consider the following statement:

```
do   do P watching immediate r1   watching immediate r2
```

The expression $(P \ \bigtriangledown \ r1) \ \bigtriangledown \ r2$ corresponds to the above fragment of Esterel. We can get the transitions for these operators by taking into account the guard expressions of Esterel into the rules (6).

Now, we could consider the general watching-construct of Esterel which takes the form: `do` P `watching immediate` r `timeout` Q `end`, which has the same interpretation as given earlier except that the occurrence of r provokes Q and the watching-statement terminates with the termination of Q; the other case of normal termination remains the same as earlier. It is expressible as: $P \bigtriangledown (R \times Q)$ where $R \triangleq r : repeat(\tau)$. Note that like in Esterel, an occurrence of signal r can be produced by the environment and not only by Q. In order to ensure termination, instead of blocking, the expression above can be improved as: $P \bigtriangledown [\ ((R \bigtriangledown b) \times Q[b/done]) \backslash b\ ]$ where b is a new name and $R \triangleq r : repeat(\tau)$.

It may be observed that the above rules also ensure priority for the outermost guard. For example in the following statement, when both $r1$ and $r2$ are available, only transition by $r2$ is possible:

```
do   do P watching immediate r1 timeout Q1 end
watching immediate r2 timeout Q2 end
```

3.2 Weak Preemption

In Esterel the `trap`-statement provides the weak-preemption operator which permits *last wills* to be performed before preemption. In fact, the trap-statement provides the exception handling feature in a concurrent environment for ESTEREL. For example, in the statement $\boxed{\texttt{trap}\ T\ \texttt{in}\ P\ \texttt{end}}$, the body P is run normally until it executes an "**exit** T" statement. On the execution of the "**exit** T" (which will be raised at some point of control in P), P is preempted and the whole trap statement terminates with the components concurrent with "**exit** T" in P performing for the last time – often referred to as the *last wills* of P. In fact, T corresponds to an exception and it can be handled through the handlers specified in the appropriate scope.

It can be easily seen that the trap-construct is state-based and is based on the control-flow. For studying/adapting such a construct in data-flow languages such as SIGNAL we will first adapt the construct to the signal-environment. Consider a new operator `wtrap` described below. The process P `wtrap` Q is similar to $P \bigtriangledown Q$ except that when Q is enabled, P can make its last transition. It follows from (7) that P `wtrap nothing` $\equiv$ P and `nothing wtrap` P $\equiv$ `nothing`.

$$\frac{P \xrightarrow{\alpha,\mathcal{E}} P' \quad \mathcal{E} \cap beg(Q) = \emptyset}{P\texttt{wtrap}\ Q \xrightarrow{\alpha,\mathcal{E}} P'\texttt{wtrap}\ Q} \qquad \frac{P \xrightarrow{\alpha,\mathcal{E}} P' \quad Q \xrightarrow{\beta,\mathcal{E}} Q' \quad \mathcal{E} \cap beg(Q) \neq \emptyset}{P\texttt{wtrap}\ Q \xrightarrow{\alpha\cdot\beta,\mathcal{E}} Q'} \tag{7}$$

We can express the behaviour of the operator `wtrap` in terms of the interrupt and the other compositional operators. The definition is given as follows:

$$P\ \texttt{wtrap}\ Q \stackrel{\triangle}{=} [\ \bar{a} * P\ \ ||\ \ [\ R_P\ ||\ (R_Q \times Q)\ ]\ \backslash b\ \ ]\ \backslash a$$

where a and b are new names, $R_P \stackrel{\triangle}{=} ((repeat(a) \bigtriangledown b)$ and $R_Q \stackrel{\triangle}{=} (a \cdot \bar{b}) : repeat(\tau)$. The above expression can be interpreted as follows: process R_P makes process P evolve as long as it performs a's, by use of the ticking operation. When process Q starts, it enforces R_Q to start as well, which in turn forces R_P to perform its b, therefore interrupting production of the a's. As the starting of R_Q contains an a, process P can make a last move at the same time as the first move of Q, as we expect for the `wtrap` operator.

It can be easily seen that the basic trap construct `trap` T `in` P `end` can be achieved by the expression P `wtrap` T assuming "T" to be arriving from the signal-environment. Thus, P continues till T is seen in the environment (it could come from P as well) or it terminates and becomes `nothing`; once "T" is observed, "P" terminates after completing the reaction at that instant. Now, we can define the equations of inference for P `wtrap` Q as given in Equations (7).

We can generalize the trap-construct with the handlers by essentially using the *seq* operator with the second operand of `wtrap`. For example, the statement `trap` T `in` P `handler` H `end` can be abstracted by (P `wtrap` (T *seq* H)).

In the construct `trap` T_1 `in trap` T_2 `in` P `end end` the outermost trap T_1 has priority; in a similar way in the expression `((P wtrap` T_2`) wtrap` T_1`)` the outermost "trap" T_1 has the priority.

3.3 Suspension Operator

Consider the `suspend` operator of ESTEREL. This is a recent addition to ESTEREL. The operator is similar[5] to the `when` operator of LUSTRE or the `when` operator of SIGNAL. Note that even though the operator `when` have essentially the same meaning in languages SIGNAL and LUSTRE, the underlying semantics could differ due to the underlying definitions of the clocks. Operationally, a `when` r is interpreted as: a can make a transition only in the presence of signal r.

The dual of operator `when` is the `suspend` defined in ESTEREL. Operationally, $\boxed{P\ \texttt{suspend}\ r}$ is interpreted as: P can make a transition only when the signal r is not there in the environment. The important difference between the `when` and the `suspend` is that the latter calls for the notion of *absence of signals*. The rules for `suspend` are given in Equations (8,9).

$$\frac{P \xrightarrow{\alpha, \mathcal{E}} P' \quad \overline{\texttt{done}} \notin \alpha \quad r \notin \mathcal{E}}{P\texttt{suspend}\ r \xrightarrow{\alpha, \mathcal{E}} P'\texttt{suspend}\ r} \tag{8}$$

$$\text{(a)}\ \frac{P \xrightarrow{\alpha, \mathcal{E}} P' \quad \overline{\texttt{done}} \in \alpha \quad r \notin \mathcal{E}}{P\texttt{suspend}\ r \xrightarrow{\alpha, \mathcal{E}} \texttt{nothing}} \qquad \text{(b)}\ \frac{r \in \alpha}{P\texttt{suspend}\ r \xrightarrow{\tau, \mathcal{E}} P\texttt{suspend}\ r} \tag{9}$$

The above rules bring out some subtle points:

1. `nothing suspend` r is not the same as `nothing`[6].
2. The side condition of rule (8) makes sure that P can make a transition only when r is not there in the input environment
3. The side conditions in rules (8) and (9a) ensure that whenever r is there, P does not make a transition.
4. As rules (9a) and (9b) are making use of the presence of signals, it is easy to see that absence of signal (like in $r \notin \mathcal{E}$) is ensured relatively to the presence of some signal (referred to as `tick` in ESTEREL).

Note: In the rules for the watching-statement (6): we have stipulated side conditions that ensure a priority for the outermost guard. However, it is not necessary to do the same thing for the suspend-statement, as this statement stops from doing transitions. For instance, (P `suspend` r_1) `suspend` r_2 will be suspended on either r_1 or r_2.

---

[5] Note that, in an imperative language, one has to account for termination of a statement unlike the case of data flow languages.

[6] Note that `nothing` is relevant in the context of imperative terminating programs.

3.4 Immediate and Delayed Operators

In the previous sections, we have discussed various preemption operators in the context of a synchronous algebraic calculus. As argued in [Ber 93], for an effective use of the *preemption feature*, it is necessary that it be used freely without any restriction. Further, for an effective use of preemption, it is necessary to work in the context of a time-dependent model. The model that allows *multi-form notions of time* without resorting to a physical clock is the *perfect synchrony or the zero delay paradigm* [Ber 93] where time is defined externally to programs by the flow of inputs, and the internal bookkeeping in the program is done in zero-delay with respect to all external time units. It is only those instructions that are explicitly written to keep track of the flow of events that can take time. Thus, at a global level, we can refer to actions that take place at the *nth instant* where *instant* corresponds to the reaction to the inputs at that point of time.

The introduction of the notion of instants enables us to specify:

– reactions at the current instant (or immediate operators).
– reactions at the future instants (or delayed operators).

For this purpose, the definitions would require the complete signal environment to check for the presence or absence of signals. In fact, such a structure has been used already while defining interrupt-operator and watching-constructs. With this modification, let us see the effect of using the *delayed* or *immediate* operators on the following operators which appear in one form or the other in the synchronous languages:

1. Strong Preemption: Here, both delayed and immediate preemptions are possible:
   (a) *Immediate strong preemption* corresponds to abortion of the program at the earliest instant, including the current instant, at which the *guard*[7] is present.
   (b) *Delayed strong preemption* corresponds to abortion of the program at the earliest instant, not including the current instant, at which the *guard* is present.
2. Weak Preemption: Here, we can arrive at *immediate* and *delayed* variants assuming that the preemption takes effect through the signal environment. However, it is control-point based as in the case of *traps* of ESTEREL, then we can have only the *immediate* operator.
3. Suspension: Here, again the *immediate* and *delayed* operators are possible with the interpretation given above.

   Berry [Ber 93] has shown how many of the delayed and immediate operators can be derived starting from weak abort and suspension.

[7] That is, the signal on which the statement is to be preempted.

4 Preemption in SIGNAL: SIGNAL*GTi*

The synchronized data-flow language SIGNAL is declarative, and more specifically equational [BLGJ 91]. Specifications are made in the form of a system of equations on the values and synchronizations of involved signals. Special signals are called `event`, and always have the boolean value `true`: they are characterized only by their presence. The behaviour amounts to solving the same system of equation for every instant of the execution. Delays on signals introduce the notion of state, in the form of the memorization of the previous value. In this sense a SIGNAL process P has an internal state (the valuation of its delayed signals) and makes transitions between these states upon the occurrence of signals. Schematically, the system of equations defines the new values of the output signals and state variables, depending upon the input signals and current value of the state variables. The kernel of SIGNAL features primitive processes on signals: functions on the values of synchronous signals (e.g. boolean functions), delays on signals: e`$ init` v_0 (giving the previous of e value, initialized at v_0), selection of occurrences of a signal e according to the value of boolean condition b: e `when` b (intersection of the presences of signal e and signal b at true), merge of two signals: e_1 `default` e_2 (union of the presences of e_1 and e_2; it is made deterministic by the priority given to e_1 in case of simultaneous presence). The synchronous composition of processes noted "`|`" constrains signals featured in the composed processes to agree regarding their presence and value (see [BLGJ 91] for a complete presentation).

Regarding the previous sections, comments may be done when we consider data-flow programs. First of all, it is worthwhile noting that termination issues make no sense in SIGNAL since programs do not terminate. However, one could introduce a kind of "termination" for data-flow programs by emitting a special signal as we artificially made it with the $\overline{\texttt{done}}$ signal in Section 2.1. Second in the case of the SIGNAL language, the *ticking* operator could be treated as the clock of a process P or could be treated as the synchronizer for P from an external observation. Finally, for the preemptive operators, $\texttt{nothing} \bigtriangledown P = \texttt{nothing}$ does not play any role, but both $\bigtriangledown$ and `reset` operators preserve non-termination.

Preemption in SIGNAL has been defined in a language extension called SIGNAL*GTi* [RLG 94]. It consists in associating processes with a time interval upon which they are active, and outside of which they are suspended, or preempted. Time intervals can be opened and closed, and then opened again; hence upon re-entering the time interval a process is re-started, either in its current state (suspension) or completely anew, from its initial state (preemption).

4.1 Time intervals

A time interval is a kind of signal with two possible values, or states: `inside` and `outside`. It is initially in a state `I0` given by its declaration.

A time interval `I` moves from state `outside` to state `inside` upon the occurrence of an event b and from state `inside` to state `outside` upon the occurrence of an event e. Such a time interval will be constructed by: $\boxed{\texttt{I :=]}b\texttt{, }e\texttt{] init I0}}$

Across transitions and rewritings, the current state of an interval will be encoded in its concrete form as follows: an interval `]`b`,`e`]` in state `inside` (resp. `outside`) is written `|`b`,`e`]` (resp. `]`b`,`e`|`) The `|` interval bound means that this bound was the last one met. The transitions of `I` are described by the following rules:

$$\frac{b \in \mathcal{E}}{\texttt{I :=]}b\texttt{,}e\texttt{| init I0} \xrightarrow{I(outside),\mathcal{E}} \texttt{I := |}b\texttt{,}e\texttt{] init I0}} \tag{10}$$

$$\frac{e \in \mathcal{E}}{\texttt{I := |}b\texttt{,}e\texttt{] init I0} \xrightarrow{I(inside),\mathcal{E}} \texttt{I :=]}b\texttt{,}e\texttt{| init I0}} \tag{11}$$

$$\frac{b \notin \mathcal{E}}{\texttt{I :=]}b\texttt{,}e\texttt{| init I0} \xrightarrow{I(outside),\mathcal{E}} \texttt{I :=]}b\texttt{,}e\texttt{| init I0}} \tag{12}$$

$$\frac{e \notin \mathcal{E}}{\texttt{I := |}b\texttt{,}e\texttt{] init I0} \xrightarrow{I(inside),\mathcal{E}} \texttt{I := |}b\texttt{,}e\texttt{] init I0}} \tag{13}$$

As shown by the above rules, time intervals are *left-open/right-closed*: the new state is enforced only *after* the instant where a bound occurs.

4.2 Suspendable tasks

A *task* is a process built up from a process P and an activity time interval `I`. Suspension of a task consists in conditioning its activity P upon the "presence" of interval `I`, while preserving its internal state when suspended otherwise. This suspendable task is written: $\boxed{P \ \texttt{on} \ I}$. The process is active inside the interval: it behaves as P. Outside the interval, it is inhibited: its clock is cut. When re-entering the interval, it re-starts from its current state.

$$\frac{P \xrightarrow{\alpha,\mathcal{E}} P' \quad \texttt{I} \xrightarrow{I(inside),\mathcal{E}} \texttt{I}'}{P \ \texttt{on I} \xrightarrow{\alpha \cdot I(inside),\mathcal{E}} P' \ \texttt{on I'}} \tag{14}$$

$$\frac{\texttt{I} \xrightarrow{I(outside),\mathcal{E}} \texttt{I}'}{P \ \texttt{on I} \xrightarrow{I(outside),\mathcal{E}} P \ \texttt{on I'}} \tag{15}$$

In terms of the algebraic operators, suspension on intervals can be derived using the *repeat* and `wtrap` operators:

$$P \ \texttt{on} \ ]\mathtt{b},\mathtt{e}] \ \stackrel{\triangle}{=} (\ (a * P) \ || \ repeat(b \ : \ (repeat(\bar{a}) \ \texttt{wtrap} \ e)) \ )\backslash a$$

where the local signal a has a new name and is used to constrain the activity of P (using $*$) from after b to the occurrence of e (inclusively, hence the use of `wtrap`). Indeed, the behaviour of an interval is given by the expression $repeat(b \ : \ (repeat(\bar{a}) \ \texttt{wtrap} \ e))$. During suspension (between e and b), the current state of the process is kept in the $a * P$ sub-expression.

4.3 Preemptible tasks

In preemption, noted $\boxed{P \texttt{ each } I}$ the preempted process is re-started (if ever) from its starting point, i.e. form the initial state of its state variables in the case of data-flow processes. The difference between suspension and preemption concerns the opening of the interval; rules are similar to those for `on`-tasks, but have to be more detailed on the opening of the interval.

$$\frac{P \xrightarrow{\alpha, \mathcal{E}} P' \quad \texttt{I} \xrightarrow{I(inside), \mathcal{E}} \texttt{I}'}{P \texttt{ each I} \xrightarrow{\alpha \cdot I(inside), \mathcal{E}} P' \texttt{ each I'}} \tag{16}$$

$$\frac{\texttt{I} \xrightarrow{I(outside), \mathcal{E}} \texttt{I}' \quad b \notin \mathcal{E}}{P \texttt{ each I} \xrightarrow{I(outside), \mathcal{E}} P \texttt{ each I'}} \tag{17}$$

When opening the interval, the process of the interruptible task must be re-started from its initial state. Therefore it is reset:

$$\frac{b \in \mathcal{E}}{P \texttt{ each]}b\texttt{,}e\texttt{|} \xrightarrow{b \cdot \alpha, \mathcal{E}} reinit(P) \texttt{ each |}b\texttt{,}e\texttt{]}} \tag{18}$$

The function $reinit(P)$ is defined structurally so as to propagate the reset down to the state variables, i.e. in our case to the delays on signals; their stored value in the delay is reset to their initial value without causing an occurrence to be present (we do not detail it here due to space limitation, see [PRS 95]).

In terms of the algebra operators, interruption on intervals is described by:

$$P \texttt{ each]}b\texttt{,}e\texttt{]} \triangleq b : (P \texttt{ wtrap } e) \;\; seq \;\; (P \texttt{ each]}b\texttt{,}e\texttt{]})$$

where after an occurrence of b, P is executed until e (inclusively) occurs, then stops, and later might go in sequence into itself recursively.

4.4 Strong, weak, immediate and delayed preemption

With regard to the classification of preemption structures in concurrency by G. Berry [Ber 93], we can say briefly that:

- suspension and abortion are handled respectively by `on` and by `each`.
- delayed preemption is the default case; immediate preemption can be obtained by defining an event subtraction: $e_1 - e_2 \triangleq (\texttt{when not } e_2 \texttt{ default } e_1)$. Then, immediate versions `i-on` and `i-each` of the task constructors `on` and `each` respectively can be defined by: $P \texttt{ i-on]}b\texttt{,}e\texttt{]} \triangleq P \texttt{ on]}(b - e)\texttt{,}e\texttt{]}$ and $P \texttt{ i-each]}b\texttt{,}e\texttt{]} \triangleq P \texttt{ each]}(b - e)\texttt{,}e\texttt{]}$.

– weak preemption is the default case here; strong preemption would involve defining another kind of intervals, instead of the *right-closed* interval: a *right-open* interval $]b, e[$. In that case the event e causing the task to stop must be forbidden to originate in the process, in order to avoid a problem of causality (the process causing its own inhibition within an instant).

5 Expressiveness and Completeness

In this section, we show how preemption and suspension operators can be *derived* from each other along with the other compositional operators. That is, given a program using one class of operators, we show how we can obtain an equivalent program using other operators.

First, let us see whether the `suspend` operator can be realized in terms of the `interrupt` and other compositional operators. We can see that `suspend` can indeed be realized by the parallel composition ($|$) and the interrupt operator ($\bigtriangledown$) defined earlier by: $(\ P[b/\mathtt{done}]\backslash r)\ \ ||\ \ (repeat(\overline{r}) \bigtriangledown \overline{b}\ \ )\backslash b\ \ =\ \ P$ `suspend` r where termination of P is carried by b, and causes termination of the *repeat* statement; but this happens only in the absence of r. Note that $P\backslash r$ cannot react for r and thus, the requirement that P`suspend` r does not react when r is present is satisfied.

We can express "`interrupt`" in terms of "`suspend`" as follows assuming $sort(P)\ =\ \{x_1, \cdots, x_n\}$: $P \bigtriangledown r\ \equiv\ (P$ `suspend` $r) \times Q$ where Q is given by $Q\ = (x_1 + x_2 + \cdots + x_n)Q + r$. The equivalence follows from the axioms and the equations for synchronous product. It may be noted that the derivation of "interrupt" from "suspend" requires the choice operator "+" which can be deribed in Meije.

This also explains why the various synchronous languages appear to be equally expressive. Of course, some abstractions are expressible more naturally in one paradigm than in others. Further, we get some differences with reference to ease of writing and expressing in Esterel and Signal due to the underlying notion of data flow and the fact that programs don't terminate.

Remarks: It must be noted that we have exploited the signalling of completion through a special signal $\overline{\mathtt{done}}$. This is not really new and need not have to be construed as enforcing another communication; in fact, it is not uncommon to define transition systems using labels reflecting the termination of constructs (see for example, transition systems of Esterel). In the above, we have shown how the various operators can mutually be simulated. It may be observed that we need to know the structure of the program for the translation. In fact, this must be evident from the derivation of "interrupt" from "suspend" and other operators. Now, the question is: Is it possible to compare the power of `suspend` and `preemption` operators at a semantic level. That is, we will be comparing the expressive powers of the two at only a behavioural level. In fact, such a study has been done in [PRS 94]. We state below the main theorems from [PRS 94].

The study in [PRS 94] is based on the notion of Esterel contexts. Such contexts represent "derived constructs" of the language. We shall assume a countably infinite supply of "statement" variables, and allow ESTEREL programs to be built up using variables also in place of program statements. A "program" with free variables will be called an ESTEREL *context*. Such contexts represent composite operators constructible in ESTEREL. We shall use C, D, E to denote contexts, and X, Y, Z to denote statement variables. In particular, a context C with X, Y, Z as its only free variables will be denoted by $C(X, Y, Z)$. When X, Y and Z are instantiated with concrete ESTEREL statements, say p, q and r, the resulting ESTEREL program is denoted by $C(p, q, r)$. We shall sometimes parameterize a context with, say, a signal variable S (or a value variable V), writing it as C_S (respectively, C_V). Such a parameterized context represents the class of contexts obtained by instantiating the parameter(s) to a value of the appropriate type. For instance, C_s (respectively, C_5) may represent a context in which S (respectively, V) has been instantiated to the signal s (respectively, the value 5).

Consider the subset of ESTEREL statements obtained using all the preemption operators but without using the suspension operators immediate suspend and delayed suspend. Then we can prove the following theorems:

Theorem 1: The behaviour of ESTEREL programs with preemption operators only is properly contained in the behaviour of ESTEREL programs with preemption and suspension operators.

Theorem 2: The behaviour of ESTEREL programs with suspension operators only is properly contained in the behaviour of ESTEREL programs with preemption and suspension operators.

Further work in the direction of the completeness notions and axiomatization ESTEREL are in progress.

6 Conclusion and Perspectives

We proposed a description of the preemption primitives of the reactive languages ESTEREL and SIGNAL in the framework of operators of a process calculus derived from MEIJE and SCCS. Using operators like synchronous product, sequencing and interrupt, we could describe the behavior of strong and weak preemption, and suspension in ESTEREL, and suspension and preemption of processes associated with time intervals in SIGNAL*GTi*, a tasking extension of SIGNAL. We have also shown that suspension and preemption could be shown to be derivable from each other. In fact, the result may not be very surprising since MEIJE is complete. However, the study brings out clearly the structure of the various preemptive operators in a single setting and provides comparisons from the point of view of expressive completeness and behavioral completeness. We also examined completeness with reference to the behavioral notion and showed that suspension and preemption are incompatible under such a behavioral notion. It would be nice to show how under such notions one can get a completeness result *à la* MEIJE. That is, to arrive at a subset of operators of suspension and preemption that are complete (under some usual conditions) such that whatever can be derived using the rules of action and reduction can also be directly defined using the chosen subset of operators.

We hope to study the features of the language 2Z [Vul 94] in this framework as it has interesting features for resetting the flow. For getting a complete picture, it would be interesting to include other languages such as LUSTRE, STATECHARTS, ELECTRE, ARGOS, ATP, ... [Hal 93] in this framework – perhaps, in the context of the clock/causal analysis of the underlying languages. Furthermore, it would be interesting and challenging to extend the study to unifying frameworks of synchrony and asynchrony as done in CRP [BRS 93]. There is a similarity between the tasks in ESTEREL, which are at the base of CRP, and those in SIGNAL *GTi*: they are a way of making reference to actions lasting on a non-instantaneous duration. A way of treating external or global nondeterminism is given in [SR 94] where a translation from CSP to ESTEREL is presented, as well as results about the inclusion of CSP behaviors in ESTEREL's. Also, another interesting perspective lies in the area of hybrid systems [GNRR 93], as shown in an earlier paper [Shy 93].

References

[BLGJ 91] A. Benveniste, P. Le Guernic, C. Jacquemot. *Synchronous programming with events and relations: the* SIGNAL *language and its semantics.* Science of Computer Programming, 16, pp. 103–149, 1991.

[BRS 93] G. Berry, S. Ramesh, and R.K. Shyamasundar, *Communicating Reactive Processes*, 20th ACM POPL, South Carolina, Jan 1993, pp. 85-99.

[Ber 93] G. Berry, *Preemption in Concurrent Systems*, Proc. FSTTCS 93, LNCS, 761, pp.72-93, Springer-Verlag, Berlin.

[GNRR 93] R.L. Grossman et al., *Hybrid Systems*, LNCS, 736, Springer-Verlag, 1993.

[Hal 93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[Hoa 85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall Int., 1985.

[Mil 89] R. Milner, *Communication and Concurrency*, Prentice-Hall Int., 1989.

[PRS 94] P.K. Pandya, Y.S. Ramakrishna, and R.K. Shyamasundar, *A Compositional Semantics of Esterel in Durational Calculus*, 2nd AMAST Workshop on Real-Time Systems: Models and Proofs, Bordeaux, France, June 1995.

[PRS 95] S. Pinchinat, E. Rutten, R.K. Shyamasundar. *Preemption Primitives in Reactive Languages.* INRIA Research Report, 1995. *(in preparation)*

[RLG 94] E. Rutten, P. Le Guernic. Sequencing data flow tasks in SIGNAL. Proc. *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, June 21, 1994.

[Shy 93] R.K. Shyamasundar, *Specification of Hybrid Systems in CRP*, Proc. of AMAST 93, Workshops in Computing Series from Springer-Verlag, Edited by M. Nivat, C. Rattray, T. Rus and G. Scollo, pp. 227-238, December 1993.

[SR 94] R.K. Shyamasundar, S. Ramesh, *Languages for Reactive Specifications: Synchrony vs. Asynchrony*, Proc. FT-RT-FTS 94, LNCS, 863, pp. 621–640, Springer-Verlag, Berlin.

[Sim 85] R. de Simone, *Higher-Level Synchronizing Devices in MEIJE-SCCS*, Theoretical Computer Science, 37, pp. 245-267, 1985.

[Vul 94] J.E. Vuillemin, *On Circuits and Numbers*, IEEE Transactions on Computers, 43, August 1994, pp. 868-879.