Representation and Analysis of Reactive Behaviors: A Synchronous Approach

Charles ANDRÉ
Laboratoire Informatique, Signaux, Systèmes (I3S)
Université de Nice-Sophia Antipolis / CNRS
41, bd Napoléon III — F – 06041 NICE Cedex
e-mail: andre@unice.fr

ABSTRACT

Reactive systems involve communication, concurrency and preemption. Few models support these three concepts, even less can correctly deal with their coexistence. The synchronous paradigm allows a rigourous approach to this problem, crucial to reactive systems.

This paper analyzes the underlying hypotheses of the synchronous approach. Reactive behaviors are characterized. A new visual model (SYNCCHARTS) is then proposed. This graphical model is fully compatible with the imperative synchronous language ESTEREL and is specially convenient to express complex reactive behaviors.

1 INTRODUCTION

A *reactive system* maintains permanent interactions with its environment. Usually, reactive systems are concurrent systems. Their global behavior results from the cooperation of their components (subsystems). In order to carry out an expected behavior, the evolutions of subsystems must be coordinated. Communication (information exchange) plays a central role in this coordination, and consequently, reactive systems are often viewed as communicating processes. This approach relegates *preemption* to a position of secondary importance, which is prejudicial to many reactive applications. Real-time operating systems, interrupt-driven systems, and more generally, control-oriented systems heavily rely on preemption. The specification of such systems needs preemption as a first class concept; their programming requires preemption primitives. Few models can deal with preemption. Languages that support preemption, to some extent, do not offer primitives tailored to reactive applications. A reason for this lack, is that most semantics dealing with both concurrency and preemption are complex or vague.

Since often involved in safety critical applications, reactive systems should be *predictable*. The most critical parts must be *reactive* (i.e., they can always react to stimuli) and *deterministic* (the reaction must be unique). To ensure reactivity and determinism in the presence of communication, concurrency and preemption is the challenging problem addressed by the *synchronous approach* [1, 2]. The "*perfect synchrony hypothesis*" assumes on the one hand that cause (stimuli) and effect (reaction) are simultaneous, and on the other hand that information is instantaneously broadcast. These assumptions lead to an abstract view of temporal behaviors, in which communication, concurrency and preemption can be considered as orthogonal concepts. Within the framework of synchronous programming, clear mathematical semantics can be given to several forms of preemption. This point of view is advocated in a G. Berry's paper entitled "Preemption in Concurrent Systems" [3].

Engineers in the field of control and manufacturing systems are somewhat reluctant to synchronous languages. As a rule, they prefer *graphical approaches*. The GRAFCET [4] (Sequential Function Charts) is widely used for industrial logic control. STATECHARTS [5], a hierarchical visual model which is part of the STATEMATE environment, allows design of complex reactive systems and it takes advantages from its industrial support. STATECHARTS are convenient but, as stated in a review of the various extensions of STATECHARTS [6], their semantics has to be clarified. ARGOS [7] is another synchronous and graphical model. It is based on a clear semantics, but it has not been widely distributed.

In a previous paper [8] we have presented some potential applications of the synchronous languages to industrial process control. The aims of the present contribution are twofold:

- First, we shall have a closer look at the basis of the synchronous approach,
- Second, we shall introduce SYNCCHARTS (an acronym for Synchronous Charts) which inherit from STATECHARTS and ARGOS.

Plan

The paper is organized as follows:

- In a first part, we propose an example of reactive system: A Cruising Speed Controller.
- We then study the underlying hypotheses of the *synchronous approach*. We characterize reactive behaviors. The causality problem induced by the synchronous approach is also analyzed.
- The third part is devoted to an informal presentation of SyncCharts.
- The last part deals with the semantics of this new model.

2 A REACTIVE SYSTEM

A *Cruising Speed Controller* is an instance of a control-dominated reactive and real-time system. We propose a simplified version that will be used throughout this paper for illustrative purpose. We adopt "Cruise Controller" as a short form of "Cruising Speed Controller".

2.1 The Cruise Control

Fig.1 is a blackbox view of the controller. The driver, the engine, the car, the road, ...compose the environment in which this controller is embedded.

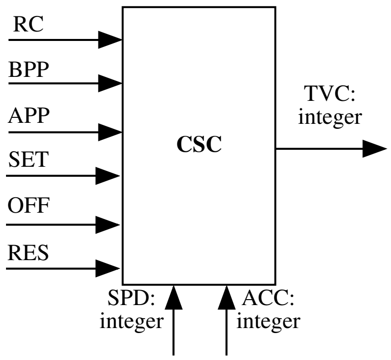

Figure 1: The Cruise Controller: Blackbox

Interface

List of sensors:

`RC` : Regulation Clock
`BPP`: Brake Pedal Pressed
`APP`: Accelerator Pedal Pressed
`SET`: SET button
`OFF`: OFF button
`RES`: RESume button
`SPD`: actual SPeeD
`ACC`: ACCelerator pedal position

Actuator: `TVC`: Throttle Valve Command
`SPD, ACC, TVC` convey integer values.

Expected Behavior

When the Cruise Control is "off", the throttle valve is controlled by the accelerator pedal through a function $A : integer \longrightarrow integer$, where the argument is the accelerator pedal position and the result is the throttle valve command. When the Cruise Control is operational, the throttle is controlled either by the accelerator pedal through A or by the regulator through the function $R : integer \times integer \longrightarrow integer$, where the two arguments are the actual speed and the desired speed.

The Cruise Control is switched on by pressing the `SET` button. The value of the current speed is assigned to the reference speed variable. It is switchted off by pressing the `OFF` button.

When the Cruise Control is operational:

- applying the brakes suspends the speed regulation which can be re-activated by pressing the `RES` button.
- each acceleration (`APP` = 1) suspends the speed regulation which is automatically re-activated when the accelerator pedal is released.
- at any time, the reference speed can be set to the current speed by pressing the `SET` button.

Of course, this informal specification does not constitute a complete specification.

2.2 Reactive System Modeling

Even if the Cruise Controller is ultimately to be implemented on a single processor, it is convenient to describe it as a set of cooperating processes.

From this example, we draw the main needs for reactive system modeling.

Hierarchy

Obviously, there are two functioning modes:

- The Throttle control when the Cruise Control is not operational,
- The Throttle control when the Cruise Control is operational.

The latter can be further decomposed into the "Brake Watching" and the "CSC Active" processes. In turn, "CSC Active" can be refined into sub-processes.

Communication

Communication with the environment is performed with the help of the sensors (incoming information) and the actuators (outgoing information). Communication between processes is also necessary, e.g., the Brake Watching process must inform the CSC Active process that the brake pedal has been pressed.

Concurrency

The two processes Brake Watching and CSC Active perform almost independently: they are concurrent.

Preemption

Coordination is most important for reactive systems. Processes may coordinate with each other by information exchange (communication, synchronization). Preemption is another way to ensure coordination. *"Process preemption ... consists in denying the right to work to a process, either permanently (abortion) or temporarily (suspension)"* (G. Berry [3]).

In our controller, the CSC Active process is suspended by action on the brake. As for abortion, it is applied when switching from a mode to the other.

A model for reactive system must support hierarchy, communication, concurrency and preemption. Moreover, the behavior must be *reactive* and *deterministic*. Combining communication, concurrency and preemption in a deterministic way is challenging. The synchronous approach, studied below, proposes a solution to this issue.

3 THE SYNCHRONOUS PARADIGM

3.1 Hypotheses

The synchronous approach adopts an abstract, indeed even ideal, view of real systems. In this section, we have a closer look at the hypotheses underlying the synchronous approach. Esterel is the oldest synchronous language. Quoting N. Halbwachs [2] it is *"the best language to highlight the specificity of the synchronous approach"*. In what follows, we liberally borrow from Esterel.

Signals

Interactions between a reactive system and its environment take on the most varied forms [9]: sensors or actuators, discrete or continuous information, collection by polling or by interruption, ...

A first simplification is to consider a unique way of exchanging information. A reactive system and its environment maintain permanent interactions by means of *signals*: the system receives *input signals* and emits *output signals*.

A signal conveys two pieces of information: its *presence status* (a signal is either *present* or *absent*) and its *values* of a given type (e.g., `TVC` is an output signal, its value is an integer).

The presence status is transient (pulsed), whereas the value is persistent. The value of a signal may change only when the signal is present. Pure signals and sensors are special cases. A *pure signal* has no value, it is used to signal that some condition has become true (e.g., `APP` is a pure signal whose presence indicates that the accelerator pedal has just been pressed). A *sensor* has no presence status; it is set by the environment, the reactive system can only read the value of a sensor (e.g., `SPD` is a sensor bearing an integer value imposed by physical laws).

Global Perception

Inputs to a reactive system are numerous and may change sporadically. The synchronous approach assumes that all input signals can be perceived simultaneously and that this perception is objective. The same assumption is made for output signals. So, the synchronous model deals with tuples of signals. We call this hypothesis "the *perfect sampling* hypothesis".

The Logical Time

A reactive system is idle most of the time, excepted when prompted by a stimulus to which it must react instantly.

The synchronous approach to reactive system considers that the system is kept aware of "time passing" by the flow of inputs, more precisely, by the pulsed presence status of input signals. So, the synchronous model does not rely on the physical time, it uses a *logical time* instead. This allows to capture the notion of *multiform time*: any input signal may be taken as a time reference, be it linked to a physical clock or to any other physical phenomenon.

The model proceeds in successive reactions, one at each *logical instant*. At each instant, the reaction is computed using the current (tuple of) input signals (and some internal information).

The Zero-Delay Hypothesis

A drastic simplification is braught by the *Zero-Delay hypothesis*: Internal operations are supposed to be done in zero-delay with respect to all external time units. A consequence is that the outputs are synchronous with the inputs that cause them.

Broadcasting

Up to now signals are involved in communication between the reactive system and its environment. *Local signals* can be used as well. They allow communication among subsystems: they are hidden from the external observation but they participate to the behavior of the system.

The synchronous approach assumes that signals are *instantaneously broadcast*. A consequence is that *all* the signals (including the output signals) have to be taken into account in order to determine the output signals to be emitted. This may induce surprising behaviors (see Section 3.3).

Summary of the Synchronous Hypotheses

Synchronous Hypotheses
1-*Signal* (support of communication)
2-*Perfect sampling* (tuples of signals)
3-*Logical time* (instant)
4-*Zero-delay* (instantaneous internal operations)
5-*Instantaneous broadcasting*

These hypotheses, even if not stated in the same terms, are shared with other models. *Abstract Machines* are one of them, especially *combinational* and *sequential* machines which are not restricted to binary variables (e.g., see J. Zahnd's book, entitled "Machines séquentielles" [10]). In Sec.3.3, we shall use Mealy machines to express the behavior of some ESTEREL programs.

3.2 Reactive Behaviors

In Sec.2.2, we saw that communication, concurrency and preemption were three major concepts in reactive systems. According to G. Berry [3], they have to be orthogonal. ESTEREL reflects this choice and so will do SYNCCHARTS (Sec.4).

Only the presence of a signal may trigger a reaction, so, presence and absence of signals play a central role in reactive behaviors. In what follows we restrict our attention to reactive systems with pure signals only. This entails no loss of reactive properties and makes explainations easier. PURE ESTEREL is a "kernel" ESTEREL with pure signals only.

In this part, we recall how the PURE ESTEREL language expresses these concepts; its primitives are given enclosed in [ ]. Preemption which is typical of reactive behaviors deserves a special attention.

Communication

Communication is done by signal broadcasting. Recall that a signal must be either present or absent at an instant. A signal must be declared as an input signal [`input` *Ident*], an output signal [`output` *Ident*], or a local signal with a scope [`signal` *Ident* `in` *stat* `end`].

An output or local signal can be emitted by a reaction [`emit` *Ident*]. Any signal can be tested. [`present` *Ident* `then` $stat_1$ `else` $stat_2$ `end`]. The emission and the test of a signal take no time.

Concurrency and other control structures

Two processes may execute in an orderly way, independently or exclusively. ESTEREL, as an imperative language, provides control structures for the *sequence* [$stat_1$; $stat_2$], the *conditional* [`present` *Ident* `then` $stat_1$ `else` $stat_2$ `end`], and the infinite *iteration* [`loop` *stat* `end`]. The *parallel* execution is also supported [$stat_1$ `||` $stat_2$]. For all structures, the control passing management takes no time, in accordance with the Zero-Delay hypothesis. For instance, for "$stat_1$; $stat_2$", at the very instant when $stat_1$ terminates, $stat_2$ begins its execution. Note that parallel statements start at the same instant, and the parallel terminates when both components terminate.

Preemption

Preemption either kills the process or suspends it temporarily. *Suspension* is forced as long as a given signal is present [`suspend` *stat* `when` *Ident*]. *Abortion* has two forms: a weak and a strong one. The *strong abortion* kills the process as soon as a given signal is present; the killed process is not even allowed to execute its "last wishes" when the abortion occurs [`do` *stat* `watching` *Ident*]. The *weak abortion* differs from the previous one in the fact that the killed process executes its reaction at the current instant, before getting killed. There is no dedicated primitive for the weak abortion in the current version of the ESTEREL language. Instead, one has to use an escape mechanism [`exit` *Ident*] and concurrency within a trap [`trap` ... `in` ... `end`].

A preemption is always triggered by the presence of a signal. The default option is to wait for the first *strict future instant* when the signal becomes present. *Immediate preemption* takes account of the possible presence of the triggering signal at the current instant [`immediate` *Ident*].

There exists a special process that does nothing for ever [`halt`]. Of course, this process takes time! Waiting for the next future presence of a signal [`await` *Ident*] is nothing else that preempting the halt process by this signal [`do halt watching` *Ident*].

3.3 Causality

Reactive Systems and Circuits

Given an ESTEREL program, it is possible to derive a circuit which exhibits the same input/output behavior [11]. And besides, the ESTEREL's compiler version 4 relies on this fact [12]. Other synchronous languages like LUSTRE, have straight translation into hardware implementation (see [13] for Boolean LUSTRE).

Fig.2 represents a circuit associated with the following ESTEREL program:

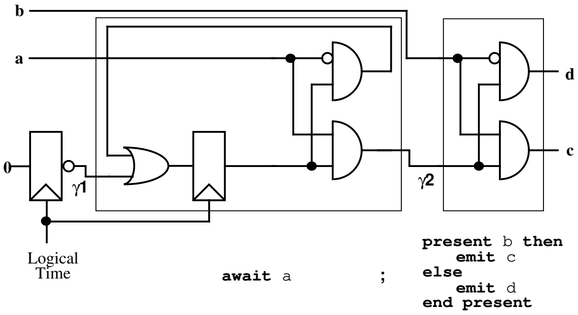

Figure 2: Circuit Representation

```
module example:
input a, b;
output c, d;
     await a;
     present b then emit c else emit d end
end module
```

A '1' (a '0') on a wire is interpreted as the presence (the absence) of the associated signal. Zero-Delay statements are represented by combinational circuits (e.g., the rightmost sub-circuit stands for the `present` ... `end` statement). Waiting for a signal presence involves registers, i.e., sequential circuits. The two registers are initially reset. Note that the succession of instants is imposed by a control line, external to the program.

Statements with concurrency and preemption imply more complex circuits, but they are still made of gates and registers.

Paradoxal Behaviors

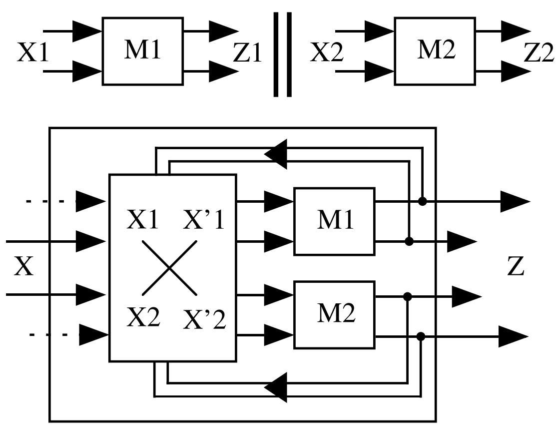

Figure 3: Circuit with loops

Since signals are instantaneously broadcast they may cause "feedback" wires in circuits. Let $M1$ and $M2$ be two circuits associated with two reactive modules $m1$ and $m2$. If we compose $m1$ and $m2$ in parallel and if they have common signals, then we get a circuit with loops (Fig.3). An input connected to an output cannot be driven by the environment any longer. Therefore, the new set of inputs X is a subset of $X_1 \cup X_2$, whereas $Z = Z_1 \cup Z_2$. The leftmost sub-circuit is an interconnection network.

Assume that $M1$ and $M2$ are (deterministic) combinational circuits defined by the Boolean functions F_1 and F_2. Connections are expressed by two Boolean functions C_1 and C_2. The new input to M_1 (M_2) is $X_1{}' = C_1(X, Z_1, Z_2)$ ($X_2{}' = C_2(X, Z_1, Z_2)$, respectively); thus:

$$\begin{aligned} Z_1 &= F_1(X_1{}') = F_1 \circ C_1(X, Z_1, Z_2) \\ Z_2 &= F_2(X_2{}') = F_2 \circ C_2(X, Z_1, Z_2) \end{aligned}$$

i.e, $Z = F(X, Z)$. For an input x, the output z must be a solution of $z = M(x, z)$, i.e., z must be a *fixpoint* of $\lambda y.F(x, y)$. Because F is not always monotonic, there may be zero, one or several minimal solutions. A more formal presentation of this statement, including sequential circuits, can be found in [14]. This is not a new result: It is known that interconnections of combinational circuits may lead to non combinational circuits [10].

We use this result to introduce "causality cycles".

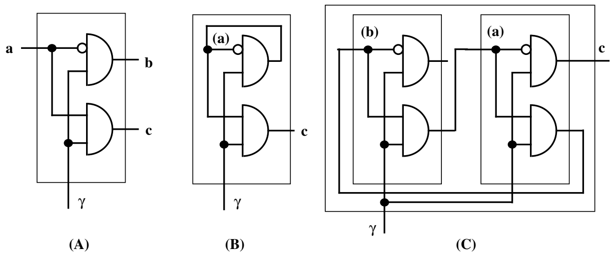

Figure 4: Causality cycles

Circuit Fig.4.A corresponds to the ESTEREL statement: `present` a `then emit` c `else emit` b `end`. γ is a control line. Obviously, $b = \gamma.\overline{a}$ and $c = \gamma.a$

Consider now, the statement:

```
signal a in
     present a then emit c else emit a end
end signal
```

The associated circuit (Fig.4.B) is cyclic. a is hidden from the outside (local signal). We have a system of two equations: $a = \gamma.\overline{a}$ and $c = \gamma.a$. If $\gamma = 0$ then, $a = c = 0$ is a solution. If $\gamma = 1$ then there is no solution since a should be equal to $\overline{a}$. The program is *not reactive*, and by the way, rejected by the ESTEREL's compiler.

Fig.4.C is the translation of the following program:

```
signal a, b in
     present a then emit b else emit c end
     ||
     present b then emit a end
end signal
```

It is easy to see that this circuit is non deterministic. There exist two different solutions when $\gamma = 1$: either $a = b = 0$ and $c = 1$, or $a = b = 1$ and $c = 0$. I.e., the program may emit c or not. This *non deterministic* behavior is also rejected by the compiler.

Non determinism and/or non reactivity due to feedback loops are often referred to as *"causality cycles"* because an effect (an output) may affect its cause (an input). We have found the same problems in another synchronous model: the *Sequential Function Charts* [15]. Causality problems are somewhat disconcerting for programmers especially since some correct programs may be rejected by the compiler (Fortunately, incorrect ones are always rejected).

Some avoid the problem by imposing restrictions: e.g., the synchronous language SL [16] adds a delay before any negative test (i.e., conditionned by the absence of a signal). Others adapt the semantics in order to accept any syntaxically correct programs (see the many semantics of StateCharts [6]). The forthcoming version 5 of the Esterel's compiler is about to bring a natural and practical solution to this problem [17].

4 SYNCCHARTS

SyncCharts[1] are a new graphical model dedicated to Reactive System Modeling. Many features are inherited from StateCharts. A special care is taken in the representation of preemption.

4.1 A Simple Example

Fig.5 is a watchdog system. `set` activates the `Counter` which counts up the occurrences of `T` from 0. If `Counter` reaches 5 then `Alarm` is emitted. At any time, `reset` disables the counting or the alarm.

The example is used to illustrate the definitions of SyncCharts components. References to this example are given enclosed in [ ].

4.2 Elements of Syntax

Fig.6 gathers the graphical elements of SyncCharts. SyncCharts are a state-based description of reactive behaviors. They support states, hierarchy of states, concurrency, transitions of several types and even textual annotations. Roughly, a macrostate is either an Esterel module or a parallel composition of state-graphs. Each state graph is a collection of one or several interconnected states, with initial, and possibly, final states. In turn, a macrostate can be substituted for a state. Instead of this top-down approach to SyncCharts, we adopt a bottom-up presentation of the various components.

[1]A comprehensive presentation of this model (syntax and semantics) is available as a technical report [18].

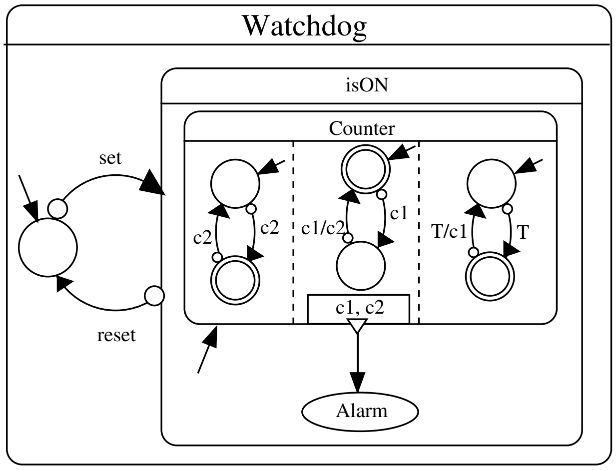

Figure 5: SyncCharts of a Watchdog

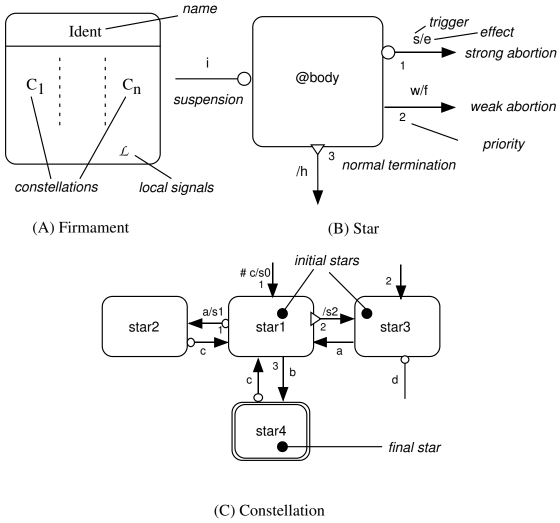

Figure 6: SyncCharts: Elements of Syntax

Star (Fig.6.B)

The graphical basic block is the *state*. In fact, this block is not only the expression of some local invariant behavior (its *body*) but also a full description of the ways to leave this state (preemptions). So, the basic block is both a classical state and its outgoing arcs. We would rather call it a *star* (with outgoing arcs seen as beams). A star is drawn as a rounded rectangle with its "beams". Arcs are numbered according to a priority ordering (the less, the highest priority). The *weak abortion* is expressed by a plain arrow ($\longrightarrow$). The *normal termination*, i.e., leaving the star because its body terminates, is drawn as an arrow with a leading triangle ($\rhd\!\!\rightarrow$) [leaving `Counter` when the count is reached]. The *suspension* is denoted by a special dangling incoming arc with a circle head ($-\!\circ$). The *strong abortion* which is a combination of weak abortion and suspension, is specified by an hybrid arrow ($\circ\!\!\rightarrow$) [leaving `isON` triggered by `reset`]. A star with n outgoing arcs (n beams) is said to be a n-star. There exists a special kind of star with no beam. It is a 0-star. There is no way to leave a 0-star but by a higher level abortion.

Constellation (Fig.6.C)

Stars are interconnected to make a *constellation*. Fig.6.C shows an instance of a constellation with two initial stars and one final star. There may be 0, 1 or several final stars. Final stars are distinguished by double-line rounded rectangles. There must be at least one initial star. Initial stars are identified by a dangling incoming arrow A constellation need not be a connected graph. A constellation can be made of a single 0-star which is implicitly an initial star.

Macrostate (Fig.6.A)

A parallel composition of constellations is a *firmament* or a *macrostate*. The components are delimited by dashed lines. [`Counter` is composed of 3 constellations]. There may be a single constellation in a macrostate. In this case, dashed lines are omitted. A macrostate may have local signals [`c1` and `c2` are local to `Counter`]. A macrostate is drawn as a rounded rectangle with a *header* in which the optional name is written.

Note that a macrostate terminates when each of its components is in a steady final star [For `Counter` when the binary code is 101, i.e., 5].

4.3 Communication

Signals

Three sets of signals (the input set $\mathcal{I}_p$, the output set $\mathcal{O}_p$, the local set $\mathcal{L}_p$) are associated with each star, constellation, macrostate p [$\mathcal{I}_{\texttt{Counter}} = \{\texttt{T}\}, \mathcal{L}_{\texttt{Counter}} = \{\texttt{c1, c2}\}, \mathcal{O}_{\texttt{Counter}} = \emptyset$]. Of course, signals may appear in ESTEREL modules. They are also present in labels (see Arc labeling) and in some states (see Terminal stars).

Let $\mathbb{S}$ be a set of signals. Signals may be combined into *compound signals*, we use disjunction '+', conjunction '·' and negation '—' (Refer to [18] for details).

Terminal stars

In a Moore machine, outputs are associated with states. In our synchronous approach we have to emit these signals at each instant. Moore-like states are special stars whose body is a simple process that emits all the signals in S at each instant, where $S \subseteq \mathbb{S}$. For convenience, we introduce a short hand notation: an oval with the set of the signals to be emitted written inside [`Alarm`].

Arc labeling

An arc is labeled by a pair composed of a compound signal and a subset of signals. The first component is the *trigger*, the second component is the *effect*. They are separated by a "/". One or both components can be omitted. Note that normal termination ($\rhd\!\!\rightarrow$ arcs) has not explicit trigger.

The label can be prefixed by the symbol "#". In this case, we use *immediate* preemption, instead of *delayed* preemption which is the default option.

The signals in "effect" can be output or local signals. Local signals are often used for synchronization [in `Counter`, `c1` is emitted every other occurrence of `T`. `c1` triggers a preemption of the middle constellation in `Counter`].

4.4 Speed Controller

Fig.7 is a SyncChart for the Cruise Controller. It is self explanatory. We add only a few comments. The constellation, in `CSC Active`, in charge of accelerations, has two initial stars because, when activating the automatic control, the accelerator pedal can be either pressed, or not. The former is the normal situation. Note the conciseness of the expression of the periodic computation of the regulation algorithm (when allowed to execute): It is a terminal star strongly aborted and immediatly restarted at each occurrence of `RC` (Regulation Clock). The body of the star is the `halt` process. Each preemption triggered by `RC`, causes the evaluation of `R(?SPD,?REF)` and the emission of the result conveyed by the signal `TVC`. Recall that, in ESTEREL, the question mark applied to a signal, returns its current value.

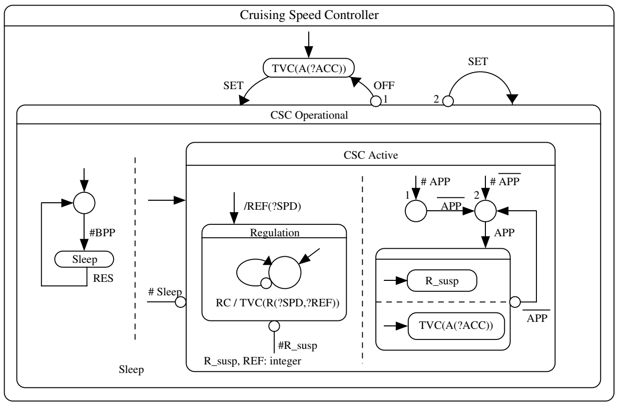

Figure 7: SYNCCHARTS of the Cruise Controller

5 SEMANTICS OF SYNCCHARTS

In this section we introduce a *process algebraic characterization of reactive behaviors*. This approach takes root in Robin Milner's works about "synchronous process algebras" [19]. G. Berry has integrated the preemption in the calculus [3]. We choose the terse algebraic presentation because it allows precise and concise expression of complex behaviors. Note that, like in PURE ESTEREL, we restrict SYNCCHARTS to pure signals.

5.1 Events and Processes

Let $\mathcal{I}$ be a set of (pure) input signals $i_1, i_2, \cdots$, and $\mathcal{O}$ a set of (pure) output signals $o_1, o_2, \cdots$ An *input event* is a subset I of $\mathcal{I}$ and an *output event* is a subset O of $\mathcal{O}$. All the signals in an event are simultaneously present. The sequence of input events $I_1, I_2, \cdots, I_n, \cdots$ at logical instants $1, 2, \cdots, n, \cdots$ is called an *input history*. The sequence of output events $O_1, O_2, \cdots, O_n, \cdots$ at the same instants is an *output history*.

A synchronous model of sort $\{\mathcal{I}, \mathcal{O}\}$ maps an input history into an output history:

$$B : I^* \longrightarrow O^*$$

Let $\mathbb{S}$ be a set of signals and $\perp$ be a distinguished element of $\mathbb{S}$; $\perp$ stands for a signal which is *never present* (the never occurring signal).

Definition 1 (Process) *Let p, n, q be processes on $\mathbb{S}$, $s, t \in \mathbb{S}$, and $s' \in (\mathbb{S} - \{\perp\})$. Then*

1. 0 is a process on $\mathbb{S}$ *(null)*
2. s' is a process on $\mathbb{S}$ *(emission)*
3. $p \mid q$ is a process on $\mathbb{S}$ *(parallel)*
4. $p*$ is a process on $\mathbb{S}$ *(loop)*
5. $p \setminus s'$ is a process on $\mathbb{S}$ *(restriction)*
6. $p\ [t/s']$ is a process on $\mathbb{S}$ *(renaming)*
7. $s \supset p$ is a process on $\mathbb{S}$ *(suspension)*
8. $s \nearrow p \rhd n, q$ is a process on $\mathbb{S}$ *(abortion)*

The first process is useful to build derived constructors. The next four constructors are classical imperative constructors. The sixth constructor is usual in process calculus. Note that $\perp$ cannot be renamed, but any signal can be renamed into $\perp$. The last two constructors are typically reactive.

With each process p, we associate three disjoint subsets of $\mathbb{S}$: $\mathcal{I}_p, \mathcal{O}_p, \mathcal{L}_p$ respectively called the input, the output, the local sorts of p. See Annex.A for details.

Definition 2 (Event) *An event E is a subset of $\mathbb{S}$ without the never occurring signal: $E \subseteq \mathbb{S} - \{\perp\}$. Given a process p on $\mathbb{S}$, $I \subseteq \mathcal{I}_p$ is an input event, $O \subseteq \mathcal{O}_p$ is an output event.*

5.2 Semantics

The semantics is expressed by a behavioral semantics. The behavior of a process is a deterministic mapping from input sequences to output sequences. A reaction is interpreted as a process rewriting. Given a process p and an input sequence $I_1, I_2, \cdots, I_n, \cdots$, the output sequence $O_1, O_2, \cdots, O_n, \cdots$ is computed as a chain of individual reactions:

$$p = p_1 \xmapsto[O_1]{I_1} p_2 \xmapsto[O_2]{I_2} \cdots p_n \xmapsto[O_n]{I_n} p_{n+1} \cdots$$

A transition $p_n \xmapsto[O_n]{I_n} p_{n+1}$ represents a single reaction. The $p \xmapsto[O]{I} p'$ relation is defined using an auxiliary relation $p \xrightarrow[E]{E_p,\, b} p'$ defined by structural induction over p. E is the set of signals that p sees as being present, E_p is the set of signals that p emits when receiving E, and b is a Boolean (termination bit) such that $b = \texttt{tt}$ if p terminates and $b = \texttt{ff}$ otherwise (p is said to wait).

The *broadcasting invariant* $E_p \subseteq E$ must be maintained during all the derivations.

Given a process p, an input event I:

$$p \xmapsto[I]{O} p' \text{ iff } p \xrightarrow[I \cup O]{O,\, b} p' \text{ for some } b$$

Rewriting rules associated with the basic processes are given in Annex.B.

Remark: Our set of operators is not minimal. For instance, "0" could have been defined as "$(s \backslash s)$". However, taking 0 as a primitive is simpler than deriving it from the somewhat complex restriction.

5.3 Derived Constructors

For convenience and for compatibility with ESTEREL, new constructors are derived from the previous ones:

Derived Imperative Constructors

1	(pause)
$p; q$	(sequence)
$s?p, q$	(conditional)

They can be defined as:

$$\begin{aligned} 1 &\equiv (s \mid (s \supset 0)) \setminus s \\ p; q &\equiv \perp \nearrow p \rhd q, 0 \\ s?p, q &\equiv s \nearrow (s \supset 0) \rhd q, p \end{aligned}$$

1 waits for the next instant. $p; q$ executes p and then q, in sequence. The conditional $s?p, q$ executes either p or q according to the presence or the absence of s.

Derived Reactive Constructors

They stand for specific preemptions.

$$
\begin{array}{lcll}
p : s\mathrel{\cdot\!>} q & \equiv & s \nearrow p \rhd 0, q & \text{(weak abortion)} \\
p : s\mathrel{\cdot\!\gg} q & \equiv & (s\mathrel{\cdot\!\supset} p) : s\mathrel{\cdot\!>} q & \text{(strong abortion)} \\
s \mathrel{\cdot\!\!\Rightarrow} p & \equiv & (1*) : s\mathrel{\cdot\!>} p & \text{(trigger)}
\end{array}
$$

Comments: The strong abortion is derived from both suspension ($\cdot\supset$) and weak abortion ($\cdot>$). That is the reason why, in SyncCharts, we have chosen the symbol ($\circ\!\!\rightarrow$) for strong abortion; it is an hybrid of $-\!\circ$ (suspension) and $\longrightarrow$ (weak abortion). Because of the suspension of p by s, the strong abortion prevents p from executing at the instant when it is preempted.

Delayed operators

Up to now, we have considered immediate and future occurrences of signals, often only strict future occurrences are desired. Suspension and abortion operators have their "delayed" counterparts:

$$
\begin{array}{lcc}
\delta_s^d & \equiv & \Big(1; (s?d, 0)\Big)* \\
s \Rightarrow p & \equiv & 1; (s \mathrel{\cdot\!\!\Rightarrow} p) \\
s \supset p & \equiv & \Bigg(\bigg(\Big(((d\mathrel{\cdot\!\supset} p)\,; t) \mid \delta_s^d\Big) \setminus d\bigg) : t\mathrel{\cdot\!>} 0\Bigg) \setminus t \\
p : s > q & \equiv & \Bigg(\bigg(\Big((s \Rightarrow d) : d\mathrel{\cdot\!>} 0\Big) \mid (p : d)\bigg) : d\mathrel{\cdot\!>} q\Bigg) \setminus d \\
p : s \gg q & \equiv & (s \supset p) : s > q
\end{array}
$$

Starting with the above mentioned operators, it is possible to derive new ones, easier to use and with a strictly defined semantics. For example, we have introduced the *generalized termination* [18]: A process can be aborted in several ways, leading to different processes. In order to preserve a deterministic behavior, triggering conditions are evaluated according to a priority ordering. We denote the generalized termination of p by: "$p : \sigma_1\, \rho_1\, q_1, \cdots, \sigma_n\, \rho_n\, q_n$" where $p, q_1, \cdots, q_n$ are processes, $\sigma_1, \cdots, \sigma_n$ are compound signals, and $\rho_1, \cdots, \rho_n \in \{\cdot>, >, \cdot\gg, \gg\}$. For at most one j, $\sigma_j\, \rho_j\, q_j$ may be replaced by $\rhd\ \ q_j$, standing for the normal termination. The priority is decreasing from left to right.

The behavior of every component of SyncCharts has been expressed with the above algebra. For instance, the generalized termination has been used to formally characterize the way of leaving a star. Interested readers are urged to refer to the technical report devoted to SyncCharts [18].

6 CONCLUSION

In this paper, we have addressed the problem of *Reactive System Modeling*. We have focused on the control-dominated systems, and we have advocated the use of the *Synchronous Approach.*

The Synchronous Approach

The synchronous paradigm leads to an elegant, rigourous and powerful abstraction of reactive behaviors. The *Zero-Delay* hypothesis is the cornerstone of this approach. Augmented with other hypotheses, like the instantaneous broadcasting of signals, synchrony is very convenient to deal with (logical) time. The notion of *preemption*, often ignored or, at best, poorly treated by the classical approaches, is raised to the rank of a first-class concept, orthogonal to communication and concurrency. Thanks to the synchronous hypotheses, reactive models compose very well in a deterministic way. Instantaneity of reactions may induce some surprising behaviors. We have explained them by analogy with *logical circuits.*

A synchronous description may be textual or graphical. The second goal of our contribution was to introduce a new synchronous graphical model: SyncCharts. This model adopts many features of StateCharts. SyncCharts have been especially tailored to support the various forms of preemption in an unambiguous way, what StateCharts cannot easily do. The semantics of SyncCharts relies on a process algebra, fully compatible with that of Esterel. Thus, SyncCharts allow insertion of Esterel code as annotations, and can be automatically translated into Esterel programs.

Applications of SyncCharts

We have used SyncCharts in specification and in effective programming of small applications. Our applications are mostly control-dominated systems. For instance, we have specified the behavior of a part of a F.M.S [20]: A SyncChart expresses the changes in functioning modes (a fully operational pipe-lined operating mode, and a degraded sequential mode). Modechart [21] and augmented StateCharts introduced in [22] would have been used, as well. The former is a specification language for real-time systems that emphasizes the specification of absolute properties of systems. The latter addresses the requirements specification for process-control systems. Both are ambitious projects, applied to large scale systems. Both rely on enhanced StateCharts, but none deals with preemptions in a so precise way as SyncCharts do.

Another potential use of SyncCharts is in object-oriented systems. Harel and Gery [23] specify the behavior of a class in an O-chart (a hierarchical OMT-like representation) by a controlling statechart. We have used SyncCharts in the same purpose [24].

Perpectives

SyncCharts are at the prototype stage: if their semantics is well-founded, their environment is still to be implemented. Many research groups in Europe participate to the *Synchron* project which aims at developing the "synchronous platform". This platform supports synchronous languages and models, interfaces to model checkers, simulator generators, and code generators. Our team "SPORTS" (Synchronous Programming Of Real Time Systems) has already contributed to this objective by providing tools dedicated to Sequential Function Charts [25]. SYNCCHARTS should be our next contribution.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9):1270–1282, September 1991.

[2] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Amsterdam, 1993.

[3] G. Berry. Preemption in concurrent systems. *Proc FSTTCS, Lecture notes in Computer Science*, 761:72–93, 1992.

[4] IEC, Genève (CH). *Preparation of Function Charts for control systems*, december 1988. International standard IEC 848.

[5] D. Harel. STATECHARTS: A visual formalism for complex systems. *Science of computer programming*, 8:231–274, 1987.

[6] M. von der Beeck. A comparison of STATECHARTS variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. FTRTFT'94, Springer-Verlag, 1994.

[7] F. Maraninchi. ARGOS*: un langage graphique pour la conception, la description et la validation des systèmes réactifs.* PhD thesis, Université Joseph Fourier, Grenoble I, Janvier 1990.

[8] C. André and M-A. Péraldi. Synchronous programming: Introduction and application to industrial process control. In 7^{th} *Annual European Computer Conference*, pages 461–470, Evry (France), May 1993. IEEE.

[9] W.A. Halang and K.M. Sacha. *Real-Time Systems: Implementation of Industrial Computerised Process Automation.* World Scientific, Singapore, 1992.

[10] J. Zahnd. *Machines Séquentielles*, volume XI of *Traité d'Electricité*. Editions Georgi, Suisse, 1980.

[11] G. Berry. A hardware implementation of pure ESTEREL. Miami, January 1991. ACM Workshop on Formal Methods in VLSI Design.

[12] F-X. Fornari. *Optimisation du contrôle et implantation en circuits de programmes Esterel.* PhD thesis, Université de Nice-Sophia Antipolis, Mars 1995.

[13] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of LUSTRE. volume 600 of *Lecture Notes in Computer Science*, pages 195–208, DePlasmolen (NL), June 1991. REX Workshop on real-Time: Theory and Practice, Springer-Verlag.

[14] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[15] Charles André and Daniel Gaffé. Sequential function charts: a synchronous point of view. Technical Report RR 95–08, I3S, Sophia-Antipolis, France, february 1995.

[16] F. Boussinot and R. de Simone. The SL synchronous language. Technical Report RR 2510, INRIA, March 1995.

[17] G. Berry. The constructive semantics of PURE ESTEREL. Technical Report Not yet published, INRIA, December 1995.

[18] Charles André. Syncccharts: a visual representation of reactive behaviors. Technical Report RR 95–52, I3S, Sophia-Antipolis, France, October 1995.

[19] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3), July 1983.

[20] C. André, J.C. Gentina, and L. Kermad. Approche synchrone des modes de marche et d'exploitation. In *Modélisation des Systèmes Réactifs*, pages 259–266, Brest, Mars 1996. Afcet.

[21] F. Jahanian and A.K. Mok. Modechart: A specification language for real-time systems. *IEEE transaction on Software Engineering*, 20(12):933–947, December 1994.

[22] N.G. Levenson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE transaction on Software Engineering*, 20(9):684–707, September 1994.

[23] D. Harel and E. Gery. Executable objects modeling with statecharts. Technical Report CS94–20, (Rev. August 1995), Weizmann Institute of Science, September 1994.

[24] C. André, F. Boulanger, M.A. Péraldi, J.P. Rigault, and G. Vidal-Naquet. Objets et programmation synchrone. In *Modélisation des Systèmes Réactifs*, pages 55–62, Brest, Mars 1996. Afcet.

[25] C. André, H Boufaïed, D. Gaffé, and J.P. Marmorat. Environnement pour la programmation synchrone des systèmes réactifs. In *Real-Time & Embedded Systems (RTS&ES'96)*, pages 27–41, Paris (France), Janvier 1996. teknea.

A Sort Transformations

Let p be a process on $\mathbb{S}$. Let $\mathcal{I}_p, \mathcal{O}_p, \mathcal{L}_p$ be its input, output, and local sorts. Two auxiliary sets are introduced:

$$\mathcal{X}_p = \mathcal{I}_p \cup \mathcal{O}_p \quad \text{and} \quad \mathcal{S}_p = \mathcal{I}_p \cup \mathcal{O}_p \cup \mathcal{L}_p$$

$\mathcal{X}_p$ is the interface set of p, $\mathcal{S}_p$ is the sort of p. These sets are defined inductively as follows:

For each operation:

$$\mathcal{L}_{pp} = \text{if } pp = p \setminus s \text{ then } \mathcal{L}_p \cup \{s\} \text{ else } \emptyset$$

$$\mathcal{I}_{pp} = \mathcal{X}_{pp} - \mathcal{O}_{pp} \text{ and } \mathcal{S}_{pp} = \mathcal{X}_{pp} \cup \mathcal{L}_{pp}$$

pp	$\mathcal{X}_{pp}$	$\mathcal{O}_{pp}$
0	$\emptyset$	$\emptyset$
s	$\{s\}$	$\{s\}$
$p \mid q$	$\mathcal{X}_p \cup \mathcal{X}_q$	$\mathcal{O}_p \cup \mathcal{O}_q$
$p*$	$\mathcal{X}_p$	$\mathcal{O}_p$
$p \setminus s$	$\mathcal{X}_p - \{s\}$	$\mathcal{O}_p - \{s\}$
$p\ [t/s]\ (s \in \mathcal{X}_p)$	$\mathcal{X}_p\ [t/s]$	$\mathcal{O}_p\ [t/s]$
$s \supset p$	$\mathcal{X}_p \cup \{s\}$	$\mathcal{O}_p$
$s \nearrow p \rhd n, q$	$\mathcal{X}_p \cup \mathcal{X}_q \cup \mathcal{X}_n \cup \{s\}$	$\mathcal{O}_p \cup \mathcal{O}_q \cup \mathcal{O}_n$

Note that our renaming is restrictive: only an input or output signal of a process can be renamed, not a local signal.

B Rewriting Rules

$$0 \xrightarrow[E]{\emptyset,\, \mathtt{tt}} 0 \qquad \textit{(null)}$$

$$s \xrightarrow[E]{\{s\},\, \mathtt{tt}} 0 \qquad \textit{(emission)}$$

$$\frac{p \xrightarrow[E]{E_p,\, b_p} p' \qquad q \xrightarrow[E]{E_q,\, b_q} q'}{p \mid q \xrightarrow[E]{E_p \cup E_q,\, b_p \wedge b_q} p' \mid q'} \qquad \textit{(parallel)}$$

$$\frac{p \xrightarrow[E]{E_p,\, \mathtt{ff}} p'}{p * \xrightarrow[E]{E_p,\, \mathtt{ff}} \bot \nearrow p' \rhd (p*), 0} \qquad \textit{(loop)}$$

$$\frac{p \xrightarrow[E \cup \{s\}]{E_p,\, b} p' \qquad s \in E_p}{p \setminus s \xrightarrow[E]{E_p - \{s\},\, b} p' \setminus s} \qquad \textit{(restr1)}$$

$$\frac{p \xrightarrow[E - \{s\}]{E_p,\, b} p' \qquad s \notin E_p}{p \setminus s \xrightarrow[E]{E_p,\, b} p' \setminus s} \qquad \textit{(restr2)}$$

$$\frac{p \xrightarrow[E]{E_p,\, b} p' \qquad s \in \mathcal{X}_p}{p\ [t/s] \xrightarrow[E\ [t/s]]{E_p\ [t/s],\, b} p'\ [t/s]} \qquad \textit{(renam)}$$

$$\frac{s \in E}{s \supset p \xrightarrow[E]{\emptyset,\, \mathtt{ff}} s \supset p} \qquad \textit{(susp1)}$$

$$\frac{s \notin E \qquad p \xrightarrow[E]{E_p,\, \mathtt{tt}} p'}{s \supset p \xrightarrow[E]{E_p,\, \mathtt{tt}} 0} \qquad \textit{(susp2)}$$

$$\frac{s \notin E \qquad p \xrightarrow[E]{E_p,\, \mathtt{ff}} p'}{s \supset p \xrightarrow[E]{E_p,\, \mathtt{ff}} s \supset p'} \qquad \textit{(susp3)}$$

$$\frac{p \xrightarrow[E]{E_p,\, \mathtt{tt}} p' \qquad n \xrightarrow[E]{E_n,\, b} n'}{s \nearrow p \rhd n, q \xrightarrow[E]{E_p \cup E_n,\, b} n'} \qquad \textit{(abort1)}$$

$$\frac{s \in E \qquad p \xrightarrow[E]{E_p,\, \mathtt{ff}} p' \qquad q \xrightarrow[E]{E_q,\, b} q'}{s \nearrow p \rhd n, q \xrightarrow[E]{E_p \cup E_q,\, b} q'} \qquad \textit{(abort2)}$$

$$\frac{s \notin E \qquad p \xrightarrow[E]{E_p,\, \mathtt{ff}} p'}{s \nearrow p \rhd n, q \xrightarrow[E]{E_p,\, \mathtt{ff}} s \nearrow p' \rhd n, q} \qquad \textit{(abort3)}$$